CERIAS Tech Report 2006-22

ENABLING INTERNET WORMS AND MALWARE INVESTIGATION AND DEFENSE USING VIRTUALIZATION

by Xuxian Jiang

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

ENABLING INTERNET WORMS AND MALWARE INVESTIGATION AND DEFENSE USING VIRTUALIZATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Xuxian Jiang

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2006

Purdue University

West Lafayette, Indiana

To my wife *Xining Wang*

ACKNOWLEDGMENTS

It is a daunting task for me to enumerate, let alone repay, all those to whom I am indebted for their great assistance during my years at Purdue. In the following, I will mention a few despite inevitable omissions.

First, I would like to thank my major advisor, Professor Dongyan Xu, in providing an energizing research environment and patiently motivating and supporting me during my graduate study at Purdue. Professor Xu has touched almost every aspect of my life in a positive way and I could not have asked for a more supportive and engaging mentor.

Second, I would like to thank Professors Eugene H. Spafford, Mikhail ("Mike") Atallah, Ninghui Li, Tony Hosking, and David K Y Yau for their time and efforts serving on my Ph.D. thesis committee and giving me valuable advice. In particular, I am deeply indebted to Professor Spafford for his great shepherding and detailed feedbacks throughout my Ph.D. research. I would also like to thank Professors Xiaojun Lin and Ninghui Li for their constructive suggestions to improve my presentation and Professor Cristina Nita-Rotaru for kindly offering me an opportunity as a CERIAS seminar speaker. All of your support and guidance have significantly helped me make research progress and advance my professional career.

West Lafayette is a nice and quiet place without much distraction. However, daily life for young graduate students such as myself would be quite mundane were it not for the constant interactions with my office mates, colleagues, and friends here. Yu (Jerry) Dong, Heung-Keung (Johnny) Chai, Wu Yan, Paul Ruth, Aaron Walters, Florian Buchholz, Jen-Yeu Chen, Gang Ding, Junghwan Rhee, and Ryan Riley are great friends and I greatly enjoy our time together. Our spontaneous and stimulating discussions on various topics

from time to time provided much-needed inspiration and laughter, beneficial to both my work and life.

I am indebted to my colleagues in industry, especially Yi-Min Wang, Helen J. Wang, Shuo Chen, and Doug Beck at Microsoft Research and Rong N. Chang, Christopher Ward, Melissa J. Buco, and Laura Z. Luan at IBM Research, for providing me with an avenue of technical exploration outside the confines of Purdue and exposing me to the commercial realities of industry research. I hope you found our work together as rewarding as I did.

William J. Gorman, Amy Ingram, Mike Motuliak, Linda Byfield, and all other staff members of the Department of Computer Science also deserve my gratitude. I still remember Dr. Gorman opened the door for me one weekend when I locked myself out of my office and left my interview materials inside. Amy patiently answered course registration questions that I repeatedly asked every semester during the last three years. Mike cleaned up my laptop monitor many times and Linda helped me fill out numerous travel forms. I appreciate all of your help!

Finally, I can not over-emphasize the importance of the persistent support and warm encouragement from my loving and beautiful wife Xining. Also, I must admit that I enormously enjoyed the distraction from my two kids – Matthew and Grace, ever since they were born.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

ABSTRACT xii

1 Introduction 1

1.1 Background and Problem Statement 1

1.2 Dissertation Contributions 3

1.3 Terminology 4

1.4 Dissertation Organization 5

2 An Integrated Framework for Malware Capture, Investigation, and Defense: An Overview 7

2.1 Framework Overview 7

2.1.1 The Front-End for Malware Capture 8

2.1.2 The Back-End for Malware Investigation 8

2.1.3 Malware Defense Mechanisms 9

3 Virtualization-Based Honeyfarm for Malware Capture and Detention 10

3.1 Introduction 10

3.2 Honeypots and Collapsar 11

3.2.1 Collapsar: Visions and Challenges 13

3.3 Architecture of Collapsar 14

3.3.1 Functional Components 16

3.3.2 Assurance Modules 17

3.4 Implementation of Collapsar 18

3.4.1 Traffic Redirection 18

3.4.2 Traffic Dispatching 20

Page

3.4.3 Virtual Honeypots . . . 21
3.4.4 Assurance Modules . . . 22
3.5 Performance Measurement . . . 23
3.6 Experiments with Collapsar . . . 25
3.6.1 Environment Setup . . . 26
3.6.2 Server-Side Honeypot Incidents . . . 26
3.6.3 Client-Side Honeypot Incidents . . . 32
3.6.4 Attack Correlation . . . 36
3.7 Related Work . . . 38
3.8 Summary . . . 40
4 Virtual Playground for Internet Worms and Malware Investigation . . . 41
4.1 Introduction . . . 41
4.2 Overview of vGround . . . 44
4.2.1 Key vGround Enabling Techniques . . . 45
4.2.2 Interface for vGround Configuration . . . 47
4.3 Design Details . . . 48
4.3.1 Full-System Virtualization . . . 48
4.3.2 Link-Layer Network Virtualization . . . 49
4.3.3 Virtual Node Optimization and Customization . . . 50
4.3.4 Worm Experiment Services . . . 51
4.4 Worm Experiments in vGrounds . . . 53
4.4.1 Target Network Space . . . 54
4.4.2 Propagation Patterns . . . 55
4.4.3 Detailed Exploitation Steps . . . 59
4.4.4 Malicious Payloads . . . 61
4.4.5 Advanced Worm Experiments . . . 62
4.5 Limitations and Extensions . . . 63
4.6 Related Work . . . 64

Page

4.7 Summary . . . 66
5 Characterizing Self-Propagating Worms with Behavioral Footprinting . . . 67
5.1 Introduction . . . 67
5.2 A Case for Behavioral Footprinting . . . 68
5.2.1 A Staged View of Worm Infections . . . 69
5.2.2 Example I: the MSBlast Worm . . . 70
5.2.3 Example II: the Lion Worm . . . 71
5.2.4 Behavioral Footprinting: a New Dimension of Worm Profiling . . . 72
5.3 Behavioral Footprint Representation and Extraction . . . 73
5.3.1 Behavioral Phenotype and Footprint . . . 73
5.3.2 Pairwise Alignment Algorithm . . . 76
5.3.3 Phylogenetic Tree Algorithm . . . 78
5.4 Evaluation . . . 81
5.4.1 Experiment Environment . . . 82
5.4.2 Extracting Behavioral Footprints . . . 82
5.4.3 Advantage of Behavioral Footprinting . . . 85
5.4.4 Robustness of Behavioral Footprinting . . . 87
5.5 Limitations . . . 92
5.6 Related work . . . 93
5.7 Summary . . . 94
6 Tracking Malware Break-ins and Contaminations with Provenance-Aware Process Coloring . . . 96
6.1 Introduction . . . 96
6.2 Process Coloring Approach . . . 98
6.2.1 Initial Coloring . . . 98
6.2.2 Color Diffusion Model . . . 100
6.2.3 Log Collection . . . 103
6.3 Implementation . . . 104

Page

6.3.1 Process Color Setting 104
6.3.2 Color Diffusion 106
6.3.3 Log Collection 107
6.4 Evaluation 108
6.4.1 Evaluation of Run-Time Overhead 108
6.4.2 Experiments with Real-World Worms 109
6.5 Other Applications and Possible Attacks 119
6.5.1 Other Applications 119
6.5.2 Possible Attacks and Countermeasures 120
6.6 Related Work 122
6.7 Summary 124
7 Conclusion and Future Work 125
7.1 Conclusion 125
7.2 Future Work 126
LIST OF REFERENCES 128
VITA 139

LIST OF TABLES

Table Page

5.1 Characterizing self-propagating worms with their behavioral footprints . . 84

5.2 Worm detection with content fingerprints 86

5.3 Snort signatures for the Slapper worm 88

6.1 A simplified color diffusion model 101

6.2 LMBench results showing low process coloring overhead 109

6.3 Statistics of process coloring log in three worm experiments 110

LIST OF FIGURES

Figure		Page
2.1	An integrated framework for malware capture, investigation, and defense	7
3.1	The Collapsar architecture supporting both honeyfarm and reverse honeyfarm	15
3.2	Comparing virtualization-incurred overhead: VMware vs. UML	24
3.3	Comparing Collapsar-incurred overhead: VMware vs. UML	24
3.4	Collapsar log of attacker activities after break-in via Apache	26
3.5	Attack via Apache leading to an *iroffer* backdoor (logged by Collapsar)	27
3.6	Collapsar log of attacker activities after break-in via Samba	29
3.7	Screenshot re-constructed from a honeypot snapshot: successful break-ins by MSBlast, Enbiei, and Nachi worms	31
3.8	Screenshots re-constructed from a honeypot snapshot after visiting a malicious URL	34
3.9	Malicious javascript code from the exploiting URL	35
3.10	Malicious javascript code from the exploiting URL	36
3.11	Collapsar log showing a possible stepping stone attack	37
3.12	Collapsar log showing an ICMP sweeping scan	38
4.1	A PlanetLab-based vGround for worm experiments	45
4.2	An example vGround specification	47
4.3	Illustration of link-layer network virtualization in vGround	49
4.4	Running *traceroute* inside a vGround on PlanetLab	50
4.5	Target network space of the Lion worm and the Slapper worm	53
4.6	Propagation of Slapper worm w/ *address-sweeping* (total: 1000 hosts)	56
4.7	Propagation of a Slapper worm variant w/ *island-hopping* (Total: 1000 hosts)	57
4.8	Exploitation details of the Lion worm	59

Figure Page

4.9 Exploitation details of the Slapper worm 60
4.10 Payloads of the Slapper worm 61
5.1 A staged view of a worm infection session 69
5.2 An infection session of the MSBlast/Windows worm 70
5.3 An infection session of the Lion/Linux worm 71
5.4 Global alignment with Needleman-Wunsch algorithm 77
5.5 An example alignment of multiple worm infection sequences 82
5.6 An example output of Sneeze 83
5.7 Behavioral footprints of the Ramen worm – a multi-vector worm 85
5.8 Worm detection and identification with behavioral footprints 87
5.9 The behavioral footprint of the Slapper worm 89
5.10 N-Gram analysis of the original Slapper worm 89
5.11 N-Gram analysis of a Slapper worm variant with encrypted transmission . 89
5.12 A phylogenetic tree built from 20 polymorphic behavioral sequences of the Slapper worm variant 91
6.1 Process coloring view of a system running multiple servers 99
6.2 A coloring diffusion view showing the initial break-in by the Slapper worm 102
6.3 Different hooking points to intercept system calls 105
6.4 Tamper-resistant log collection by positioning the interceptor on the system call virtualization path 107
6.5 A process coloring view of a vulnerable system *before* Lion infection . . . 111
6.6 Lion worm contamination reconstructed from “RED” log entries 113
6.7 A process coloring view of a Slapper-vulnerable system *before* infection . 114
6.8 Slapper worm contamination reconstructed from “RED” log entries 115
6.9 Log excerpt showing the first exploitation of the Slapper worm attempting to get the overwritable heap address in the vulnerable Apache server. BROWN log entries are not related. 116
6.10 A process coloring view of a Redhat 8.0 system running multiple servers *after* it is infected by the SARS worm 117
6.11 SARS worm contamination reconstructed from “RED” log entries 118

ABSTRACT

Jiang, Xuxian. Ph.D., Purdue University, August, 2006. Enabling Internet Worms and Malware Investigation and Defense Using Virtualization. Major Professor: Dongyan Xu.

Internet worms and malware remain a threat to the Internet, as demonstrated by a number of large-scale Internet worm outbreaks, such as the MSBlast worm in 2003 and the Sasser worm in 2004. Moreover, every new wave of outbreak reveals the rapid evolution of Internet worms and malware in terms of infection speed, virulence, and sophistication. Unfortunately, our capability to investigate and defend against Internet worms and malware has not seen the same pace of advancement.

In this dissertation, we present an integrated, virtualization-based framework for malware capture, investigation and defense. This integrated framework consists of a front-end and a back-end. The front-end is a virtualization-based honeyfarm architecture, called Collapsar, to attract and capture real-world malware instances from the Internet. Collapsar is the first honeyfarm that virtualizes full systems and enables centralized management of honeypots while preserving their distributed presence. The back-end is a virtual malware "playground," called vGround, to perform destruction-oriented experiments with captured malware or worms, which were previously expensive, inefficient, or even impossible to conduct.

On top of the integrated framework, we have developed a number of defense mechanisms from various perspectives. More specifically, based on the unique infection behavior of each worm we run in vGround, we define a behavioral footprinting model for worm profiling and identification, which complements the state-of-the-art content-based signature approach. We also develop a provenance-aware logging mechanism, called process coloring, that achieves higher efficiency and accuracy than existing systems in revealing malware break-ins and contaminations.

1 INTRODUCTION

1.1 Background and Problem Statement

Internet worms and malware remain a threat to the Internet, as demonstrated by a number of large-scale Internet worm outbreaks, such as the MSBlast worm in 2003 and the Sasser worm in 2004. Moreover, every new wave of outbreak reveals the rapid evolution of Internet worms and malware with respect to their infection speed, virulence, and sophistication. Examples of malware capabilities include infecting via multiple software vulnerabilities [2–4]; propagating to a large machine population in tens of seconds [9]; planting "backdoors" in victim machines [2, 3]; installing malicious programs for spam relay [4] or personal information collection [2]; and forming botnets among victim machines [10, 159].

Unfortunately, our capability to investigate and defend against Internet malware has not seen the same pace of advancement since the Code Red episode in mid-2001. The current approach of detection, characterization, and containment was developed to address the spread of file-based viruses, which mainly corrupt file contents, and has not changed significantly over the last five years. Furthermore, emerging Internet worms and malware are notably different from earlier file-based viruses in their infection methods, propagation means, and malicious payloads. As a result, advanced mechanisms are required to defend against emerging Internet worms and malware.

In this dissertation, we argue that our lack of thorough understanding of Internet worms and malware and of corresponding defense techniques is partially due to the absence of systematic experimental platform and scientific methodology for observing, investigating, and modeling Internet worms and malware. Such platform and the corresponding methodology should help answer the following questions: How to monitor the health of the Internet and generate timely attack alerts? Once an alert is generated, how to trace

the root cause of the threat? How to collect samples of new worms and malware in a timely fashion? After a sample is collected, how to safely and realistically reproduce its malicious behavior while avoiding real damages for timely investigation and accurate characterization? How to expose possibly hidden or obfuscated features in the new infection? Furthermore, we envision that new malware defense techniques can be developed based on results from in-depth malware observation and investigation.

In this dissertation, we propose an integrated, virtualization-based framework for malware capture, investigation and defense. Virtualization technology [1] has recently regained tremendous research interest with a unique system perspective by creating a level of indirection between physical resources and software systems. Such indirection provides new capabilities to address computer systems problems such as security, reliability, and resource provisioning. The focus of this dissertation is the use of virtualization technology for malware investigation and defense.

The proposed framework consists of a front-end and a back-end. The front-end is a virtualization-based honeyfarm architecture, called Collapsar, to attract and capture real malware attacks from the Internet. The back-end is a virtual malware "playground," called vGround, which enables destruction-oriented experiments with captured real-world malware or worms.

Based on the integrated framework, we have developed a number of defense mechanisms from various perspectives. More specifically, based on the unique infection behavior of each worm we run in vGround, we define a behavioral footprinting model for worm profiling and identification, which complements the content-based signature approach. We also develop a provenance-aware logging mechanism, called process coloring, that achieves higher efficiency and accuracy than existing log-based forensics systems in tracing malware break-ins and contaminations.

1.2 Dissertation Contributions

The contributions of this dissertation are three-fold: malware capture, malware investigation, and malware defense.

- **Malware capture** We have designed, implemented, and evaluated a virtualization-based honeyfarm architecture, Collapsar [11, 12], to capture real-world malware attacks from the Internet. Collapsar realizes the honeyfarm vision of distributed presence and centralized management of honeypots. Moreover, Collapsar supports both server-side honeypots and client-side honeypots [13]. Server-side honeypots run vulnerable server programs and passively wait for incoming attacks, while client-side honeypots act as vulnerable clients (e.g., running a vulnerable web browser) and actively crawl the Internet to be compromised by real-world malicious servers. Collapsar is the first virtualization-based honeyfarm system that supports both server-side and client-side honeypots.

- **Malware investigation** We have designed, implemented, and evaluated a virtualization based malware playground, vGround [14], to safely reproduce malware behavior. vGround is the first safe, scalable playground that can be used to unleash and observe real-world worms and malware in a confined, realistic virtual environment on top of a general-purpose shared infrastructure (e.g., a physical machine or a cluster). vGround enables destruction-oriented experiments with real-world malware or worms captured by the Collapsar front-end. Such experiments were previously expensive, inefficient, or even impossible to conduct.

- **Malware defense** Using Collapsar and vGround as an integrated experiment platform, we have developed a number of defense mechanisms [15, 16]. In this dissertation, we describe two new defense mechanisms, one for worm behavior profiling and one for malware forensics: (1) For worm profiling, we have defined a behavioral footprinting model [15] that complements the content-based signature model and therefore enriches a worm's profile for more accurate worm identification; (2) For

malware forensics, we have designed and implemented a provenance-aware logging mechanism called process coloring [16] to accurately and efficiently trace malware break-ins and contaminations.

1.3 Terminology

This section establishes terminology that is used throughout the rest of the dissertation. We inherit the same definitions for worm and virus by Eugene H. Spafford in 1989 [19]. The definition of honeypot is based on the definition by Lance Spitzner [20].

- **Worm** A worm is "a program that can run independently and can propagate a fully working version of itself to other machines". As noted in [19], "it is derived from the word *tapeworm*, a parasitic organism that lives inside a host and uses its resources to maintain itself."

- **Virus** A virus is "a piece of code that adds itself to other programs, including operating systems." It cannot run independently – it requires that its "host" program be run to activate it. As pointed out in [19], it has "an analog to biological viruses – those viruses are not considered alive in the usual sense; instead, they invade host cells and corrupt them, causing them to produce new viruses."

- **Rootkit** A rootkit is "a set of software tools or programs frequently used by an intruder after gaining access to a computer system." [5] It allows an intruder to access the victim's system without being noticed. A rootkit can intentionally conceal certain status of a running system, such as current running processes, existing files, or open network connections. Various rootkits exist for a variety of operating systems including Microsoft Windows, Linux, and Solaris.

- **Backdoor** A backdoor is "an undocumented way to get access to a computer system or the data it contains." [6] The backdoor is usually combined with a rootkit. For example, when a backdoor is being provided by a malicious process, a rootkit can be deployed to hide its existence from a legitimate system administrator.

- **Trojan** A trojan is “a malicious program that installs itself or runs surreptitiously on a victim’s machine.” [7] It does not run automatically – it requires that its “masqueraded” program be run to activate it. As such, “it may entice users into installing or executing by masking as another legitimate program.” [7]

- **Malware** Malware represents “malicious software,” a generic term “covering a range of malicious software programs to attack or degrade the intended use of a computer system or network.” [8] Types of malware include worms, viruses, rootkits, backdoors, and trojans.

- **Honeypot** A honeypot is a “security resource whose value lies in being probed, attacked, or compromised.” [20] It is also considered a trap set to detect or deflect attempts at unauthorized use of information systems. A honeypot usually consists of a computer, data or a network site that appears to be part of a network, but is actually isolated and protected, and usually contains information that would be of value to attackers. Based on different criteria, there exist various types of honeypots. A further classification will be described in Chapter 3.

- **Honeyfarm** A honeyfarm is a collection of honeypots that are deployed and administrated in the same location. Compared with multiple individual honeypots, a honeyfarm has the benefits of centralized honeypot management, distributed honeypot presence, and convenient attack correlation.

1.4 Dissertation Organization

This dissertation is organized into seven chapters, including this introductory chapter. Chapter 2 gives an overview of the integrated framework for Internet worm and malware capture, investigation, and defense. The design, implementation, and evaluation of the framework’s front-end and back-end will be presented in Chapter 3 and Chapter 4, respectively. Chapter 5 and Chapter 6 present two defense mechanisms: behavioral footprinting

and process coloring, which are developed and evaluated on top of the integrated platform. We make concluding remarks and outline future work in Chapter 7.

2 AN INTEGRATED FRAMEWORK FOR MALWARE CAPTURE, INVESTIGATION, AND DEFENSE: AN OVERVIEW

In this chapter, we present an overview of our integrated framework, followed by a brief description of its three key components and their relation.

2.1 Framework Overview

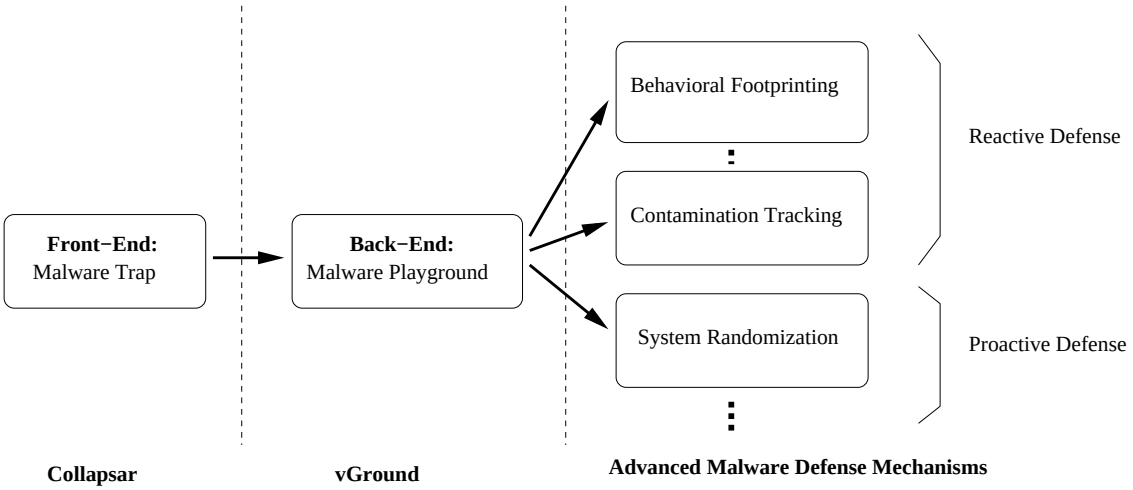


Figure 2.1. An integrated framework for malware capture, investigation, and defense

Figure 2.1 shows the overall organization of the integrated framework. This framework has three main components: (1) a honeyfarm front-end for malware capture (Collapsar), (2) a back-end playground for malware investigation (vGround), and (3) a suite of malware defense mechanisms.

2.1.1 The Front-End for Malware Capture

The front-end interacts with computers on the Internet and invites malware attacks. To capture attacks against multiple network domains, a common practice is to deploy honeypots in these domains. Unfortunately, this creates a conflict between attack detection coverage and honeypot management. A large number of honeypots deployed in different domains will achieve a wide attack coverage. However, they will make honeypot management challenging, especially considering the inherent security risks of honeypot operation and the need for expertise in honeypot monitoring and analysis [1].

Our front-end is a full-system virtualization-based honeyfarm architecture called Collapsar that resolves the above conflict. In Collapsar, honeypots are logically present in different physical production networks achieving wide attack coverage. However, these honeypots are physically hosted in a dedicated network and centrally managed by security experts. As a result, Collapsar achieves the seemingly conflicting goals of distributed presence and centralized management of honeypots. Virtualization technology plays a key role in achieving the scalability, confinement, and realism of Collapsar

Between its initial deployment in August 2003 and May 2006, Collapsar has captured a number of high-profile Internet worms and malware such as MSBlast (2003), Enbiei (2003), Welchia (2003), Sasser (2004), and Zotob (2005). Collapsar has also demonstrated feasibility of real-time distributed attack correlation and mining. Collapsar will be presented in Chapter 3.

2.1.2 The Back-End for Malware Investigation

After capturing an Internet worm, it is desirable to unleash it in an environment for close observation of its infection, contamination, and propagation behavior. Unfortunately, major challenges exist in realizing such a “malware playground,” that achieves

[1] Based on our experience, it is usually within hours or even minutes for a newly-deployed honeypot to be probed and compromised.

all of the following: fidelity, confinement, scalability, and experiment efficiency and convenience.

We address these challenges by developing a virtualization-based malware playground called vGround. A vGround is an all-software virtual environment dynamically created for a destruction-oriented malware experiment. It contains realistic end-hosts and network entities, all realized as virtual machines (VMs) connected and confined by a virtual network (VN). When running in a vGround, a worm's behavior, such as probing, exploitation, replication, and payload functions, can be fully revealed and recorded in the vGround. vGround will be presented in Chapter 4.

2.1.3 Malware Defense Mechanisms

With Collapsar and vGround, we create an integrated experiment platform that effectively keeps track of emerging Internet threats (e.g., worm outbreaks) and safely reproduces malware behavior for close observation and investigation. Based on the insights and observations obtained from this platform, we gain unique advantages in investigating malware defense mechanisms.

We have developed a suite of malware defense mechanisms. These mechanisms are either *reactive* by tracing malware break-ins and contaminations and generating behavior-based worm signatures or *proactive* in making existing systems more robust or even immune to malware infection mechanisms (e.g., code-injection attacks). In this dissertation, we focus on reactive malware defense mechanisms, to be presented in Chapter 5 and Chapter 6.

3 VIRTUALIZATION-BASED HONEYFARM FOR MALWARE CAPTURE AND DETENTION

3.1 Introduction

There has been an increase in the number and scale of Internet malware attack incidents from 2000 to 2006 [23]. This has motivated research efforts in developing systems and tools for capturing, monitoring, analyzing, and, ultimately, defending against Internet malware. Among the most notable approaches, the honeypot [24] has emerged as an effective tool for observing and understanding attackers' motivations, toolkits, and tactics. A honeypot, by nature, suspects every packet transmitted to/from it, enabling the collection of highly concentrated, low-noise datasets for network attack analysis.

However, honeypots are not panacea and suffer from a number of limitations. In this dissertation, we will address the following two limitations of independently operated honeypots. First, a single honeypot or multiple independently operated honeypots only provide limited local views of large-scale malware attacks. There is also a lack of coordination among honeypot operations in different networks, missing the opportunity of creating a wide diverse view for global malware attack monitoring, correlation, and trend prediction. Second, honeypot deployment has inherent security risks and requires non-trivial efforts in monitoring and data analysis. Security expertise is needed for safe and effective honeypot operations. However, such expertise is not widely available, making it necessary to resort to centralized honeypot management backed by special expertise and strict regulations.

It is challenging yet desirable to accommodate the two conflicting goals of honeypot deployment and operation: decentralized presence and centralized management. To address this challenge, we present Collapsar, a virtual machine (VM) based architecture for Internet worms and malware capture. A Collapsar center hosts and manages a large

number of honeypots in a local dedicated physical network. However, to attackers, these honeypots appear to be in different network domains. Hence, the two seemingly conflicting goals are achieved simultaneously by Collapsar. The logical distributed presence of honeypots provides a more global view of malware attacks, while the centralized physical location enables security experts to locally manage honeypots and collect, analyze, and correlate attack data pertaining to multiple production networks.

Collapsar realizes the *honeyfarm* vision, or more specifically *server-side* honeyfarm vision, where multiple server-side honeypots running vulnerable services (e.g., Apache web servers) are centrally operated while each of them virtually belongs to different network domains. Furthermore, Collapsar realizes our new vision of *reverse honeyfarm* or *client-side honeyfarm*, where multiple honeypots running vulnerable *client-side software* (e.g., web browsers) actively crawl the web to draw possible exploitations by malicious servers. For convenience, we will use the term *reverse honeyfarm* and *client-side honeyfarm* interchangeably. The client-side honeypots also have virtual presence in different network domains, while they are physically launched from the Collapsar center. For both server-side and client-side honeyfarms, Collapsar achieves three key advantages over individual honeypot systems: (1) distributed presence, (2) centralized management, and (3) convenient attack correlation and data mining.

The rest of this chapter is organized as follows: Section 3.2 presents the background of honeypots as well as the vision and challenges of Collapsar. The architecture of Collapsar is presented in Section 3.3, while the implementation details are described in Section 3.4. Section 3.5 evaluates Collapsar's performance. Section 3.6 presents several real-world attack incidents captured by our Collapsar prototype. Related work is presented in Section 3.7. Finally, Section 3.8 summarizes this chapter.

3.2 Honeypots and Collapsar

Honeypots can be classified by the level of interaction with attackers (can be either human or malware). This classification differentiates *high-interaction*, *medium-interaction*,

and *low-interaction* honeypots. High-interaction honeypots allow attackers to access full-fledged operating systems with few restrictions, although, for security reason, the surrounding environment may be restricted to confine any hazardous impact of honeypots. This is highly valuable because new vulnerabilities in real operating systems and applications can be brought to light [13, 28]. However, such value comes at the price of high risk and operator responsibility. Medium-interaction honeypots involve less risk but more restrictions than high-interaction honeypots. One example is the use of *jail* or *chroot* in a UNIX environment. Medium-interaction honeypots provide more functionalities than low-interaction honeypots, which are, on the contrary, easier to install, configure, and maintain. Low-interaction honeypots emulate a variety of services that the attackers can interact with.

Another classification criteria differentiates between *physical* and *virtual* honeypots. A physical honeypot is a real machine on the network, while a virtual honeypot is a virtual machine hosted in a physical machine. For example, *honeyd* [26] is a low-interaction virtual honeypot framework. In recent years, advances in virtual machine technologies have boosted the development and deployment of virtual honeypots. Virtual machine platforms such as VMware [21], User-Mode Linux (UML) [22], and Xen [29] enable high-fidelity emulation of physical machines and have been increasingly adopted to support virtual honeypots [24].

A new classification criteria distinguishes between *server-side* and *client-side* honeypots. Server-side honeypots are passive entities running vulnerable server-side software and they wait for attackers' contact and intrusion. Most current honeypot systems are server-side honeypots. Client-side honeypots are proactive entities running vulnerable client-side software and they initiate contact with servers on the Internet to get exploited (e.g., a vulnerable web browser getting exploited by a malicious web server). The client-side honeypot is unique in detecting possible exploitation of client-side software, a capability not provided by traditional server-side honeypots. Examples of client-side honeypot systems include the Strider HoneyMonkey exploit detection system [13] and the Honeyclient system [30].

3.2.1 Collapsar: Visions and Challenges

Honeypots in Collapsar can be categorized as *high-interaction* and *virtual*. Moreover, Collapsar supports both server-side and client-side honeypots. Different from individual honeypots, Collapsar honeypots are physically located in a dedicated local network but logically dispersed in multiple network domains. This property reflects the vision of *honeyfarm* [25]. However, there has been *no* realization of honeyfarm before Collapsar that uses high-interaction honeypots with detailed design, implementation, and real-world experiments. Moreover, we demonstrate that by using *high-interaction* honeypots, the honeyfarm vision can be more completely realized than using low-interaction honeypots or passive traffic monitors. Extending the honeyfarm vision, we further propose and realize the reverse honeyfarm vision. The reverse honeyfarm is different from the traditional honeyfarm in that it hosts client-side honeypots. Instead of passively waiting for attacks, client-side honeypots in Collapsar actively request services from servers on the Internet. To a server, requests from Collapsar appear to come from different network domains.

The development of Collapsar is more challenging than that of a stand-alone honeypot system. System fidelity requires honeypots to behave, from an attacker's point of view, as normal hosts in their associated network domains. From the perspective of Collapsar operators, the honeypots should be easy to configure, monitor, and manipulate for system manageability. To develop Collapsar, the following problems, common in both traditional honeyfarms and reverse honeyfarms, need to be addressed:

- How to redirect traffic? Traffic toward/from a honeypot should be transparently redirected between the target network and the Collapsar center without the attacker becoming aware of the redirection. Moreover, a virtual honeypot in the Collapsar center is expected to exhibit similar network configuration and behavior as the regular hosts in the same production network.

- What traffic to redirect? To achieve high fidelity, all traffic to a honeypot needs to be redirected, even if some traffic (such as broadcast) is not exclusively for the honeypot. However, redirection of all related traffic will incur considerable over-

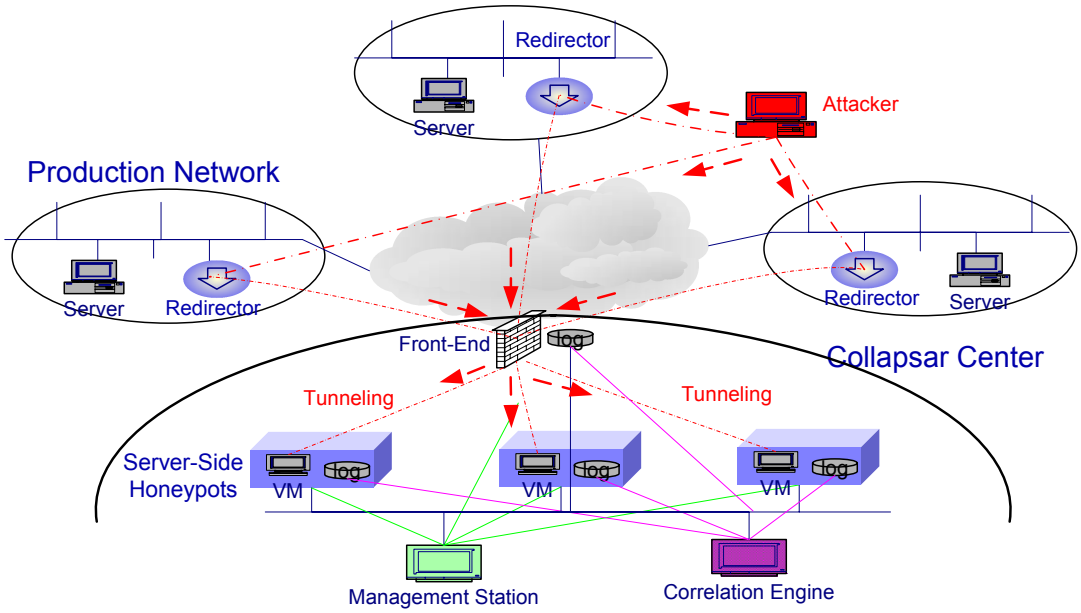
head. More seriously, some traffic may contain sensitive information that the attacker should not be receiving.

- When to stop an attack? Honeypots are designed to exhibit vulnerability and are expected to be attacked. However, the attack may cascade. A compromised honeypot can be used in another round of worm propagation or DDoS attack. Collapsar should detect and prevent such attacks before any real damage is done. However, simply blocking all outgoing traffic is not a good solution, because it will curtail the collection of evidence of the attacks, such as communication with other cohorts and the downloading of rootkits. The challenge is to decide the right time to say 'Freeze!' to the attacker.

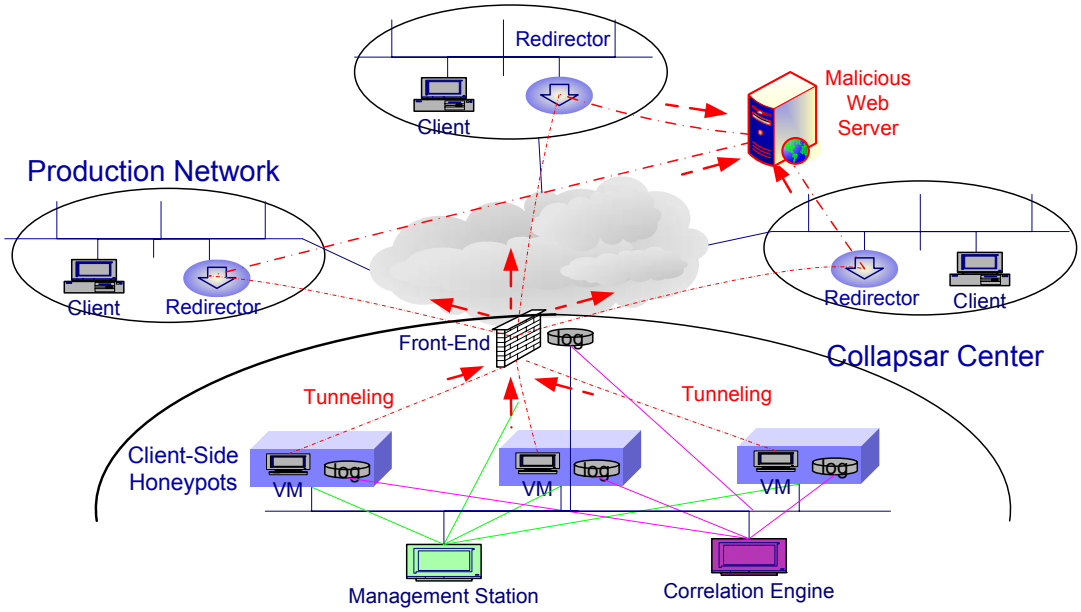
We present our solutions to the first problem. For the second and the third problems, we present Collapsar's components and mechanisms for the enforcement of traffic filtering and attack curtailing policies specified by Collapsar administrators. We in this dissertation do not address any specific policy and its impact. Instead, we focus on the architecture and mechanisms of Collapsar.

3.3 Architecture of Collapsar

The architecture of Collapsar is shown in Figure 3.1. Figures 3.1(a) and 3.1(b) show how Collapsar realizes the honeyfarm and reverse honeyfarm visions, respectively. Collapsar is comprised of three main functional components: the *redirector*, the *front-end*, and the *virtual honeypot* (VM). These components work together to achieve fidelity-preserving traffic redirection. Collapsar also includes the following assurance modules to capture, contain, and analyze the activities of attackers: the *logging module*, the *tarpitting module*, and the *correlation module*.



(a) A honeyfarm (hosting server-side honeypots) realized by Collapsar



(b) A reverse honeyfarm (hosting client-side honeypots) realized by Collapsar

Figure 3.1. The Collapsar architecture supporting both honeyfarm and reverse honeyfarm

3.3.1 Functional Components

3.3.1.1 Redirector

The redirector is a software component running on a designated machine in each participating production network. Its task is to forward relevant traffic to virtual honeypots in the Collapsar center. A redirector has three main functions: traffic capture, filtering, and diversion. Traffic capture involves the interception of all packets (including unicast and multicast packets) toward a honeypot. As the captured packets may contain sensitive information, traffic filtering needs to be performed according to rules specified by the network administrator. Finally, packets that have gone through the filter will be encapsulated and diverted to the Collapsar center by the traffic diversion function.

3.3.1.2 Front-end

The front-end is a gateway to the Collapsar center. It receives encapsulated packets from redirectors in different production networks, decapsulates the packets, and dispatches them to corresponding virtual honeypots in the Collapsar center. To avoid becoming a performance bottleneck, multiple front-ends may exist in a Collapsar center.

In the reverse direction, the front-end accepts outgoing traffic from the honeypots, and scrutinizes all packets with the help of assurance modules (to be described in Section 3.3.2) for attack stoppage. If necessary, the front-end will curtail the interaction with the attacker to prevent a compromised honeypot from attacking other hosts on the Internet. If a policy determines that continued interaction is allowed, the front-end will forward the packets back to their original redirectors, which will then redirect the packets into the network such that the packets appear to the remote attacker as originating from the target network.

3.3.1.3 Virtual Honeypot

Collapsar supports both server-side and client-side honeypots: Server-side honeypots accept packets coming from redirectors and behave as if they are hosts in the target pro-

duction network. Physically, the traffic between the attacker and the honeypot follows the path "attacker's machine → redirector → Collapsar front-end → honeypot." Logically, the attacker interacts *directly* with the honeypot. To achieve fidelity, the honeypot has the same network and system configuration as other hosts in the production network, including the default router, DNS servers, and mail servers. Client-side honeypots actively initiate service requests, which are relayed transparently by redirectors to malicious servers. Client-side honeypots appear to a malicious server as regular hosts running vulnerable client-side software in different production network domains.

Both types of honeypots in Collapsar run as virtual machines. Virtualization not only achieves resource-efficient honeypot consolidation, but also enables attack investigation capabilities such as tamper-proof logging, live image snapshooting, and dynamic honeypot creation and customization [31].

3.3.2 Assurance Modules

While the Collapsar functional components enable virtual distributed presence of honeypots, assurance modules provide necessary facilities for attack investigation and mitigation of associated risks.

3.3.2.1 Logging Module

Recording how an attacker exploits software vulnerabilities is critical to the understanding of exposed vulnerabilities as well as attack tactics and strategies [24]. All communications with honeypots are suspicious and need to be recorded. However, the traditional Network Intrusion Detection System (NIDS) based on packet sniffing may not be effective if the attack traffic is encrypted. It has become common for attackers to communicate with compromised hosts using encryption-enabled backdoors, such as trojaned *sshd* daemons. To log the details of such attacks without attackers tampering with the log, the logging module in each honeypot consists of sensors embedded in the honeypot's guest

OS as well as log storage in the underlying physical host. As a result, log collection and storage achieve high tamper-resistance.

3.3.2.2 Tarpitting Module

Deploying high-interaction honeypots is risky in that they can be used by the attacker as a platform to launch a second round of attack (e.g. worm propagation). To mitigate such risk, Collapsar's tarpitting module subverts attacks by (1) throttling out-going traffic from honeypots [32] by limiting the rate packets are sent (e.g. TCP-SYN packets) and (2) scrutinizing out-going traffic based on known attack signatures, and crippling detected attacks by invalidating malicious attack code [33].

3.3.2.3 Correlation Module

Collapsar provides opportunities to aggregate and mine log data for attack correlation, which an individual honeypot or multiple independently operated honeypots cannot offer. Such capability is supported by the correlation module. For example, the correlation module is able to detect network scanning by correlating simultaneous or sequential probing (ICMP echo requests or TCP-SYN packets) of honeypots that logically belong to multiple production networks. The correlation module can also be used to detect *on-going* DDoS attacks [34], worm outbreaks [35], and hidden attack networks such as IRC-based or peer-to-peer-based botnets created by certain worms.

3.4 Implementation of Collapsar

3.4.1 Traffic Redirection

There are two approaches to transparent traffic redirection: the router-based approach and the end-system-based approach. In the router-based approach, an intermediate router or the edge router of a network domain can be configured to activate the Generic Routing Encapsulation (GRE) [36, 37] tunneling mechanism to forward honeypot traffic to the Collapsar center. The approach has the advantage of high network efficiency. However, it

requires the privilege of router configuration. The end-to-end approach does not require access and changes to routers. Instead, it requires an application-level redirector in the target production network for forwarding packets between the attacker and the honeypot. In a cooperative environment such as a university campus, the router-based approach may be a more efficient option, while in an environment with multiple autonomous domains, the end-system-based approach may be adopted for easy deployment. In the following, we describe the design and implementation of the end-system-based approach.

To illustrate the end-system-based approach, let R be the default router of a production network, H be the IP address of the physical host where the redirector component runs, and V be the IP address of the honeypot as appearing to attackers. H, V, and an interface of R, say I_1, belong to the same network. When there is a packet addressed to V, router R will first receive it and then try to forward the packet based on its routing table. As address V appears in the same network as I_1, R will send the packet over I_1. To successfully forward the packet to V, R needs to know the corresponding MAC address of V in the ARP cache table. If the MAC address is not in the table, an ARP request packet will be broadcast to get the response from V. H will receive the ARP request. H knows that there is no real host with IP address V. To answer the query, H responds with its own MAC address, so that the packet to V can be sent to H and the redirector in H will then forward the packet to the Collapsar center. Note that one redirector can support the virtual presence of *multiple* honeypots in the same production network.

The redirector is implemented as a virtual machine running our enhanced version of UML. This approach adds considerable flexibility to the redirector because the VM is able to support policy-driven configuration for packet filtering and forwarding, and can be conveniently extended to support useful features such as packet logging, inspection, and in-line rewriting. The redirector has two virtual NICs: the *pcap/libnet* interface and the *tunneling* interface. The *pcap/libnet* interface performs the actual packet capture and injection. Captured packets will be echoed as input to the UML kernel. The redirector kernel acts as a bridge, and performs policy-driven packet inspection, filtering, and subversion. The *tunneling* interface tunnels the inspected packets transparently to the Collapsar center.

For communication in the opposite direction, the redirector kernel's *tunneling* interface accepts packets from the Collapsar center and moves them into the redirector kernel itself, which will inspect, filter, and subvert the packets from the honeypots, and re-inject the inspected packets into the production network through the *pcap/libnet* interface.

3.4.2 Traffic Dispatching

The Collapsar front-end is similar to a transparent firewall. It dispatches incoming packets from redirectors to their respective honeypots based on the destination field in the packet header. The front-end can also be implemented using UML, which creates another point for packet logging, inspection, and filtering.

Ideally, packets should be forwarded directly to the honeypots after dispatching. However, virtualization techniques in different VM platforms complicate this problem. To accommodate various VMs (especially those using VMware), the front-end will first inject packets into the Collapsar network via an injection interface. The injected packets will then be claimed by the corresponding virtual honeypots and be moved into the VM kernels via their virtual NICs. This approach supports commercial VMs (e.g., VMware and Virtual PC) without any modification. However, it incurs additional overhead (as shown in Section 3.5). Furthermore, it causes undesirable *cross-talk* between honeypots that logically belong to different production networks. Cross-talk may decrease the fidelity of Collapsar. A systematic solution to this problem requires a slight modification to the virtualization implementation, especially the NIC virtualization. Unfortunately, modifying the VM requires access to the VM's source code. With open-source VM implementations such as UML, the injection interface of the front-end can be modified to feed packets directly into the VM (honeypot) kernels. As shown in Section 3.5, considerable performance improvement can be achieved by this technique.

3.4.3 Virtual Honeypots

The current Collapsar prototype supports virtual honeypots based on both VMware and UML. Other VM platforms such as Xen [29], Virtual PC [38], and UMLinux [39] may also be supported in the future.

VMware is a commercial system and one of the most mature and versatile VM platforms. A key feature is the ability to support various commodity operating systems and to take snapshots of live virtual machine images. Support for commodity operating systems provides more diverse views of network attacks, while image snapshot generation and restoration add to the convenience of forensic analysis. As mentioned in Section 3.4.2, the network interface virtualization of VMware is not readily compatible with Collapsar design. More specifically, in a Linux platform, VMware creates a special *vmnet*, which emulates an inner bridge. A VMware-based virtual machine injects packets directly into the inner bridge, and receives packets from the inner bridge. A special host process is created to be attached to the bridge and acts as an agent to forward packets between the local network and the inner bridge. The ability to read packets from the local network is realized by a loadable kernel module named *vmnet.o*, which installs a callback routine registering for all packets on a specified host NIC via the *dev_add_pack* routine. The packets will be re-injected into the inner bridge. Meanwhile, the agent will read packets from the inner bridge and call the *dev_queue_xmit* routine to directly inject packets to the specified host NIC. It is possible to re-write the special host process to send/receive packets directly to/from the Collapsar front-end so that we can avoid packet injection and capture overhead, incurred in both the front-end and the special host process. However, this solution requires modifications to VMware.

UML is an open-source VM platform that runs directly in the unmodified *user space* of the host OS. Processes within a UML (the guest OS) are executed in the virtual machine in the same way as they would be executed in a native Linux machine. Leveraging the capability of *ptrace*, a special thread is created to intercept the system calls made by any process thread in the UML kernel, and redirects them to the guest OS kernel. Meanwhile,

the host OS has a separate *kernel space*, eliminating any security impact caused by the individual UMLs. We enhanced UML's network virtualization implementation so that each packet from the front-end can be immediately directed to the virtual NIC of a UML-based VM. This technique not only avoids unnecessary packet capture and re-injection (as in VMware) but also eliminates the *cross-talk* between honeypots in the Collapsar center.

3.4.4 Assurance Modules

Logging modules are deployed in multiple Collapsar components including redirectors, front-ends, and honeypots. Transparent to attackers, logging modules in different locations record attack-related information from *different* view points. Simple packet inspection tools, such as tcpdump [40] and snort [41] are able to record plain traffic, while embedded sensors inside the honeypot (VM) kernel are able to uncover an attacker's encrypted communications. In section 3.6.2, we will present details of several attack incidents demonstrating the power of in-kernel logging. The in-kernel logging module in VMware-based honeypots leverages an open-source project named *sebek* [42], while the in-kernel logging for UML-based honeypots is performed by *kernort* [43], a kernelized snort [41].

Tarpitting modules are deployed in both the front-end and redirectors. The modules perform in-line packet inspection, filtering, and rewriting. Currently, the tarpitting module is based on *snort-inline* [33], an open-source project. It can limit the number of out-going connections within a time unit (e.g., one minute) and can also compare packet content with known attack signatures in the *snort* package. Once malicious code is identified, the packets will be rewritten to invalidate their functionality.

The Collapsar center provides a convenient venue to perform correlation-based attack analysis such as wide-area DDoS attacks or stepping stone attacks [44]. The current prototype is capable of attack correlation based on simple heuristics and association rules. The correlation module can be extended to support more complex event correlation and

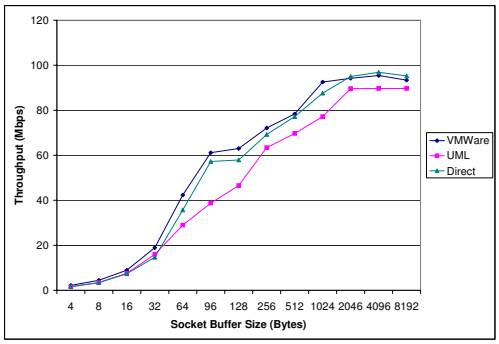
data mining algorithms, enabling the detection of non-trivial attacks such as low and slow scanning and hidden botnets.

3.5 Performance Measurement

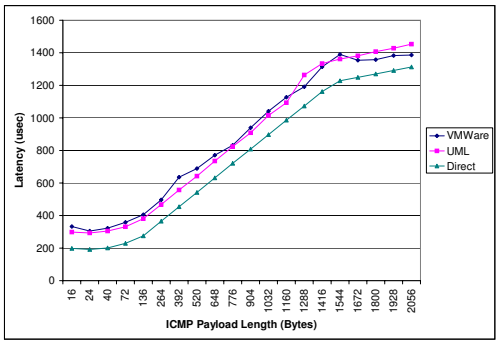
The VM technology provides effective support for high-interaction honeypots. However, the use of virtual machines inevitably introduces performance degradation. In this section, we first evaluate the performance overhead of two currently supported VM platforms: VMware and UML. We then evaluate the end-to-end networking overhead caused by the Collapsar functional components for traffic redirection and dispatching.

To measure the virtualization-incurred overhead, we use two physical hosts (with aliases *seattle* and *tacoma*, respectively) with no background load, connected by a lightly loaded 100Mbps LAN. *Seattle* is a Dell PowerEdge server with a 2.6GHz Intel Xeon processor and 2GB RAM, while *tacoma* is a Dell desktop PC with a 1.8GHz Intel Pentium 4 processor and 768MB RAM. A VM runs on top of *seattle*, and measurement packets are sent from *tacoma* to the VM. The TCP throughput is measured by repeatedly transmitting a file of 100MB using different socket buffer sizes, while the latency is measured using standard ICMP packets with different payload sizes. Three sets of experiments are performed: (1) from *tacoma* to a VMware-based VM in *seattle*, (2) from *tacoma* to a UML-based VM in *seattle*, and (3) from *tacoma* directly to *seattle* with *no* VM running. The results in TCP throughput and ICMP latency are shown in Figures 3.2(a) and 3.2(b), respectively. The curves “VMware,” “UML,” and “Direct” correspond to experiments (1), (2), and (3), respectively.

Figure 3.2(a) indicates that UML performs worse in TCP throughput than VMware, because of UML’s user-level virtualization implementation. More specifically, UML uses a *ptrace*-based technique implemented at the user level and emulates an x86 machine by virtualizing system calls. VMware employs a *binary rewriting* technique implemented in the kernel, which inserts a breakpoint in place of sensitive instructions. However, both VMware and UML exhibit similar latency degradation because the (much lighter) ICMP

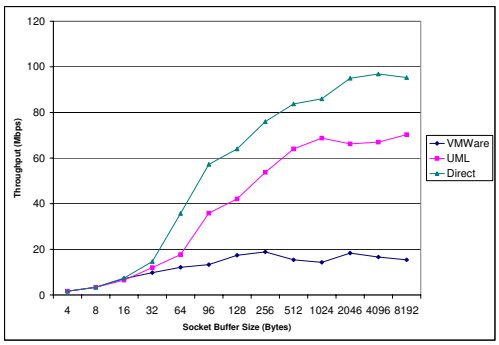


(a) TCP throughput degradation

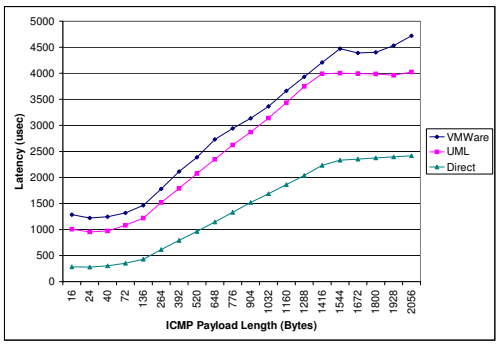


(b) ICMP latency degradation (increase)

Figure 3.2. Comparing virtualization-incurred overhead: VMware vs. UML



(a) TCP throughput degradation



(b) ICMP latency degradation (increase)

Figure 3.3. Comparing Collapsar-incurred overhead: VMware vs. UML

traffic does not incur high CPU load therefore hiding the difference between kernel and application level virtualization. A more thorough and rigorous comparison between VMware and UML is presented in [29].

We next measure the performance overhead incurred by the traffic redirection and dispatching mechanisms of Collapsar. We set up *tacoma* as the Collapsar front-end. In a

different LAN, we deploy a redirector running on a machine with the same configuration as *seattle*. The two LANs are connected by a high performance Cisco 3550 router. A machine M in the *same* LAN as the redirector serves as the "attacker" machine, connecting to the VM (honeypot) running in *seattle*. Again, three sets of experiments are performed for TCP throughput and ICMP latency measurement: (1) from M to a VMware-based honeypot in *seattle*, (2) from M to a UML-based honeypot in *seattle*, and (3) from M to the machine hosting the redirector (but without the redirector running). The results are shown in Figures 3.3(a) and 3.3(b). The curves "VMware," "UML," and "Direct" correspond to experiments (1), (2), and (3), respectively.

Contrary to the results in Figures 3.2(a) and 3.2(b), the UML-based VM achieves *better* TCP throughput and ICMP latency than the VMware-based VM. This is because of the optimized traffic dispatching mechanism implemented for UML (Section 3.4.2). Another important observation from Figures 3.3(a) and 3.3(b) is that traffic redirecting and dispatching in Collapsar incur a non-trivial network performance penalty (comparing with the curve "Direct"). For remote attackers (or those behind a weak link), such a penalty may be "hidden" by the already degraded end-to-end network performance. However, for "nearby" attackers, such a penalty may be observable by comparing performance to a real host in the same network. This is a limitation of the Collapsar design. Router-based traffic redirection (Section 3.4.1) as well as future hardware-based virtualization technology are expected to alleviate this problem.

3.6 Experiments with Collapsar

In this section, we present a number of real-world network attack incidents captured by Collapsar. We also present the recorded attacker activities to demonstrate the effectiveness and practicality of Collapsar. Finally, we demonstrate the potential of Collapsar in log mining and event correlation.

3.6.1 Environment Setup

In our Collapsar testbed, there are five production networks: three Ethernet LANs, one wireless LAN, and one DSL network. A Collapsar center is located in another Ethernet LAN. The virtual honeypots in the Collapsar center run a variety of operating systems, including RedHat Linux 7.2/8.0, Windows XP Home Edition, FreeBSD 4.2, and Solaris 8.0. Before the start of Collapsar operation, the checksum of every file (via *md5sum*) in a honeypot (virtual machine) has been calculated and stored for future reference. For each representative attack incident, we examine the specific vulnerability, describe how the system was compromised, and show the attacker's activities after the break-in. Our purpose is to demonstrate the effectiveness of Collapsar when facing real-world attacks.

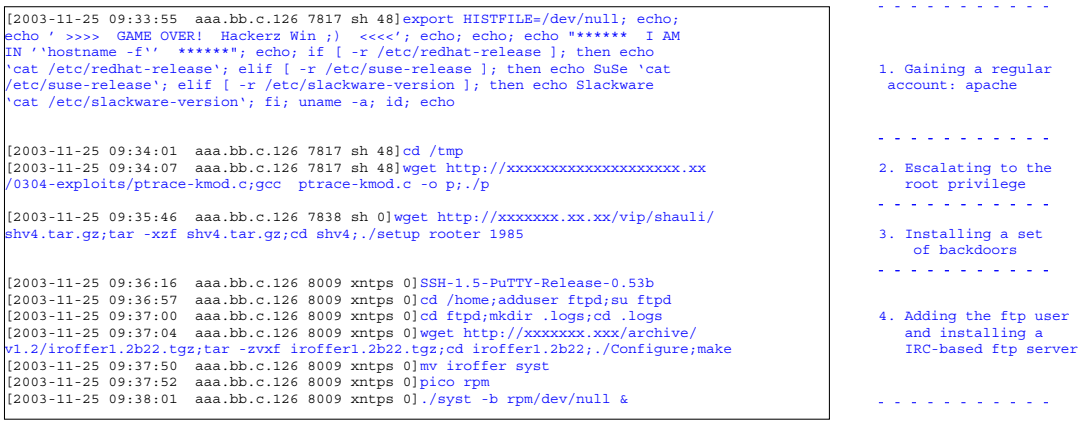


Figure 3.4. Collapsar log of attacker activities after break-in via Apache

3.6.2 Server-Side Honeypot Incidents

3.6.2.1 A Linux/VMware Server-Side Honeypot

The first recorded incident was an attack on an Apache server version 1.3.20-16 running on RedHat 7.2 using the Linux kernel 2.4.7-10. The honeypot compromised was a

VMware-based virtual machine in the Collapsar center, with logical presence in one of the LAN production networks.

Vulnerability description: Apache web server versions up to 1.3.24 contain a vulnerability [45] in the chunk-handling routines. A carefully crafted invalid request can cause an Apache child process to call the *memcpy()* function in a way that will write past the end of its buffer, corrupting the stack and thus resulting in a stack overflow. Remote attackers can exploit this vulnerability to access the system using the system's Apache account. Meanwhile, unpatched Linux kernels version 2.4.x contain a *ptrace* vulnerability [46], which can be exploited by malicious local users to escalate their privileges to root.

Incident: An Apache honeypot was deployed in the Collapsar center at 11:44:03PM on 11/24/2003 and was compromised at 09:33:55AM on 11/25/2003. Collapsar captured all information related to the vulnerability-exploiting process, including the attacker's keystrokes after the break-in as shown in Figure 3.4. The complete log of the break-in is available on the Collapsar website [47].

```
 **  0 packs  **   30 of 30 slots open, Min: 3.0KB/s
 **  Bandwidth Usage  **  Current: 0.0KB/s,
 **  To request a file type: "/msg xxxxxxxxxxx xxxx send #x"  **
 **  Brought To You By xxxxxx  **
Total Offered: 0.0 MB  Total Transferred: 0.00 MB
```

Figure 3.5. Attack via Apache leading to an *iroffer* backdoor (logged by Collapsar)

First, a TCP connection to port 443 on the honeypot was initiated, the attacker then sent one malicious packet (actually several TCP segments), triggering buffer overflow in the Apache web server. The malicious code contained in the packets spawned a shell with the privilege of the system's Apache account. With the shell, the attacker quickly downloaded, compiled, and executed a program exploiting the *ptrace* vulnerability [46]. Once executed, the *ptrace* exploitation code gave the attacker root privileges. After obtaining root privileges, the attacker downloaded a rootkit called *SHv4 Rootkit* [48] and installed a trojaned *ssh* backdoor with a password *rooter* on port 1985. Upon successfully installing the trojaned *ssh* server, a login session was initiated from PuTTY version 0.53b, a popular

Windows SSH client, to port 1985 accessing the trojaned *ssh* server, so that all communications between the honeypot and the attacker could be encrypted. Traditional techniques such as tcpdump and NIDS become less effective once traffic is encrypted. However, the Collapsar in-kernel logging module *sebek* [42] was able to hijack *SYS_read* system calls and recognize the attacker's keystrokes (Figure 3.4).

Backdoor in action: Based on the logged keystrokes, we were able to infer the attacker's tactics and goals. The attacker first added a new user account *ftp*, then installed *iroffer* [49]. *Iroffer* is a program that enables the hosting machine to act as a file server for an IRC channel similar to the *Napster* file sharing system [50]. Once started, *iroffer* connected to an IRC server and logged into a certain channel. The attacker was able to remotely re-configure *iroffer* which would periodically report its status in the channel, including available space, files, and transmission status. Figure 3.5 shows a status report generated by *iroffer* and logged by Collapsar logging module. It indicates that the attacker was able to request/offer files from/to others in the channel.

Forensic analysis: After detecting *iroffer* installation, no further keystrokes were captured. We took a snapshot of the honeypot image (available in [47]) and disconnected the honeypot from the Collapsar center. A quick verification using *md5sum* revealed several trojaned system routines, including *netstat, ls, ps, find,* and *top*; one *ssh* backdoor; and the *iroffer* program.

3.6.2.2 A Linux/UML Server-Side Honeypot

The second incident was an attack on the Samba server version 2.2.1a-4 running on RedHat 7.2. The honeypot was a UML-based virtual honeypot with enhanced network virtualization. The honeypot resided in the Collapsar center but had a logical presence in one of the LAN production networks.

Vulnerability description: The Samba server versions 2.0.x through 2.2.7a contains a buffer overflow vulnerability associated with the re-assembly of SMB/CIFS packet fragments [51]. This vulnerability allows a remote attacker to gain root privileges in a host running the Samba server.

Figure 3.6. Collapsar log of attacker activities after break-in via Samba

Incident: The Samba honeypot was activated in the Collapsar center at 12:01:03PM on 11/25/2003, and was compromised at 11:41:17AM on 11/26/2003. With the help of logging module *kernort*, Collapsar captured all information related to the attack, including scanning attempts and attacker keystrokes after the break-in (shown in Figure 3.6). The complete log can be found at [47]. First, a scanning NetBIOS name packet was sent to UDP port 137 and the honeypot running a vulnerable Samba server responded with MAC address 00-00-00-00-00-00, which indicated that a Samba server is running. After receiving the response, a TCP connection to port 139 was established and several malicious packets guessing different return addresses were sent in the hope of launching a buffer overflow attack. The malicious packets contained port-binding shell-code, which will listen on TCP port 45295 if correctly executed. Based on information in the Collapsar log information, we are able to identify six attempts to guess the return address, i.e., 0xbffffed4, 0xbffffda8, 0xbffffc7c, 0xbffffb50, 0xbffffa24, and 0xbffff8f8, in the malicious code.

After successfully exploiting the Samba server, the remote attacker gained root privileges and installed a rootkit wrapper *rkzz.tgz*, which contains a trojaned *sshd* backdoor and a sniffer program. Once the *sshd* backdoor was installed, the attacker quickly created an *ssh* connection using PuTTY-0.53b, encrypting all subsequent traffic. Using the *ssh* connection, the attacker downloaded a program package *skk.tgz*, which is the *SucKit* rootkit. It seemed that *SucKit* could not be installed successfully in the UML, so the attacker down-

loaded another attack package, *flood.tgz*, and immediately started a DoS attack. The attack package contained several DoS attack tools, including the infamous *smurf*, *overdrop*, and *synsend*.

Forensic analysis: Once the DoS attack was started, the tarpitting module in Collapsar detected a burst of out-going TCP-SYN packets, which indicated a successful compromise and an on-going DoS attack. The tarpitting module immediately raised an alarm and the Samba honeypot was disconnected from the Collapsar center. Forensic analysis revealed the installation of many flooding tools in /tmp/share/flood, which is consistent with the log information generated by the Collapsar logging module.

Another VMware-based virtual honeypot running the same *Samba* service was also compromised by the same IP, and an IRC bot, *psyBNC* [52], was installed enabling the attacker to remotely control the compromised honeypot via an IRC network. With VMware support, a snapshot of the honeypot was taken, demonstrating VMware's flexibility and convenience for forensic analysis over UML.

3.6.2.3 A Windows XP/VMware Server-Side Honeypot

The third incident was related to the RPC DCOM vulnerability in the Windows Platform. We deployed a VMware-based virtual honeypot running an unpatched Windows XP Home Edition operating system in the Collapsar center.

Vulnerability description: Windows DCOM contains a vulnerable Remote Procedure Call (RPC) interface [53], which can be exploited to run arbitrary code with local system privileges in the affected system. After a successful compromise, the attacker is free to take any action in the system including installing programs, modifying data, and creating new accounts with full privileges.

Incident: A honeypot running the unpatched Windows XP was deployed in the Collapsar center at 10:10:00PM on 11/26/2003, and was compromised several times on 11/27/2003: one at 00:36:47AM by the MSBlast.A worm [54], one at 01:48:57AM by the Enbiei worm (namely MSBlast.F worm), and another at 07:03:55AM by the Nachi worm [55]. Collap-

sar recorded all important log information covering the infection process of each worm. The complete log is available at [47].

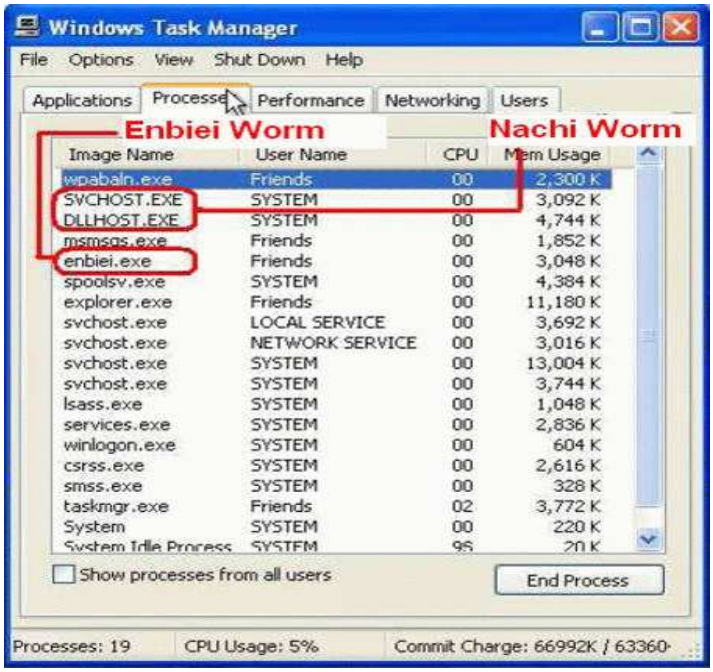


Figure 3.7. Screenshot re-constructed from a honeypot snapshot: successful break-ins by MSBlast, Enbiei, and Nachi worms

For each worm, an initial TCP connection was established with port 135 in the Windows XP honeypot (Nachi worm will use an ICMP echo request to test whether the target is alive before the TCP connection attempt). To the worm, a successful connection is an indication of possible existence of the RPC vulnerability. Once the connection had been established, malicious packets were sent, which caused a stack buffer overflow in the RPC interface implementing DCOM services. The malicious code contained port-binding shell-code, which would listen on TCP port 4444. After a shell was invoked, each worm downloaded and executed a copy of itself, completing one round of worm propagation.

The MSBlast and Enbiei worms mounted Denial of Service (DoS) attacks against two specific web sites. The Nachi worm tried to terminate and delete the MSBlast worm. In addition, after installing *tftpd.exe*, the TCP/IP trivial file transfer daemon, the Nachi worm tried to download and install an RPC DCOM vulnerability *patch* named *WindowsXP-*

KB823980-x86-ENU.exe, so that no other worms or attacks could break into the system by exploiting the same vulnerability.

Backdoor in action: Figure 3.7 shows a screenshot re-constructed from the honeypot's snapshot. It illustrates the running of *Enbiei* and *Nachi* worms. The original MSBlast worm has been terminated and deleted by the *Nachi* worm, which is the reason why no MSBlast process can be found in the screenshot. These worms also generated a large volume of scanning packets (ICMP echo request packets and TCP connection attempts to port 139 of other hosts), which were mitigated by the Collapsar tarpitting module.

Forensic analysis: After disconnecting the infected honeypot from the Collapsar center, a quick examination revealed the following files: *enbiei.exe* in directory *C:\WINDOWS \system32* and *SVCHOST.exe* and *DLLHOST.exe* in directory *C:\WINDOWS\system32 \wins*. File *enbiei.exe* corresponds to the Enbiei worm; while *SVCHOST.exe* and *DLL-HOST.exe* are for the Nachi worm. We also expected that file *msblast.exe* would exist in *C:\WINDOWS\system32*. However, it had been deleted by the Nachi worm.

3.6.3 Client-Side Honeypot Incidents

The previous three attack incidents were captured by the server-side honeypots in Collapsar. In the following, we present two malware attacks captured by the client-side honeypots in Collapsar.

3.6.3.1 A Windows XP/VMware Client-Side Honeypot

This incident is related to the Internet Explorer (IE) JView Profiler Vulnerability (MS05-037/CVE-2005-2087) on the Windows platform. We deployed a VMware-based client-side honeypot running an unpatched Windows XP system with the default IE web browser in the Collapsar center.

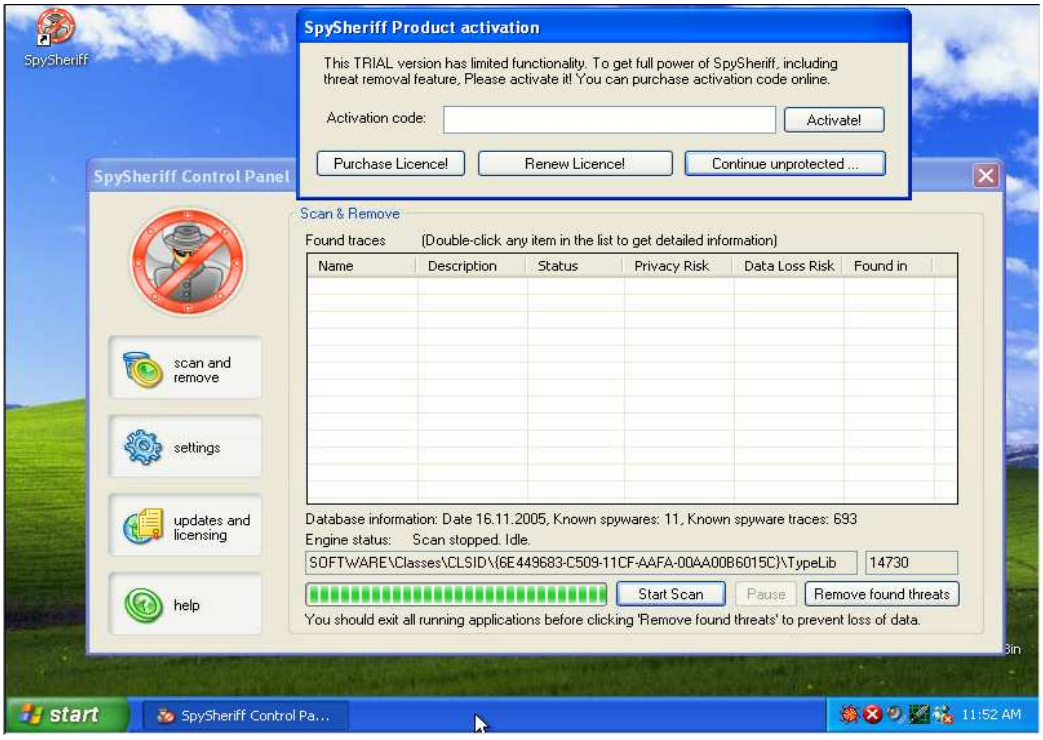
Vulnerability description: As described in CVE-2005-2087 [56], Internet Explorer 6.0.2900.2180 on Windows XP allows remote attackers to cause a denial of service (application crash) and execute arbitrary code via a web page with embedded CLSIDs that

reference certain COM objects that are not ActiveX controls. An example of non-ActiveX control is the JVIEW Profiler (Javaprxy.dll).

Incident: The client-side honeypot running the unpatched Windows IE was deployed in the Collapsar center at 08:10:00PM on 9/26/2005 and was driven to visit a URL, which we anonymized as *http://www.superxxxxx.com/xxxxxxx/yes.html*. After the visit, the wallpaper (shown in Figure 3.8(a)) of the honeypot's desktop displayed a warning that the system was infected with spyware. One minute later, "uninvited" software named *SpySheriff* was installed without user permission and ironically began to scan the local disk to remove possible spyware. The screenshot with its scanning activity is shown in Figure 3.8(b).

Forensic analysis: The honeypot was disconnected after the anomaly was observed. With the output of the Collapsar logging module, we were able to identify the cause and damages of this intrusion. More specifically, Figure 3.9 displays the malicious javascript snip from the exploit URL. The highlighted CLSID, which refers to the Javaprxy.dll COM object, was unsafely initiated to exploit the JView Profiler vulnerability. The successful exploitation resulted in the execution of embedded attack code (Figure 3.9), which replaced the desktop wallpaper (Figure 3.8(a)) and connected to another web server to download and install the *SpySheriff* program (Figure 3.8(b)).

(a) The first screenshot: the desktop wallpaper was replaced



(b) The second screenshot: "unwanted" software was installed and launched

Figure 3.8. Screenshots re-constructed from a honeypot snapshot after visiting a malicious URL

```
<SCRIPT language="javascript">
  shellcode = unescape("%u4343"+"%u4343"+"%u4040%u4040.....%u98ff");
  bigblock = unescape("%u0D0D%u0D0D");
  headersize = 20; slackspace = headersize+shellcode.length

  while (bigblock.length<slackspace)
       bigblock+=bigblock;

  fillblock = bigblock.substring(0, slackspace);
  block = bigblock.substring(0, bigblock.length-slackspace);

  while(block.length+slackspace<0x40000)
       block = block+block+fillblock;

  memory = new Array();
  for (i=0;i<750;i++)
       memory[i] = block + shellcode;
</SCRIPT>

 <object classid="CLSID:03D9F3F2-B0E3-11D2-B081-006008039BF0"></object>
```

Figure 3.9. Malicious javascript code from the exploiting URL

3.6.3.2 Another Windows XP/VMware Client-Side Honeypot

The fifth incident is also related to the Internet Explorer (IE) browser but involves another vulnerability: DHTML Method Heap Memory Corruption (MS05-014/CAN-2005-0055). A VMware-based client-side honeypot running an unpatched Windows XP system with the default IE web browser was deployed in the Collapsar center.

Vulnerability description: The unpatched IE browser contained a bug in its handling of certain DHTML methods. An attacker could exploit the vulnerability by constructing a malicious web page and luring a user to visit it. A successful exploitation of this vulnerability could allow attackers to take complete control of the compromised system [57].

Incident: The client-side honeypot running the unpatched Windows IE was deployed in the Collapsar center at 09:35:00PM on 10/06/2005, and was driven to visit a URL, anonymized as *http://xxx.9x.xx8.8x/users/xxxx/xxx/laxx/z.html*. After the visit, a total of 22 programs were installed in the honeypot without user permission.

Forensic analysis: Forensic analysis showed that this intrusion was obfuscated - the actual exploiting code taking advantage of the IE DHTML heap memory corruption vulnerability (MS05-014) did not unfold until after four stages of obfuscation: the first stage contained a customized javascript decode; the second stage exploited the IE support for dynamic

code generation with the *document.write()* primitive; the third stage contained another customized javascript decoder, which was revealed after the first two stages; finally, the fourth stage leveraged the Unicode character-set support in IE to further obfuscate attack code. The final attack code is shown in Figure 3.10: The DHTML method, i.e., *createControlRange()*, was exploited to cause heap memory corruption in the IE process and trigger the execution of an embedded well-crafted machine instruction sequence, which resulted in the installation of the 22 unwanted programs.

```
<script>
try{

  sc=unescape("%u4040%u4040 ... ... %ufbe6%u9e9e");
  bb=unescape("%u0d0d%u0d0d");
  while(bb.length<0x40000){ bb+=bb }
  bb=bb.substring(0,0x40000-sc.length-28);

  me=new Array();
  for(i=0;i<450;i++){
        me[i]=bb+sc
  }

  z=Math.ceil(0xd0d0d0d);
  z=document.scripts[0].createControlRange().length;

}catch(e){}
</script>
```

Figure 3.10. Malicious javascript code from the exploiting URL

3.6.4 Attack Correlation

The Collapsar center creates opportunities to perform correlation and mining-based attack analysis. In the following, we present two simple incidents based on the correlation of honeypot logs collected in Collapsar.

3.6.4.1 Stepping Stone Suspect

In the Collapsar log, a honeypot running a vulnerable version of the Apache web server was compromised by a remote machine with IP address (anonymized) *iii.jjj.kkk.11*. A rootkit and a trojaned *sshd* backdoor were then installed in the honeypot. The *sshd*

backdoor was configured with a password known to the attacker. One minute later, an *ssh* connection was initiated from a *different* remote IP address *xx.yyy.zzz.3* using the *same* password. There is a possibility that machine *iii.jjj.kkk.11* had itself been compromised before the attack on the honeypot running the Apache server was launched. This log information is shown in Figure 3.11. We note that such evidence is by no means sufficient to confirm a stepping stone [44] case. However, with a wider range of target networks and longer duration of log accumulation, a future Collapsar center will be more likely to detect stepping stones and trace back original attackers.

```
/* Exploit codes for Apache Chunk Handling Vulnerability */
... ...

17:45:43.014405 iii.jjj.kkk.11.4775 > aaa.bb.c.125.443: P 790:797(7) ack 5340
win 34880 <nop,nop,timestamp 22920631 5764072> (DF)
0x0000   4500 003b 71ef 4000 3306 fa74 cbc6 860b        E..;q.@.3..t....
0x0010   800a 097d 12a7 01bb 9b4c ee60 9b51 2c3e        ...}.....L.`.Q,>
0x0020   8018 8840 e50e 0000 0101 080a 015d bdb7        ...@.........]..
0x0030   0057 f3e8 2e2f 696e 7374 0a                    .W.../inst.

... ...
/* SSH connection against sshd backdoor from another different IP! */
17:46:46.104626 xx.yyy.zzz.3.1126 > aaa.bb.c.125.cfinger: S
389507617:389507617(0) win 8760 <mss 536,nop,nop,sackOK> (DF)
0x0000        4500 0030 1ac2 4000 6f06 30b7 51c4 e503  E..0..@.o.0.Q...
0x0010        800a 097d 0466 07d3 1737 6a21 0000 0000  ...}.f...7j!....
0x0020        7002 2238 16a3 0000 0204 0218 0101 0402  p."8............
17:46:46.105445 aaa.bb.c.125.cfinger > xx.yyy.zzz.3.1126: S
2758367448:2758367448(0) ack 389507618 win 5840 <mss 1460,nop,nop,sackOK> (DF)
0x0000        4500 0030 0000 4000 4006 7a79 800a 097d  E..0..@.@.zy...}
0x0010        51c4 e503 07d3 0466 a469 58d8 1737 6a22  Q......f.iX..7j"
0x0020        7012 16d0 211c 0000 0204 05b4 0101 0402  p...!...........
17:46:46.422319 xx.yyy.zzz.3.1126 > aaa.bb.c.125.cfinger: . ack 1 win 9112
(DF)
0x0000        4500 0028 1ac3 4000 6f06 30be 51c4 e503  E..(..@.o.0.Q...
0x0010        800a 097d 0466 07d3 1737 6a22 a469 58d9  ...}.f...7j".iX.
0x0020        5010 2398 4118 0000 4100 0000 0000       P.#.A...A.....
17:46:46.728800 aaa.bb.c.125.cfinger > xx.yyy.zzz.3.1126: P 1:16(15) ack 1 win
5840 (DF) [tos 0x10]
0x0000        4510 0037 55d5 4000 4006 248d 800a 097d  E..7U.@.@.$....}
0x0010        51c4 e503 07d3 0466 a469 58d9 1737 6a22  Q......f.iX..7j"
0x0020        5018 16d0 ac5b 0000 5353 482d 312e 352d  P....[..SSH-1.5-
0x0030        312e 322e 3235 0a                        1.2.25.
17:46:47.050246 xx.yyy.zzz.3.1126 > aaa.bb.c.125.cfinger: P 1:28(27) ack 16
win 9097 (DF)
0x0000        4500 0043 1ac5 4000 6f06 30a1 51c4 e503  E..C..@.o.0.Q...
0x0010        800a 097d 0466 07d3 1737 6a22 a469 58e8  ...}.f...7j".iX.
0x0020        5018 2389 4c55 0000 5353 482d 312e 352d  P.#.LU..SSH-1.5-
0x0030        5075 5454 592d 5265 6c65 6173 652d 302e  PuTTY-Release-0.
0x0040        3533 0a                                  53.
```

Figure 3.11. Collapsar log showing a possible stepping stone attack

3.6.4.2 Network Scanning

Network scanning has become common, with the existence of various scanning methods such as ping sweeping, port knocking, OS finger-printing, and firewalking. Figure

```
14:49:44.139231 xx.yy.zzz.125 > aaa.bb.9.126: icmp: echo request
0x0000   4500 005c 30de 0000 7301 0798 0c26 797d        E..\0...s....&y}
0x0010   800a 097e 0800 95dc 0200 0ace aaaa aaaa        ...~............
0x0020   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0030   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0040   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0050   aaaa                                        ..
14:50:21.853938 xx.yy.zzz.125 > ccc.dd.8.32: icmp: echo request
0x0000   4500 005c 2ece 0000 7301 0b06 0c26 797d        E..\....s....&y}
0x0010   800a 0820 0800 f2dd 0200 adcc aaaa aaaa        ................
0x0020   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0030   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0040   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0050   aaaa
14:50:50.970419 xx.yy.zzz.125 > eee.ff.21.9: icmp: echo request
0x0000   4500 005c 3e04 0000 7301 eee6 0c26 797d        E..\>...s....&y}
0x0010   800a 1509 0800 16d1 0200 89d9 aaaa aaaa        ................
0x0020   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0030   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0040   aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa        ................
0x0050   aaaa
```

Figure 3.12. Collapsar log showing an ICMP sweeping scan

3.12 shows the ICMP (ping) sweeping activity from the same source address (*xx.yy.zzz.125*) against three honeypots within a short period of time (1.0 second). The honeypots are virtually present in three different production networks. Based on the payload, it is likely that a Nachi worm [55] is performing the scan.

3.7 Related Work

Honeyd [26] is one of the most comparable projects with respect to support for multiple honeypots and traffic diversion. Simulating multiple virtual computer systems at the network level with different personality engines, honeyd is able to deceive network fingerprinting tools and provide arbitrary routing topologies and services for an arbitrary number of virtual systems. The main difference between honeyd and Collapsar is that honeyd is a low-interaction virtual honeypot framework, while all honeypots in Collapsar are high-interaction virtual machines. Honeyd is more scalable than Collapsar, because every computer system in honeyd is simulated. With high-interaction honeypots, Collapsar is able to provide a more authentic environment for attackers to interact with and has a potential for capturing attacks with zero-day exploits.

Potemkin [58] is another related project that creates a high-interaction honeyfarm with high fidelity as well as scalability. Although the original Collapsar (without reverse hon-

eyfarm support) [11] demonstrated the feasibility of the honeyfarm vision, Potemkin, via aggressive memory sharing and late binding of resources, significantly improves a honeyfarm's scalability by up to six orders of magnitude while still preserving the honeypots' fidelity in emulating the behavior of real-world hosts on the Internet. The current Potemkin prototype leverages the Xen [29] virtual machine platform, which requires open-source access and modification to the guest operating systems.

Network Telescope [34] is an architecture that provides distributed presence for the detection of global-scale security incidents. Using similar architectures, both iSink [59] and Netbait [60] run a set of simplified network services in participating machines. The services will log all incoming requests and aggregate log data in a centralized server, so that pattern matching techniques can be applied to identify well-known signatures of worms and viruses. None of Network Telescope, iSink, and Netbait involves real-time traffic diversion mechanisms. They either perform passive monitoring or provide limited interactivity with attackers. The Internet Storm Center [61] is an effort at the SANS institute to gather log data from participating intrusion detection sensors. The sensors are distributed around the world. However, this system does not present an interactive environment to attackers, nor is it capable of real-time attack traffic diversion.

Both the Strider HoneyMonkey exploit detection system [13] and the Honeyclient system [30] are among the first to propose and implement client-side honeypots. Different from server-side honeypots, client-side honeypots face the challenge of maintaining wide network coverage. To achieve wide coverage, client-side honeypots crawl the web to reveal possible exploits of client-side software. Unfortunately, the effectiveness of client-side honeypots may *decrease over time*, once the "launching" domain of the honeypots is *blacklisted* by malicious servers. Collapsar addresses this problem by realizing the reverse honeyfarm vision, so that client requests from the reverse honeyfarm will appear to the servers as coming from many different network domains.

3.8 Summary

We have presented the design, implementation, and evaluation of Collapsar, a virtualization based server-side and client-side honeyfarm architecture for Internet malware capture. Collapsar achieves the following salient properties: centralized honeypot management and decentralized honeypot presence. Centralized management ensures consistent, effective operations in deploying, administering, investigating, and correlating multiple honeypots, while distributed presence provides a diverse view of network attack activities. Real-world deployment and several attack incidents captured by Collapsar demonstrate its effectiveness and practicality in realizing both the server-side honeyfarm and client-side honeyfarm visions using one integrated architecture.

4 VIRTUAL PLAYGROUND FOR INTERNET WORMS AND MALWARE INVESTIGATION

4.1 Introduction

In the previous chapter, we present a virtualization-based honeyfarm architecture as the front-end to attract and capture Internet worms and malware. The next research problem is how to analyze a captured malware and understand its features and behavior. As mentioned in Chapter 1, we have witnessed increasingly novel features of emerging Internet worms and malware in their infection and propagation strategies [62], such as polymorphic appearance [63], multi-vector infection [64], self-destruction [65], and intelligent payloads with self-organized attack networks [10] or mass-mailing capability [66]. To understand key aspects of malware infection behavior such as probing, exploitation, propagation, and malicious payloads, researchers have hoped to have a safe and convenient environment to run and observe real-world malware. Such a "playground" environment is useful not only in assessing the impact of malware intrusion and propagation, but also in testing malware detection and defense mechanisms [32, 67–69]. For the rest of this chapter, we will focus on the design and implementation of playgrounds for self-propagating worms, though such a playground can readily be leveraged for general Internet malware investigation.

Despite its usefulness, there are research challenges 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. Unfortunately, this approach may not effectively address the above challenges, for the following reasons: (1) Because of the coarse granularity (the whole 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 not be shareable to remote researchers.

In this chapter, we present the design, implementation, and evaluation of a virtualization-based platform to create safe virtual worm playgrounds called *vGrounds*, on top of a general-purpose infrastructure. vGround has been used to analyze Linux worms, which represent a non-negligible source of Internet insecurity, with the increasing popularity of Linux in the server market. Though the current prototype does not support Windows-based worms, our design concepts and methodology can be applied to the development of Windows-based vGrounds.

The vGround platform conveniently turns a physical infrastructure into a base to host multiple vGrounds. An infrastructure can be a single physical machine, a local cluster, or a multi-domain overlay infrastructure such as PlanetLab [70]. 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, applications, and networking software, a vGround allows real worms to propagate as it does in the real Internet. Our full-system virtualization approach achieves the fidelity that leads to more opportunities to capture the nuances, tricks, and variations of worms, compared with simulation-based approaches [71].

- *Strict confinement* Using our VM and VN (virtual network) technologies, the real Internet is invisible (unaddressable) from inside a vGround, preventing the leakage

of negative impact caused by worm infection, propagation, and malicious payloads [2, 65] into the underlying infrastructure and cascadingly, the rest of the Internet. Furthermore, the damages caused by a worm affect only the virtual entities and components in one vGround and do *not* affect other vGrounds running on the same infrastructure.

- *Flexible and efficient worm experiment control* Because of the all-software nature of vGround, the instantiation, re-installation, and tear-down of a vGround are fast and automatic, saving worm researchers time and labor. For example, in our Lion worm experiment, it takes only 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*, as these operations could take hours or even days if the same experiment is performed in a physical playground. More importantly, vGround can be used by researchers *without* the need for administrator privileges 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 experimented with the Lion worm [2] in a vGround with 2000 virtual end hosts, based on 10 physical machines in a Linux cluster.

  Although such scalability is effective in exposing worm propagation strategies based on our limited physical resources (Section 4.4), it is *not* comparable to the scale achieved by worm simulations. With different goals and focuses, vGround is more suitable for analyzing system-level worm actions and damages, while the simulation-based approach is better for network-level modeling of worm propagation under Internet scale and topology. Moreover, lacking realistic background computation and traffic load, current vGrounds are not appropriate for quantitative modeling of worm propagation.

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 vGround in worm investigation and defense research.

The rest of this chapter is organized as follows: Section 4.2 provides an overview of vGround . The detailed design is presented in Section 4.3. Section 4.4 demonstrates the effectiveness of vGround using our experiments with several real-world worms. A discussion on its limitations and extensions is presented in Section 4.5. Related works are discussed in Section 4.6. Finally, Section 4.7 summarize this chapter.

4.2 Overview of vGround

A vGround is a virtualization-based self-confined worm playground where every entity, including the end host, firewall, router, and network cable, is virtualized. Moreover, all network traffic is strictly confined within the vGround. vGrounds can be safely created on a wide range of general-purpose infrastructures, including desktops, clusters, and wide-area shared infrastructures such as PlanetLab. Figure 4.1 shows a simple vGround [1] created on top of three PlanetLab hosts A, B, and C. The vGround emulates three enterprise networks connected by three routers ($R1$, $R2$, and $R3$). Within the vGround, the "seed" worm node ($AS1\_H1$ in network A 128.10.0.0/16) is set to infect other nodes running vulnerable services. The network address space of a vGround is orthogonal to that of the real Internet. Furthermore, multiple simultaneously running vGrounds can safely overlap their address spaces without affecting each other because of their mutual isolation.

Using a vGround specification language, a worm researcher is able to specify the worm experiment setup in a vGround, including software systems and services, IP addresses, and routing information in virtual nodes (i.e. virtual end hosts and routers). Based on the specification, the vGround platform will perform *automatic* vGround instantiation, bootstrapping, and clean-up. In a typical worm experiment, multiple runs are often needed,

[1] The vGrounds for our worm experiments are much larger in scale.

as each run may be configured using a different parameter setting (e.g., different worm signatures [41, 72] and different traffic throttling thresholds [32]). Because of the worm's destructive behavior, the vGround will become unusable after each run and need to be re-installed. The vGround platform is especially efficient in supporting such an *iterative worm experiment workflow*.

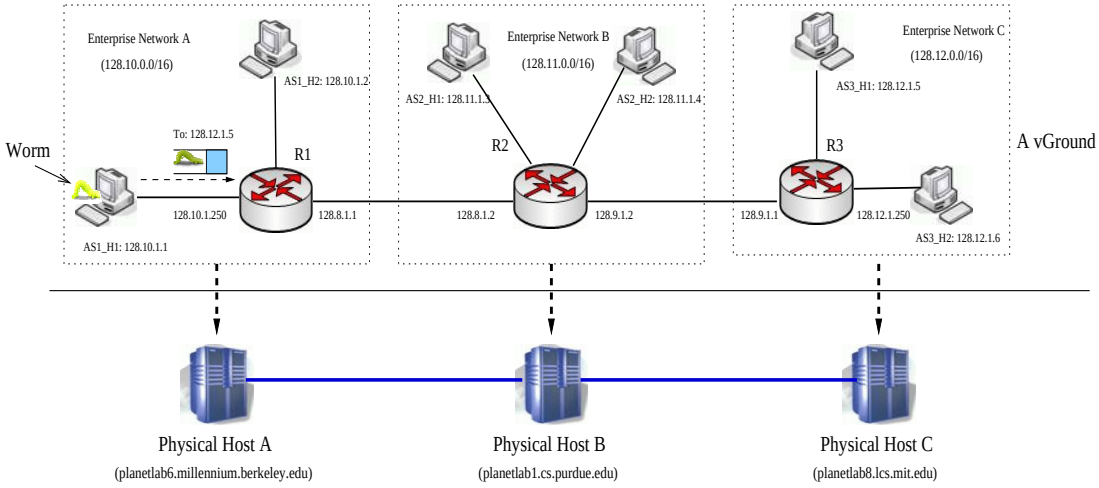


Figure 4.1. A PlanetLab-based vGround for worm experiments

4.2.1 Key vGround Enabling Techniques

Full-system virtualization is adopted to achieve high *fidelity* of vGrounds. Worms infect machines by remotely exploiting certain vulnerabilities in the OS or application services (e.g., BIND, Sendmail, DNS). Therefore, vGrounds should provide the same vulnerabilities as those in real software systems. Furthermore, vGround should help uncovering *unknown vulnerabilities* exploited by worms, which cannot be discovered by worm simulations.

There exist various VM technologies that enable full-system virtualization. The differences in their implementations lead to different levels of cost, deployability and configurability: VMware [21] and similarly Virtual PC [38] require several loadable kernel modules for virtualizing underlying physical resources; Xen [29] and Denali [73] "paravirtualize" physical resources by running in place of the host OS; and UML [22] 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 root privileges in the shared infrastructure. As a result, the current vGround prototype can be widely deployed in most Linux-based systems (including PlanetLab). We note that the original UML is *not* able to satisfy vGround needs and we have developed new extensions to UML as described next.

New network virtualization techniques are developed to achieve vGround *confinement*. Running a worm experiment simply in a number of VMs will not confine the worm traffic within these VMs and thus prevent potential worm "leakage." Although existing UML implementation has some support for virtual networking, it is not capable of organizing different VMs into an *isolated* virtual topology. In particular, when the underlying shared infrastructure spans multiple physical domains, additional VPN software is needed to create the illusion of a 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 isolation requirement; (2) A VPN usually needs to be statically and manually configured as it requires root privileges 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 intercepts the traffic at the link-layer and is able to constrain both the topology and volume of VM traffic generated in the vGround. Such a VN creates the illusion of 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 network virtualization technique, 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 can therefore be safely removed. We also develop a new method to safely and efficiently generate VM images in each physical host (Section 4.3.4). 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.

```
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
    }
}
```

Figure 4.2. An example vGround specification

4.2.2 Interface for vGround Configuration

The vGround platform provides a specification language interface 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 address. In addition, the virtual nodes are properly connected using routing mechanisms. Currently, the vGround platform supports RIP, OSPF, and BGP protocols.

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 notion of a system template is motivated by the observation that most end-hosts to be victimized by a certain worm look 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.

As an example, Figure 4.2 shows the specification for the vGround in Figure 4.1. 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. The keyword *cluster* is not used in this example. However, for a large-scale vGround, it is convenient to use *cluster* to specify a subnet, which has a large number of end-hosts of similar configuration.

Once 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, to be described in Sections 4.3 and 4.4.

4.3 Design Details

4.3.1 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 dissertation and interested readers are referred to [22].

It is interesting to note that in an earlier implementation of UML, i.e., the "tt" mode [22], the UML guest-OS kernel needs to reside in 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 [22], where the tracing process acts as a kernel-level thread and does not impose any restriction or risk. This explains why certain worms such as *Lion* cannot successfully propagate in a vGround on top of PlanetLab, as the OS kernel of PlanetLab hosts does not usually support the "skas" mode.

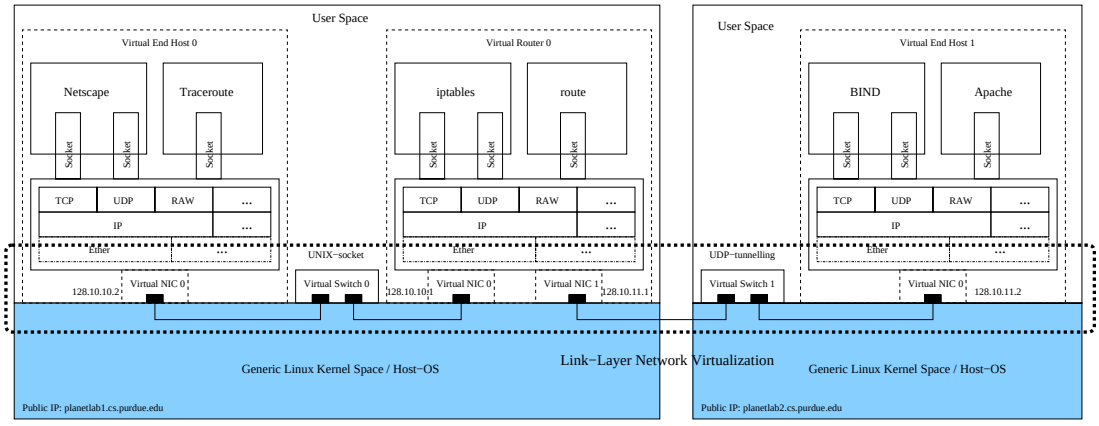


Figure 4.3. Illustration of link-layer network virtualization in vGround

4.3.2 Link-Layer Network Virtualization

Figure 4.3 illustrates the link-layer network virtualization technique (highlighted by the dotted rectangle) developed for vGround. 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 connection requests from other virtual NICs. Each successful connection emulates a virtual cable. The

virtual NIC is largely based on the original UML implementation with certain extensions to communicate with remote virtual switch daemons. These entities are link-layer "devices," which are not tamperable from inside a VM. This new design differentiates our technique from other virtual networking techniques [74, 75] and is critical to achieving strict confinement for a vGround. Also, the user-level implementation of our network virtualization methods brings significant deployability and topology flexibility to vGrounds.

To demonstrate vGround's address space isolation, we again use the PlanetLab example in Figure 4.1. We execute the command *traceroute* in the VM $AS1\_H1$ to find the route to $AS3\_H2$. The result is shown in Figure 4.4, indicating that the route is orthogonal to the real Internet. More details of our network virtualization technique can be found in [76].

```
[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]#
```

Figure 4.4. Running *traceroute* inside a vGround on PlanetLab

4.3.3 Virtual Node Optimization and Customization

A virtual node in a vGround can be one of the following: (1) an end-host exposing certain software vulnerabilities to 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 disk 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 4.2.1, images of the same type of virtual nodes have a lot in common though they may have different network configurations. Every image file in vGround is composed of two parts: a shared system template and a node-specific part. In the example in Figure 4.2, 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. Every virtual node has its unique networking configuration (e.g., IP address and routing table), which is specified in the node-specific part. To execute a vGround specification, we leverage the Copy-On-Write (COW) support in UML, which also helps achieve high image generation efficiency.

Another optimization is the stripping down of system templates. When a vGround contains hundreds or thousands of virtual nodes, the templates will have to be tailored by removing unneeded services. For worm experiments, this proves feasible because most worms infect and propagate via only one or a small number of vulnerabilities. For example, for the Lion worm experiment, a tailored system image with BIND-$8.2.1$ service only requires $7MB$. Since the system templates are regular *ext2/ext3* file systems, it is possible to build customized system templates from scratch. Packaging tools such as *rpm* can also be leveraged to simplify this process.

4.3.4 Worm Experiment Services

To provide users with worm experiment convenience, vGround provides a number of 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 root privileges of the underlying physical host. To efficiently generate image files without root privileges, a *"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* [22], 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 be performed using the VM's own root privileges. 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 for worms to access 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.

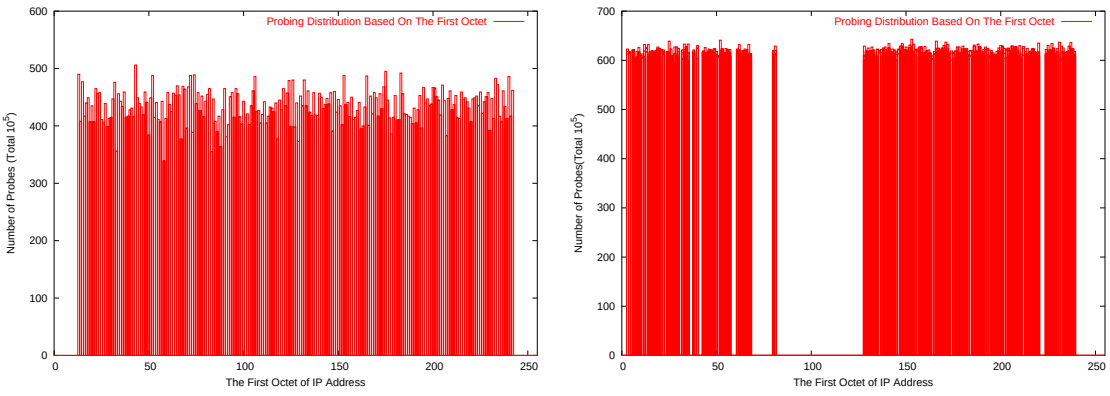
Generation and collection of worm traces Each virtual node in vGround has an embedded logging module as part of its VM image. The logger generates worm traces, which will be collected for analyzing different aspects of worms. The vGround platform supports different types of logging modules. Linux-based monitoring or intrusion detection systems, such as *tcpdump* [40], *snort* [41], and *bro* [72], can be readily packaged and installed in vGround. In addition, we have designed and implemented a *kernelized* version of *snort*, named *kernort* [43], 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 obviously 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 instantiated to mount the image file to be inspected (an *ext2/ext3* file), and "evidence" collection will be performed via the special VM.

4.4 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 [2], Slapper worm [10], and Ramen worm [77]. The experiments span from individual stages for worm infections (e.g., target network space selection (Section 4.4.1), propagation pattern and strategy (Section 4.4.2), exploitation steps (Section 4.4.3), and malicious payloads (Section 4.4.4)) to more advanced schemes such as intelligent payloads (Section 4.4.4), multi-vector infections (Section 4.4.5), and polymorphic appearances (Section 4.4.5). Throughout this section, we will highlight the new benefits vGrounds bring to a worm researcher, as well as some worm analysis results obtained during our experiments. We discuss the limitations and extensions in Section 4.5.

The infrastructure in our experiments is a Linux cluster, which belongs to the Computing Center of Purdue University (ITaP). Neither do we have root privileges nor do we obtain special assistance from the cluster administrator, indicating vGround's 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.



(a) Target network space of Lion worm

(b) Target network space of Slapper worm

Figure 4.5. Target network space of the Lion worm and the Slapper worm

4.4.1 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 [78], but also reveals the address blocks that have been "black-listed" by the black-hat community (for example, the address blocks used for sinkhole networking [59]).

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" [2]. 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 about 150 virtual nodes. The image files are generated within 60 seconds and the vGround is booted-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 4.5(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 13.0.0.0/8 - 243.0.0.0/8. It seems that the Lion worm does not skip private or reserved address blocks [78]. To verify this observation, we also perform reverse engineering using *objdump* [79] 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 [10]. The Slapper worm was captured and thoroughly analyzed by researchers at Symantec [10].

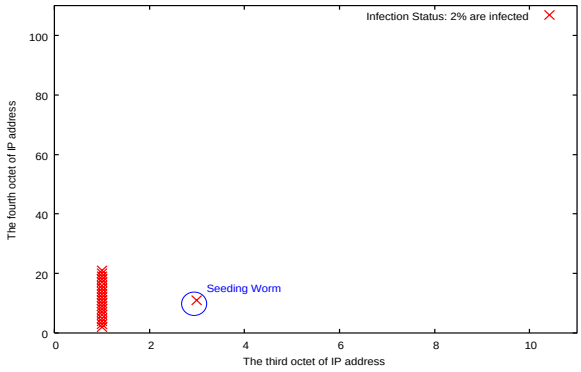
A system template *slapper.ext2* contains the vulnerable version of *Apache* server. The size of the image is approximately $32M$. A vGround with 2000 virtual nodes is deployed on 20 physical hosts of the Linux cluster, with each hosting 100 virtual nodes. Similar to the Lion worm experiment, we extract the probing traffic from the Slapper-infected nodes and then plot the target address distribution in Figure 4.5(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, the address blocks of 82.0.0.0/8 - 88.0.0.0/8 were reserved by IANA (Internet Assigned Numbers Authority) and therefore skipped by the Slapper worm, as shown in Figure 4.5(b).

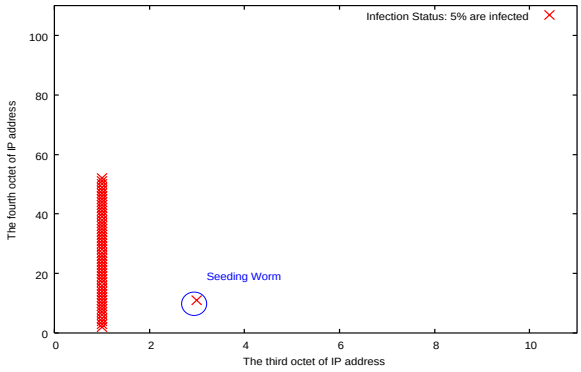
4.4.2 Propagation Patterns

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.

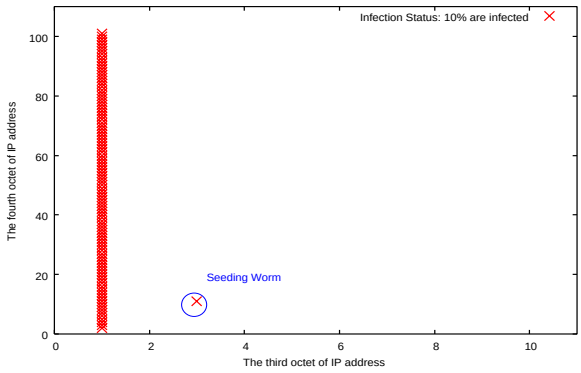
We create a vGround with 1000 vulnerable end-hosts running in 10 networks each with 100 end hosts (192.168.x.y, $x = 1 \cdots 10$, $y = 1 \cdots 100$). At the beginning, there is *one* Slapper-infected "seeding" node (192.168.3.11) in the vGround. We allow the Slapper worm to propagate in the vGround and the propagation progress is recorded. Based on the vGround traces, the propagation pattern of Slapper worm can be visualized in Figure 4.6. 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.



(a) When 2% hosts infected

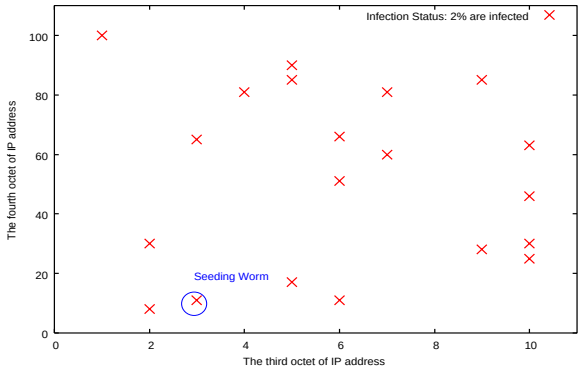


(b) When 5% hosts infected

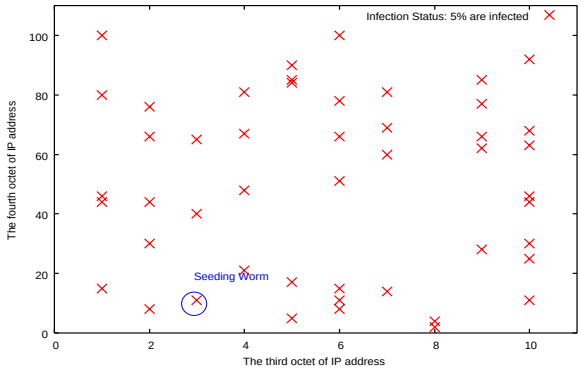


(c) When 10% hosts infected

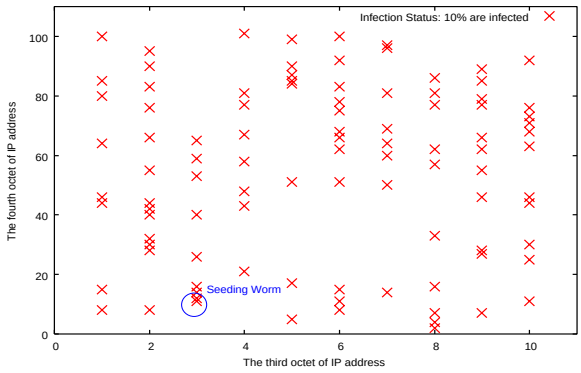
Figure 4.6. Propagation of Slapper worm w/ *address-sweeping* (total: 1000 hosts)



(a) When 2% hosts infected



(b) When 5% hosts infected



(c) When 10% hosts infected

Figure 4.7. Propagation of a Slapper worm variant w/ *island-hopping* (Total: 1000 hosts)

From Figure 4.6, it can be conjectured that the Slapper worm is using an *address-sweeping* strategy when selecting victims: once an address range such as 192.168.0.0/16 is chosen, hosts within the address range will then be *sequentially scanned*. Figure 4.6 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.

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 [80]. 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 4.7. 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, because of 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, 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.

4.4.3 Detailed Exploitation Steps

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

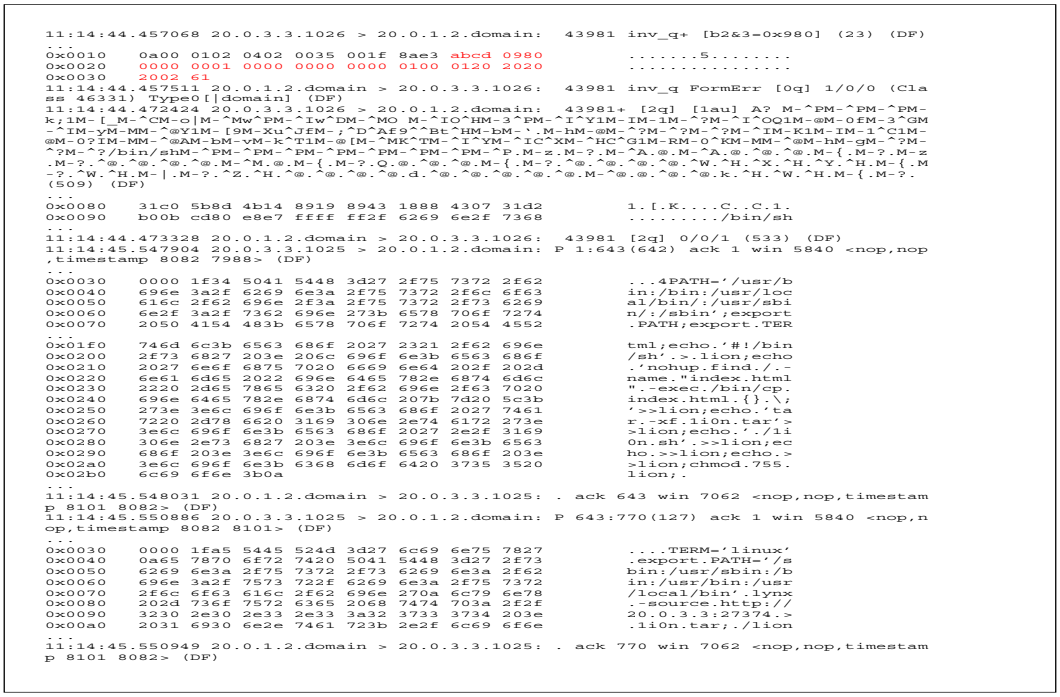


Figure 4.8. Exploitation details of the Lion worm

Lion worm Figure 4.8 shows a *tcpdump* trace generated in the vGround for the Lion worm experiment in Section 4.4.1. 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 [81] 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 [41] for Lion worm detection. The trace also shows the sequence of specially-crafted commands executed next, which result in the transfer and activation of a worm copy.

Slapper worm The Slapper worm is unique in its heap-based exploitation [82]. vGround successfully reproduces the detailed exploits: Initially, a TCP connection is initiated to

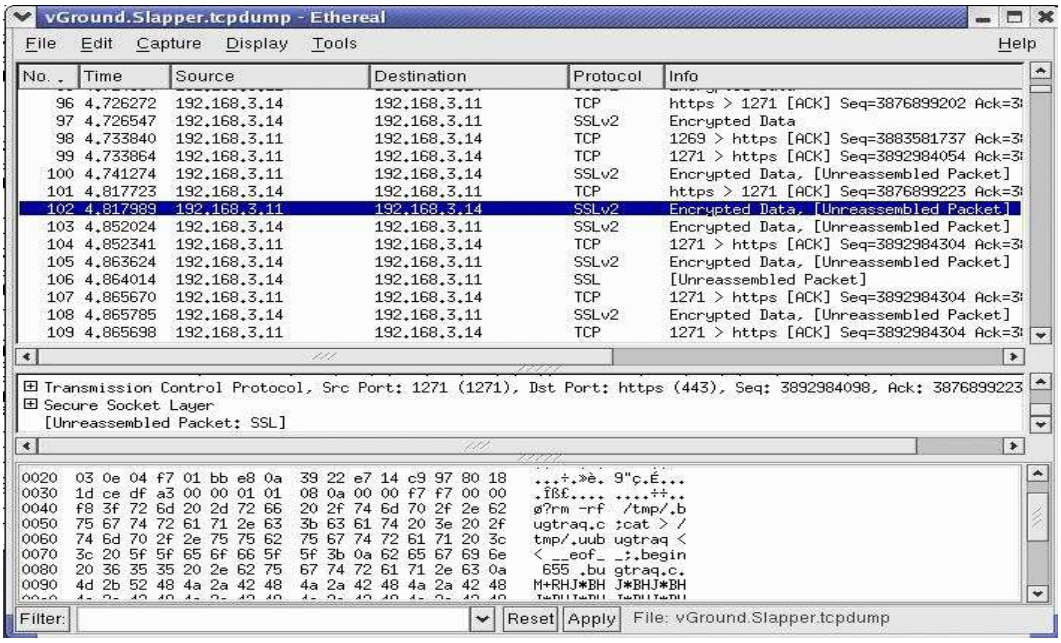


Figure 4.9. Exploitation details of the Slapper worm

verify the reachability of a victim, which, if reachable, is followed 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 occupies all existing Apache processes 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, which is used in the second SSL connection to correctly patch the attack buffer. The second SSL connection re-triggers the heap-based buffer overflow that transfers to the control of the just-patched attack buffer.

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 4.9. From the decoded area of Figure 4.9, it can be seen that the worm source is transferred in the *uuencoded*[2] format.

[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.

4.4.4 Malicious Payloads

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 because of the all-software user-level implementation.

The following string is found in the Lion worm trace in Figure 4.8: *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 4.3.4). 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* [83], which will destroy the infected 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.

```
[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
```

Figure 4.10. Payloads of the Slapper worm

The Slapper worm 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 [84] in one of the end hosts. The special client will issue commands (listed in Figure 4.10) to the infected hosts. Meanwhile, each Slapper worm carries a DDoS payload component [84]. 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.

4.4.5 Advanced Worm Experiments

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

Multi-vector worms Multi-vector worms are able to infect hosts via *multiple infection vectors (IVs)*. In this experiment, we run the Ramen worm [77], which carries three different IVs in three 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 time, we notice *different* infection attempts based on all three IVs.

Our vGround experiments reveal that the Ramen exploitation code for the vulnerable wu-ftpd server is flawed - a result not mentioned in popular bulletins such as [77]. 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 confirms the vGround observation.

Stealthy/polymorphic worms Using various polymorphic engines [63], 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 difficult, if not impossible, for worm simulators [71] to experiment with 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 4.4.3, the Slapper worm will transfer a *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 [41] 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 [67–69].

Routing worms vGround can also be used to study the relation between worm propagation and the underlying routing infrastructure. We have synthesized the routing worm introduced in [85]. The routing worm takes advantage of the information in BGP routing tables to reduce its scanning space, without missing any potential target. With network virtualization and real-world routing protocol support, vGround provides a new venue to study such an infrastructure-aware worm and the corresponding defense mechanisms.

4.5 Limitations and Extensions

It has been noted that a UML-based VM exposes certain system-wide footprints [24]. 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 a deficiency may undesirably disclose the existence of vGround. As a counter-measure, methods have been proposed [86] 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." Meanwhile, the trend is that VMs are increasingly used for general-purpose computing such as web hosting [17], education [18], and Grid computing [22, 75]. If such a trend continues, the arms race tension may be *mitigated* because a worm might as well infect a VM in such a "mixed-reality" cyberspace.

A confined vGround may *disable* some worm experiments where the worm has to communicate with hosts outside the vGround to succeed. For example, the Santy worm [87]

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

relies on the *Google* search engine to locate targets for infection and it can be effectively mitigated by filtering the worm-related queries [88]. However, vGround is not yet effective in observing the dynamics of such worms. Although the vGround platform is capable of intercepting an external connection attempt and forging a corresponding response, it remains an open problem whether such technique can survive the subsequent countermeasures taken by the worms.

Another limitation of the current vGround prototype is that it is only applicable to Linux worms, though the design principles and concepts can be generally applied to future vGrounds for Windows worms. One challenge in extending the current vGround implementation for Windows worms is to develop highly scalable system virtualization and customization techniques for Windows systems. It is encouraging to note that recent advances in system virtualization technologies such as the VMware ESX server [21] and hardware-based virtualization support such as Intel's Vanderpool technology [89] have shown promise in addressing this challenge.

4.6 Related Work

Testbeds for destructive experiments The DETER project [90] provides a shared testbed for researchers to conduct a wide variety of security experiments. With a pool of physical machines at a number of sites, the DETER testbed is able to provide each researcher with a virtually dedicated experiment environment. In 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. 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 [91], Modelnet [92], and PlanetLab [70] are highly valuable and accessible testbeds or environments for general networking and distributed system experiments. The vGround platform is an enabling software system that can potentially 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 involved.

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 skillful and trained experts. As a result, wide deployability, infrastructure sharing, and user convenience are *not* their primary design goals. One of the pioneering industry testbeds is Internet-inna-Box [93] 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. Furthermore, vGrounds support virtual routers and user-specified network topologies.

VM-based worm investigation Virtual machines provide an additional layer of indirection for running and observing untrusted services and applications. Especially, VM technologies have been heavily leveraged to study worms: VMs can be used as honeypots to capture worms [11, 58]; VMs can also be used to analyze worms. An advanced VM-based forensic platform is ReVirt [94]. ReVirt enhances individual VMs with efficient logging and replay capabilities for intrusion analysis, enabling a worm researcher to replay the worm exploitation process in an instruction-by-instruction fashion. Finally, to study *how worms propagate*, we argue that only VMs are not enough, which leads to our development of new network virtualization techniques.

Virtual networks In [95], 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 [74], VNET [75], and VIOLIN [76]. Both X-bone and VNET create a "virtual Internet" without hiding

the presence of the underlying physical hosts and their network connections. If used in vGround, they would not be able to confine worm traffic. VIOLIN is our earlier effort in network virtualization and it does not provide automatic virtual network generation and bootstrapping capabilities.

4.7 Summary

The vGround back-end enables impact-confined, resource-efficient experiments with Internet worms and malware. The key features of vGround are supported by a suite of virtualization techniques. Using real-world worms, we demonstrate that vGrounds are high-fidelity, mutually-isolated environments to unleash worms and to observe multiple aspects of their behavior, including network space targeting, propagation pattern, exploitation steps, and malicious payload. These results are critical to the development of worm defense mechanisms, which will in turn be tested in vGrounds.

5 CHARACTERIZING SELF-PROPAGATING WORMS WITH BEHAVIORAL FOOTPRINTING

The previous two chapters describe the front-end and back-end of our integrated framework for Internet malware capture and investigation. These two components combined create a unique experiment platform for the design and evaluation of advanced malware defense techniques. In this chapter, we present a malware defense mechanism that we have developed on top of the integrated platform for worm characterization.

5.1 Introduction

To effectively identify and defend against Internet worms, it is important to characterize a worm along *multiple* dimensions and derive its profile. Content-based signature [67–69, 96] is a well-established dimension to capture a worm's characteristics by deriving the most representative content sequence as the worm's signature. In practice, various intrusion detection systems (IDSes) [41, 72], together with recent honeypot systems [24, 26, 47, 58], are deployed to collect live worms. Once a worm specimen[1] is collected, anti-worm experts will manually examine the specimen and extract the worm-identifying content sequence as the worm's signature or "fingerprint." Recent works [67–69, 96] take one step further by automatically generating worms' content-based signatures. These systems have demonstrated a degree of success. However, they all focus on the same dimension of worm characterization, namely content, while missing other aspects of a worm's profile. This single-dimension characterization may limit the capability of worm identification and recognition. For example, it has been demonstrated that advanced worms are now capable of exploiting the weakness of content-based signatures

[1]The worm specimen might not only contain the worm binary itself, but also include other corresponding traffic associated with a worm infection (e.g., exploitation).

by mutating [97] or encrypting [98] their content or payload, hence escaping the detection and identification by content-based fingerprints.

We are therefore motivated to explore other dimensions to enrich a worm's profile and thus to enhance worm identification capabilities. Especially, we realize that content-based fingerprinting does not capture a worm's *temporal* infection behavior, which contains valuable self-identifying information that leads to the worm's recognition. In the following, we propose and justify a new dimension, *behavioral footprinting*, for worm characterization. We note that behavioral footprinting is orthogonal and complementary to content fingerprinting. This new dimension alone also suffers from ineffectiveness towards certain worms. In this dissertation, we target the type of worms that exploit traditional vulnerable servers (e.g., Apache/IIS, DNS, and Sendmail) to propagate themselves without any human intervention [9, 10, 54, 77, 99–103]. Other types of worms (e.g., mass-mailing or IM worms [4] requiring user actions) are subjects of future work.

The rest of this chapter is organized as follows: In Section 5.2, we demonstrate the existence of behavioral footprints in self-propagating worms and make a case for the new dimension. We then describe in Section 5.3 our algorithms to extract a worm's behavioral footprint, which are followed by experimental results with a number of real worms using Collapsar and vGround in Section 5.4. Limitations and possible improvements are described in Section 5.5. We present related work in Section 5.6 and finally summarize this chapter in Section 5.7.

5.2 A Case for Behavioral Footprinting

In this section, we first present a staged view of a worm infection session to motivate the characterization of worm behavior. As two representative examples, we illustrate the existence of behavioral footprints in two well-known worms: the MSBlast worm (propagating on Windows platform) and the Lion worm (propagating on Linux platform). Finally, we make a case for behavioral footprinting.

5.2.1 A Staged View of Worm Infections

In general, the infection session of a self-propagating worm from an infected host to a victim host can be broken into three phases:

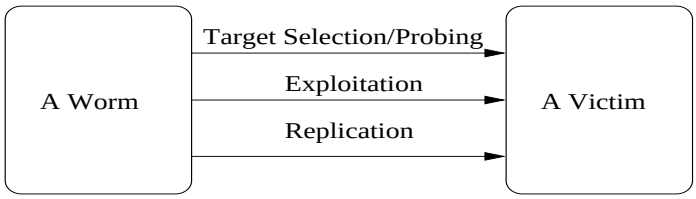


Figure 5.1. A staged view of a worm infection session

Phase 1: Target selection and probing Using a strategy such as random or biased address scanning, a scanning worm during this stage attempts to select a victim for infection. For example, an ICMP echo request packet or a TCP *SYN* probe is used to infer the reachability of a chosen target. Additional packets may also be used to obtain the version of a possibly vulnerable service. We note that this phase may *not* exist for non-scanning worms because they may carry a pre-computed target list.

Phase 2: Exploitation Once the worm receives a positive response from the victim host, a number of malicious packets[2] may be sent in an attempt to exploit the targeted vulnerability. Successful exploitation will result in the execution of specifically crafted code by the victim host. Different worms usually implement different functionalities in the crafted code.

Phase 3: Replication If the exploitation is successful, the replication phase will follow to transmit a worm replica to the victim host. The replica will be installed in the victim host, completing this infection session.

We will show through examples that the behavior exhibited by a worm during its infection session contains valuable self-identifying information that can be used to characterize the worm. The temporal order of infection steps taken by the worm reflects their intrinsic dependencies that must be followed to ensure a successful infection.

[2]There are certain worms (e.g., Slammer [9]) that blindly send exploitation packets to any hosts probed.

5.2.2 Example I: the MSBlast Worm

We consider the infamous MSBlast worm [54] as the first motivating example. The MSBlast worm exploits an RPC-DCOM vulnerability (MS03-026) for its infection. An MSBlast infection session is illustrated in Figure 5.2. The infection session consists of the following steps:

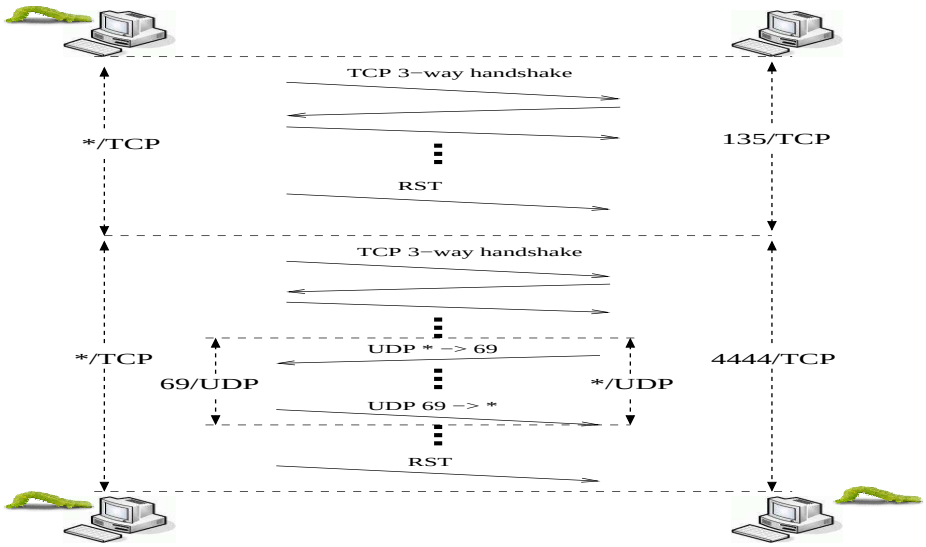


Figure 5.2. An infection session of the MSBlast/Windows worm

- A three-way TCP handshake on port 135 [3] is used by the worm to check the reachability of the selected target (Phase 1).
- Upon the establishment of the TCP connection, the worm sends a number of malicious packets (Phase 2), which exploit the RPC-DCOM vulnerability [54] and contain specially crafted shell-code. A successful exploitation will lead to the execution of the shell-code in the victim node. In the case of the MSBlast worm, a new shell service will be started on TCP port 4444 by the shell-code.

[3] Microsoft's DCOM Service Control Manager (also known as the RPC Endpoint Mapper) uses this port as a well-defined means to provide port-mapping services associating available services with their ports.

- The new shell service on $4444/TCP$ is immediately contacted by the worm to send instructions on how to download the worm replica, i.e., *msblast.exe* (Phase 3). It can be seen from Figure 5.2 that the *TFTP* protocol is used for the downloading.

The above sequence of actions significantly deviates from a normal access to the RPC-DCOM service: First, after the service request, a new shell service would not suddenly appear and listen on $4444/TCP$ in the victim host. Second, a new TCP connection to *this* port would not follow the service request. Third and most importantly, the victim should not have taken the initiative in using the *TFTP* protocol to download a file (with the name *msblast.exe* and a size of $6,372$ bytes) from the client.

5.2.3 Example II: the Lion Worm

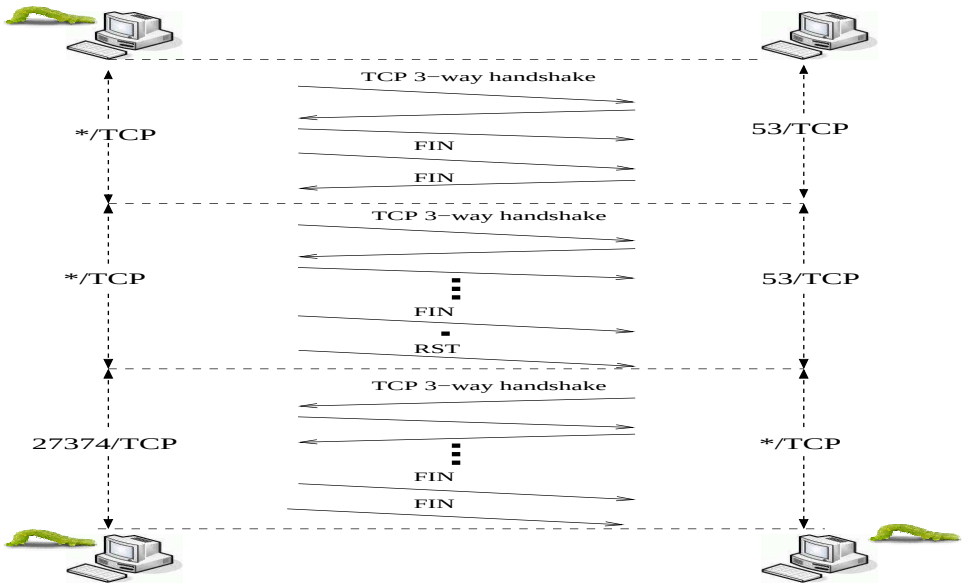


Figure 5.3. An infection session of the Lion/Linux worm

The second illustrative example is the historical Linux-based Lion worm [2]. The Lion worm exploits a BIND vulnerability (CA-2001-02) for its infection. A Lion worm infection session is shown in Figure 5.3.

- The Lion worm first makes an explicit TCP connection attempt to destination port 53. A successful connection indicates the reachability and possible vulnerability of the selected target (Phase 1). This connection, if established, is then immediately torn down without transmitting any payload.

- Another TCP connection to the same destination port is then established. This time, certain exploitation code is sent (Phase 2).

- If the exploitation is successful, the shell script, which is transmitted together with the exploitation code, will be executed to retrieve a worm replica from the infecter (Phase 3).

Deviation from the normal access to DNS lookup service is observed: First, it is unlikely that the access would begin with a plain TCP connection with no payload. Second and more importantly, after the DNS lookup request, it is highly unusual that the BIND server *initiates* a TCP connection to the DNS client on an unusual port $27374/TCP$, followed by an HTTP session on this connection to transfer a file of $71,680$ bytes *from the client to the server*.

5.2.4 Behavioral Footprinting: a New Dimension of Worm Profiling

In general, to a (vulnerable) service, there exist intrinsic differences between a normal access to the service and a worm infection through the service:

First, during the exploitation phase of a worm infection session, a worm will attempt to misuse a vulnerable service in a way that is different from a normal access. Several recent works [104–106] have leveraged such difference to derive vulnerability models for worm defense.

Second, the replication phase of a worm infection session should *not* happen during a normal access to the vulnerable service. In sharp contrast, it will appear in every successful worm infection. As shown in Figure 5.2, the $4444/TCP$ connection and its encapsulated TFTP transmission will appear in every MSBlast worm infection. Similarly,

the $27374/TCP$ connection and its encapsulated HTTP session can be observed for every Lion worm infection (Figure 5.3).

Finally, the entire sequence of infection steps during an infection session characterizes the worm's behavior, and is highly unlikely to appear in normal traffic. Our experiments with real-world network traces result in *zero false positives*. Furthermore, for different worms exploiting the same vulnerable service, their sequences of infection steps are different. The reason is that different worms tend to have different exploitation means, replication idiosyncrasies, and payloads, even though they are exploiting the same vulnerability (Section 5.4.2).

Based on the observations above, we are motivated to adopt a worm's infection step sequence during an infection session to characterize and hence uniquely identify the worm. We call this new dimension *behavioral footprinting*, in contrast to the existing dimension of content-based fingerprinting. We emphasize that the two dimensions complement each other and they should be combined to overcome their own weaknesses (Section 5.5). Since behavioral footprinting does not rely on payload content analysis, it is by nature resistant to content-based mutation and encryption attacks (Sections 5.4.4).

5.3 Behavioral Footprint Representation and Extraction

In this section, we first define the behavioral footprint and its representation. A simple *pairwise alignment* algorithm is then presented to extract a behavioral footprint from the traces of two infection sessions. To increase robustness against more intelligent worms, we develop an advanced footprint extraction algorithm to derive a worm's behavioral footprint from multiple infection traces.

5.3.1 Behavioral Phenotype and Footprint

The term "*behavioral phenotype*" was originally coined in 1972 by Nyhan [107] to represent a behavior that was genetically determined in the same way as the physical features of a phenotype. If we denote a worm's infection steps as the worm's behavioral pheno-

types, the sequence of behavioral phenotypes manifested during the infection session will be defined as the worm's intrinsic behavioral footprint. From Section 5.2, the behavioral footprint uniquely reflects the behavioral characteristics of the worm (e.g., vulnerability exploitation, propagation, and payload).

Our algorithms to extract worm behavioral footprints are based on the sequence analysis techniques extensively adopted in bio-informatics areas. A common and important issue for bio-informatics research is to operate over a large sequence of strings such as DNA, RNA, and protein sequences to find certain pattern(s) among them. Notice that any type of protein is a sequence of amino acid sub-units and there are only 20 different amino acids, which constitute the whole alphabet for protein sequence analysis. Similarly, if we consider all possible behavioral phenotypes during the worm infection as the alphabet, the behavioral footprint of a worm can be represented as a sequence of characters in the alphabet. For example, the behavioral footprint of the MSBlast worm, based on the infection session in Figure 5.2, can be represented as $S_1\overleftarrow{S_1^A}A_1 \cdots R_1 S_2 \overleftarrow{S_2^A} A_2 \cdots \overleftarrow{U_1} U_1 \cdots R_2$, where the letters' definitions are as follows:

$$
\begin{aligned}
S_1 &: < TCP, 4581/infecter, 135/victim, SYN > \\
\overleftarrow{S_1^A} &: < TCP, 135/victim, 4581/infecter, SYN, ACK > \\
A_1 &: < TCP, 4581/infecter, 135/victim, ACK > \\
R_1 &: < TCP, 4581/infecter, 135/victim, RST > \\
S_2 &: < TCP, 4599/infecter, 4444/victim, SYN > \\
\overleftarrow{S_2^A} &: < TCP, 4444/victim, 4599/infecter, SYN, ACK > \\
A_2 &: < TCP, 4599/infecter, 4444/victim, ACK > \\
\overleftarrow{U_1} &: < UDP, 1552/victim, 69/infecter > \\
U_1 &: < UDP, 69/infecter, 1552/victim > \\
R_2 &: < TCP, 4599/infecter, 4444/victim, RST >
\end{aligned}
$$

The letters in the above footprint denote either TCP flows with different control bits (SYN, ACK, RST) or UDP/ICMP flows (U/I). The subscripts denote different flows. For

example, $\overleftarrow{S_1^A}$ or $\overleftarrow{S_2^A}$[4] represents the second step (with SYN and ACK bits set) in a normal three-way TCP handshake. Without ambiguity, a unique well-known *subsequence* can be shortened as a single character. For example, a TCP three-way handshake sequence (e.g., $S_i\overleftarrow{S_i^A}A_i$, $i = 1, 2$, in the previous sequence) can be simplified as C_i.

In this example, every character is a tuple of several fields: the character representing a TCP flow has four fields $< TCP, source\_port, dest\_port,$ TCP control bits $>$; the character denoting a UDP flow has three fields $< UDP, source\_port, dest\_port >$. As different infection sequences might involve different ports, a special *wildcard* field is introduced. Using the MSBlast worm as an example, the source ports (e.g., ports 4581, 4599, 1552 in S_1, S_2, $\overleftarrow{U_1}$, respectively) vary in different infection sessions while the destination ports are fixed (e.g., ports 135, 4444, 69 in S_1, S_2, $\overleftarrow{U_1}$, respectively). As such, the special wildcard field is used for the source port field. On the other hand, there exist some worms that might have a constant source port number (e.g., the Witty worm has a static UDP source port 4000), but a random destination port. In this case, the wildcard is used to represent the destination port field. Although a worm infection session usually involves only two hosts (the infecter and the victim), a *coordinated* worm infection may involve more hosts (e.g., downloading the worm replica from a third host). In this case, the wildcard field can be used to represent the infecter field.

The number of fields in a phenotype may not be fixed. Additional fields can be added to include other "context" information, such as the packet length, content sequence, or timing information relative to the previous step. The extensible nature of behavioral phenotype representation makes it easy to integrate worm characteristics from *other dimensions*. For example, the content-based signature of a worm can be added as an additional field in the behavioral phenotype, indicating the occurrence of the content during the corresponding infection step. Protocol compliance analysis and vulnerability-specific information can also be integrated to further improve the completeness of worm profiles.

[4]The arrow is used to mark the traffic flow direction and can be omitted when it is unambiguous.

5.3.2 Pairwise Alignment Algorithm

Using the behavioral footprint representation, we first present an algorithm to extract a worm's behavioral footprint from two infection sequences.

Given two infection sequences $\mathscr{F}_1 = x_1x_2\cdots x_n$ and $\mathscr{F}_2 = y_1y_2\cdots y_m$, a pairwise alignment algorithm is used to align these two sequences so that they could have the same length. Based on a pre-defined scoring matrix (e.g., a match yields 1 while a mismatch yields 0), the alignment algorithm inserts gaps, if necessary, to achieve maximum alignment of the two sequences. The maximum alignment is defined as the sum of terms for each aligned pair of characters $< x_i, y_j >$ within the sequences (representing similarity $s(x_i, y_j)$), plus terms for each gap (representing penalty, p). The similarity and gap penalty are defined as a part of the scoring matrix and may be specific to different scenarios. A global alignment scheme obtains the optimal alignment between two sequences while a local alignment scheme looks for the best alignment between subsequences. There exist two well-known dynamic programming algorithms, i.e., Needleman-Wunsch Algorithm [108] and Smith-Waterman Algorithm [108].

The idea in Needleman-Wunsch Algorithm is to build an optimal alignment using previous solutions or optimal alignments of smaller subsequences. A matrix $\mathscr{M}$, indexed by i and j with one index for each sequence, is iteratively constructed. The cell $\mathscr{M}(i, j)$ is the score of the best alignment between the initial segment $x_1x_2\cdots x_i$ of x up to x_i and the initial segment $y_1y_2\cdots y_j$ of y up to y_j. Initially, $\mathscr{M}(0, 0) = 0$, $\mathscr{M}(i, 0) = -ip$, $\mathscr{M}(0, j) = -jp$. Then, the matrix is iteratively filled from top-left cells to bottom-right cells based on Eqn.(5.1).

$$\mathscr{M}(i,j) = \max \begin{cases} \mathscr{M}(i-1, j-1) + s(x_i, y_j), & i \geq 1, j \geq 1 \\ \mathscr{M}(i-1, j) - p, & i \geq 1 \\ \mathscr{M}(i, j-1) - p, & j \geq 1 \end{cases} \tag{5.1}$$

Each case represents an option how the current $\mathscr{M}(i, j)$ cell is derived from one of the other three cells (above-left $[i-1, j-1]$, above $[i-1, j]$, or left $[i, j-1]$). Once

all values are calculated, the choices taken at each cell starting from the bottom rightmost one are traced back so that an optimal global alignment can be derived. An example alignment applying Needleman-Wunsch Algorithm to the Welchia worm [101] is shown in Figure 5.4.

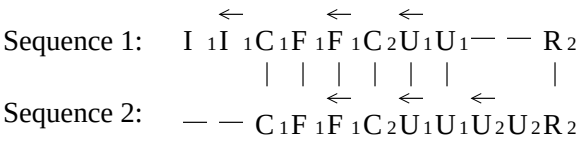


Figure 5.4. Global alignment with Needleman-Wunsch algorithm

Smith-Waterman Algorithm works similarly except that Eqn.(5.1) is modified for local alignment purpose. Particularly, one more case is added to reflect the possibility of starting a new local alignment. As such, the entry of $\mathscr{M}(i,j)$ is refined with the value $\max(\mathscr{M}(i,j),0)$ during the iterative calculation of Eqn.(5.1). The traceback is not performed from the bottom rightmost cell, but from the cell with the maximum value[5]. Another application of Smith-Waterman Algorithm is that if we associate a metric (e.g., number of matches) to the best alignment between subsequences of $\mathscr{F}_1$ and $\mathscr{F}_2$, the metric can be used to indicate the similarity between the two sequences. Smith-Waterman alignment is used in our next algorithm as a similarity-based scoring mechanism to build the relevant phylogenetic tree from a number of worm infection sequences.

Most existing self-propagating worms are primitive with no *behavior-polymorphic* capabilities. Our experiments in Section 5.4 show that pairwise alignment is highly effective in extracting their behavioral footprints. However, it is likely that many future worms will be polymorphic in both content and behavior, given that libraries [63, 109, 110] making code polymorphic are readily available. As a result, the pairwise alignment algorithm may not be capable of characterizing emerging advanced worms.

[5]A tie can be broken by arbitrarily choosing any cell with the maximum value.

5.3.3 Phylogenetic Tree Algorithm

In this section, we propose a robust algorithm to extract behavioral footprints of more advanced worms. The algorithm is based on our observation on the existence of *behavioral invariants*. Before presenting the algorithm, we first justify the existence of behavioral invariants – even in advanced worms.

5.3.3.1 Examining Behavioral Invariant

Similar to its counterpart, a content-polymorphic worm, a behavior-polymorphic worm exhibits varying behavior during different infection sessions. We consider single-vector worms that target one vulnerability, as a multi-vector worm can be considered as the combination of several single-vector worm variants. We have so far studied at least twenty self-propagating worms and their variants (including behavior-polymorphic worms we synthesize) that target different services on various operating systems. We have found behavioral invariant in each of them. Although we are not claiming that all worms exhibit behavioral invariants, we believe that a significant fraction of them do, as behavioral invariants typically result from (1) restrictions imposed for successful exploitations, (2) common components in each infection session (e.g., same payload and replication method of a worm), or (3) in some cases, a worm's idiosyncrasies in its exploitation means, replication mechanism, and self-carrying payload. We present two examples to illustrate how restrictions for successful exploitations determine a worm's behavioral invariants.

The first example is related to the OpenSSL heap-based buffer overflow exploited by the Slapper worm. As described in [10], the overflow is used twice by the worm to achieve a reliable infection. The first OpenSSL exploitation only attempts to locate the over-writable heap address within the vulnerable Apache address space, which is not predictable across all the servers. After the first exploitation, the acquired heap address is patched in the attack buffer within the second OpenSSL exploitation. It is expected that this two-phase exploitation enables a reliable infection. However, it has one more restriction: the two Apache processes handling these two exploitation connections should have

the same heap layout, and thus ensure the validity of the heap address obtained from the first exploitation connection to the second exploitation connection. To satisfy the restriction, the worm must first exhaust the Apache's pool of servers before actual exploitation. The exhaustion is achieved by opening a succession of 20 connections[6] so that two fresh Apache processes can be spawned to handle the two exploitation connections. As such, a reliable Slapper worm infection requires a series of resource-exhausting TCP connections and two additional exploitations. These requirements are essentially imposed as the *behavioral invariants* of the Slapper worm for successful infection.

The second example is the Slammer worm, which exploits a simple buffer-overrun vulnerability in MS SQL servers. To exploit this vulnerability, only a UDP packet with destination port 1434, packet type 4, and size larger than 60 bytes will successfully trigger the buffer overflow. This requirement results in the behavioral invariant of the Slammer worm, and is reflected in its behavioral footprint as: $< UDP, */*, 1434/*, payload :$ "$|04|$", $size > 60 >$.

5.3.3.2 Building the Phylogenetic Tree

From a collection of a worm's infection sequences[7], the worm's behavioral invariant can be extracted by advanced sequence analysis techniques. More specifically, pairwise alignment is first performed to derive their relative similarities (i.e., the Smith-Waterman alignment). Based on the similarities, a *phylogenetic tree* will be built to guide the final stage of multiple sequence alignment and to extract the behavioral invariants.

A phylogenetic tree was originally proposed to depict the evolutionary relationships of a group of life organisms. We build the phylogenetic tree to extract the intrinsic footprint subsequences or invariants that are embedded within a number of related infection sequences $\mathscr{F}_k$, $k = 1..n$. Some of the sequences may be mutated by inserting irrelevant subsequences or replacing a subsequence with another functionally-equivalent string. An algorithm called UPGMA [108] originally used in gene analysis has been applied to con-

[6]The number 20 is related to the *StartServers* entry in the Apache configuration file.

[7]These infection sequences can be collected by unleashing the worm in vGround (Chapter 4).

struct such a tree. Initially, each sequence $\mathscr{F}_k$ is considered as a cluster C_k. These clusters are iteratively grouped with the most related one so that, eventually, there is only one cluster left. The relevance or similarity between two clusters C_i and C_j is defined as:

$$d_{ij} = \frac{1}{\|C_i\| \, \|C_j\|} \sum_{p \in C_i, q \in C_j} d_{pq} \tag{5.2}$$

where $\|C_i\|$ and $\|C_j\|$ denote the number of sequences in clusters C_i and C_j. The value of d_{pq} is derived based on Smith-Waterman Algorithm. The clustering algorithm is described as follows:

PHYLOGENETICTREECONSTRUCTION$(\mathscr{F}_k, k = 1 \cdots n)$

1 $C \leftarrow \emptyset; T \leftarrow \emptyset$
2 **for** each sequence $\mathscr{F}_i$ $i \in 1..n$
3 **do**
4 Assign a cluster $C_i \leftarrow \{\mathscr{F}_i\}$
5 and add it into $C \leftarrow C \bigcup C_i$
6 Define a leaf N_i in T for $\mathscr{F}_i$
7 **for** each any other sequence $\mathscr{F}_j$, $j \in i + 1 \cdots n$
8 **do**
9 Calculate the similarity between $\mathscr{F}_i$ and $\mathscr{F}_j$
10 $d_{ij} \leftarrow$ SMITH-WATERMAN$(\mathscr{F}_i, \mathscr{F}_j)$
11 **while** $\|C\| \neq 1$
12 **do** Determine the two clusters C_i and C_j
13 s.t. d_{ij} is maximum
14 Define a new cluster $C_k = C_i \bigcup C_j$
15 and calculate d_{kl} for all l
16 Remove C_i and C_j from C, i.e., $C \leftarrow C - C_i - C_j$
17 Add C_k to C, i.e., $C \leftarrow C \bigcup C_k$
18 Add a parent node N_k to T with children N_i and N_j
19 **return** T

The calculation of d_{kl} in step 15 can be conveniently performed based on following equation:

$$d_{kl} = \frac{d_{il} \|C_i\| + d_{jl} \|C_j\|}{\|C_i\| + \|C_j\|} \tag{5.3}$$

The time and space complexity of the algorithm is $O(n^2)$, because there are $n - 1$ iterations, with $O(n)$ steps in each one.

5.3.3.3 Aligning Multiple Sequences

The phylogenetic tree is used to categorize the worm footprint sequences and to guide the actual alignment of the multiple sequences. Within the generated tree T, the leaves contain the raw footprint sequences while the intermediate nodes contain the sequences representing their children nodes. A recursive post-order tree traversal algorithm (shown below) can be applied to construct the representative sequences until the root of T is reached.

```
MULTIPLESEQUENCEALIGNMENT(T : PhylogeneticTree)
1  if T ≠ NULL
2    then MULTIPLESEQUENCEALIGNMENT(T.left);
3         MULTIPLESEQUENCEALIGNMENT(T.right);
4         if T.left ≠ NULL AND T.right ≠ NULL
5           then T.sequence ←
6                NEEDLEMAN-WUNSCH(T.left, T.right)
```

The sequence construction is based on global alignment using Needleman-Wunsch Algorithm (Section 5.3.2). An example run of the algorithm for a Welchia worm variant is illustrated in Fig 5.5. The sequence shown at the root of the tree

$$< variable > C_1 F_1 \overleftarrow{F_1} C_2 \overleftarrow{U_1} U_1 < variable > R_2$$

is the behavioral footprint of the Welchia worm.

5.4 Evaluation

In this section, we first describe our experiment environment, which is used to trap "live" worms and analyze historical worms. We then derive the worms' behavioral foot-

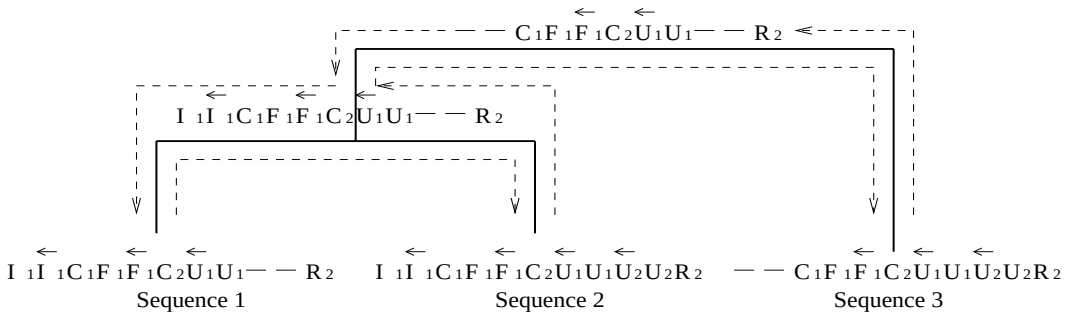


Figure 5.5. An example alignment of multiple worm infection sequences

prints. By comparing with content-based fingerprinting, we further demonstrate the effectiveness and robustness of behavioral footprinting for worm identification.

5.4.1 Experiment Environment

We perform our experiments on our integrated experiment platform, which consists of Collapsar honeyfarm (Chapter 3) and vGround playground (Chapter 4).

The honeypots in Collapsar run a variety of commodity operating systems, including RedHat Linux 7.2/8.0, Windows XP Home Edition, FreeBSD 4.2, and Solaris 8.0. All traffic from/to these honeypots are fully logged using *tcpdump*. As mentioned in Chapter 3, a number of real-world worms such as MSBlast [54], Enbiei [99], Welchia [101], and Sasser [102] have been captured live. We also use vGround to experiment with a number of historical worms and their variants, including the Lion worm [2], the Slapper worm [10], the Ramen worm [77], and the SARS worm [100]. For each experiment, the raw worm infection traces are recorded using *tcpdump*.

5.4.2 Extracting Behavioral Footprints

From the collected *tcpdump* log files, we first extract flow sequences relevant to worm infections. We develop a tool named *sneeze* for this purpose: All TCP/UDP/ICMP flow sequences in the log are extracted and packet reassembly or re-ordering is also performed

if necessary. The TCP/UDP/ICMP sequences are identified by their address pairs and ordered by the associated time-stamp. The duration and payload size of each flow is also automatically calculated by sneeze.

An example output from sneeze is shown in Figure 5.6. The raw trace is from a complete infection session of the Sasser worm, captured by Collapsar on May 1, 2004.

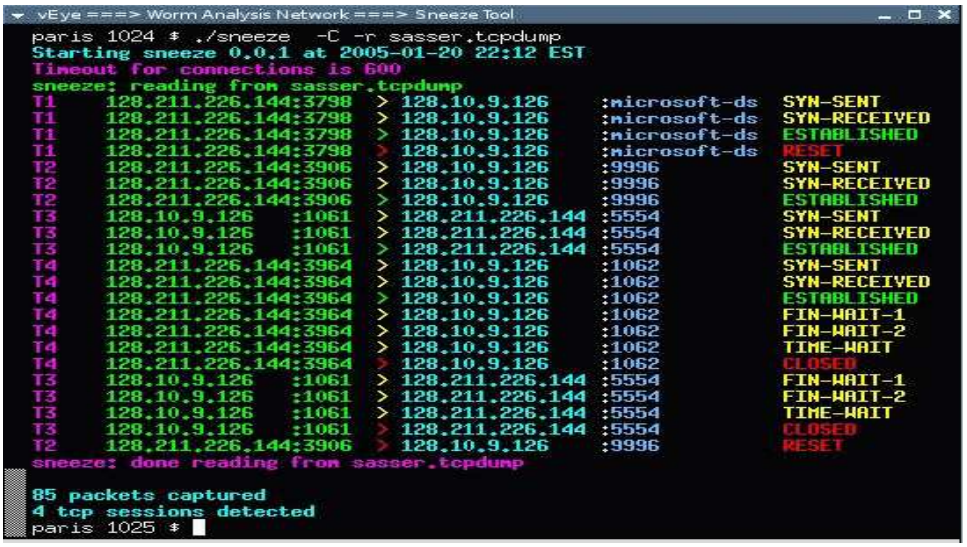


Figure 5.6. An example output of Sneeze

When analyzing TCP flows, sneeze is able to track relevant TCP states. Specifically, within the extracted TCP flows, any TCP control packets with SYN, ACK, FIN, or RST bit set are included in the resulting infection sequence. The TCP data packets are usually ignored. UDP and ICMP flows are also included in the sequence.

We apply the algorithms described in Section 5.3 on these infection sequences to extract behavioral footprints. The results are shown in Table 5.1. In the table, the letters denote either TCP flows with different control bits or UDP/ICMP flows. The letter C_i represents the three-way TCP connection handshake. Note that the same letter has different field values (e.g., different destination port numbers) in different footprints.

We are able to extract behavioral footprints for all worms examined. The Welchia worm[8] is similar to the MSBlast worm except that an initial ICMP probing packet is generated before actual infection and the second TCP connection ($\overleftarrow{C_2}$) is initiated from the victim with connect-back shell-code. Though MSBlast and Welchia exploit the *same* vulnerability, their behavioral footprints are *different*. The Enbiei worm exhibits a footprint similar to that of MSBlast but has a different worm binary and payload. The Sasser worm uses the *ftp* protocol ($\overleftarrow{C_3}$) to download the worm replica. Within the ftp session, a *PORT* primitive is initiated to start another reverse connect-back activity (C_4).

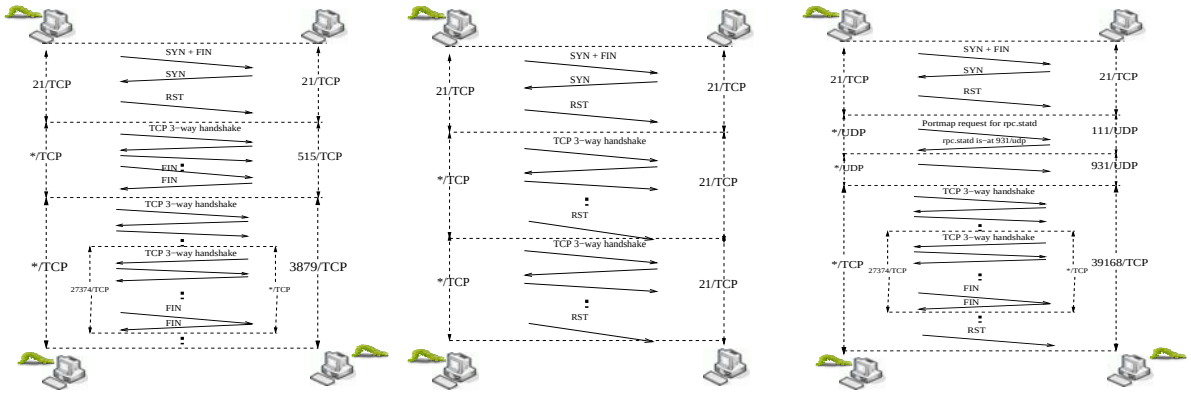
Table 5.1
Characterizing self-propagating worms with their behavioral footprints

Name	*Infection Vector*	*Behavioral Footprints Derived*
MSBlast	RPC-DCOM vulnerability (MS03-026)	$C_1R_1C_2\overleftarrow{U_1}U_1R_2$
Welchia	RPC-DCOM vulnerability (MS03-026) WebDAV vulnerability (MS03-007)	$I_1\overleftarrow{I_1}C_1F_1\overleftarrow{F_1}\overleftarrow{C_2}\overleftarrow{U_1}U_1\overleftarrow{U_2}U_2R_2$
Enbiei	RPC-DCOM vulnerability (MS03-026)	$C_1R_1C_2\overleftarrow{U_1}U_1R_2$
Sasser	LSASS vulnerability (MS04-011)	$C_1R_1C_2\overleftarrow{C_3}C4F_4\overleftarrow{F_4}F_3\overleftarrow{F_3}R_2$
Ramen	LPRng vulnerability (CVE-2000-0917) WU-FTPD vulnerability (CVE-2000-0573) NFS-UTILS vulnerability (CVE-2000-0666)	$S_1^F\overleftarrow{S_1}R_1C_2F_2\overleftarrow{F_2}C_3\overleftarrow{C_4}F_4\overleftarrow{F_4}$ $S_1^F\overleftarrow{S_1}R_1C_2R_2C_3R_3$ $S_1^F\overleftarrow{S_1}R_1U_1\overleftarrow{U_1}U_2C_2\overleftarrow{C_3}F_3\overleftarrow{F_3}R_2$
Lion	BIND vulnerability (CA-2001-02)	$C_1F_1\overleftarrow{F_1}C_2\overleftarrow{C_3}F_3\overleftarrow{F_3}R_2$
Slapper	OpenSSL vulnerability (CA-2002-23)	$C_1F_1\overleftarrow{F_1}C_2\overleftarrow{F_2}\prod_{i=3}^{22}C_iC_{23}C_{24}$
SARS	Samba vulnerability (CAN-2003-0201)	$U_1\overleftarrow{U_1}U_2\overleftarrow{U_2}C_1F_1C_2F_2\overleftarrow{F_2}C_3\overleftarrow{C_4}\overleftarrow{F_4}F_4R_3$

Table 5.1 also shows the footprints of several historical worms, which we run in vGround. The Ramen worm is a multi-vector worm with three infection vectors (IVs): LPRng (CVE-2000-0917), wu-ftpd (CVE-2000-0573), and nfs-utils (CVE-2000-0666) [9]. The three footprints of Ramen worms are also illustrated in Figure 5.7. We observe that an initial TCP control packet with SYN and FIN bits (S_1^F) set, source port 21, and

[8]The Welchia worm is a multi-vector worm, which takes advantage of two vulnerabilities, i.e., the RPC-DCOM vulnerability (MS03-026) and WebDAV vulnerability (MS03-007). Due to the lack of the vulnerable IIS server in our environment, the WebDAV-based infection cannot be reproduced.

[9]The exploitation of the wu-ftp IV is flawed and fails to result in a successful infection.



(a) The behavioral footprint of the Ramen worm exploiting the LPRng IV

(b) The behavioral footprint of the Ramen worm exploiting the WU-FTPD IV (w/ flawed implementation)

(c) The behavioral footprint of the Ramen worm exploiting the RPC.STATD IV

Figure 5.7. Behavioral footprints of the Ramen worm – a multi-vector worm

destination port 21, is used to probe the victim in all three footprints. The other three worms, Lion, Slapper, and SARS worms, are single-vector worms. The SARS worm is a multi-platform worm, which is able to spread across various platforms (e.g, Debian 3.0, Gentoo 1.4.x, Mandrake 8.x/9.0, Redhat 6.x/7.x/8.0/9.0, Slackware 8.x/9.0, SuSE 7.x/8.x, FreeBSD 4.x/5.0, NetBSD 1.5/1.6, and OpenBSD 3.2).

5.4.3 Advantage of Behavioral Footprinting

In this section, we demonstrate the advantage of behavioral footprinting using trace-driven worm identification experiments. More specifically, sneeze (Section 5.4.2) is modified to serve as a worm identification tool using worms' behavioral footprints. We use a 7-hour trace ($80M$ containing 3 worm infection sessions) collected by Collapsar. For comparison, we also use *snort*, a popular open-source content-based IDS, on the same

trace [10]. Sneeze is able to identify all three worm infections in the trace with zero false positives. Detailed results from snort and sneeze are shown in Table 5.2 and Figure 5.8, respectively.

Table 5.2
Worm detection with content fingerprints

	Snort Signature	*# Alerts*	*# Sources*	*# Dests*
1	NETBIOS DCERPC ISystemActivator path overfl ow attempt little endian	539	12	201
2	NETBIOS SMB-DS Session Setup And X request unicode username overflow attempt	15	1	1
3	NETBIOS SMB-DS DCERPC NTLMSSP asn1 overfl ow attempt	14	2	1
4	ICMP Source Quench	28	28	1
5	ICMP redirect host	27	1	1
6	TFTP Get	24	1	4
7	ICMP Large ICMP Packet	3	2	2
8	ICMP PING CyberKit 2.2 Windows	307551	33	153549
9	ICMP Destination Unreachable Communication Administratively Prohibited	156	2	1
10	SCAN UPnP service discover attempt	30	1	1
11	NETBIOS SMB-DS IPC$ share unicode access	6	3	1

As Table 5.2 shows, snort performs reasonably well in recognizing various RPC DCOM buffer overflow attempts, and in reporting numerous alerts for “ICMP PING CyberKit 2.2 Windows”. However, these alerts are raised separately even if they are caused by the same worm. On the other hand, sneeze identifies 3 successful worm infections and also reports 2 unsuccessful worm infections. In-depth analysis shows that one of the unsuccessful worm

[10]The signature database used in snort has been updated to contain the *latest* content fingerprints for known intrusions.

TIME	DUR.	SRC:PORT <-> DST:PORT	SERVICE	BYTES	SIGNATURE
Thu Nov 27 02:48:57 2003	8 s	81.168.168.127:4030 -> 128.10.9.127:135	135/TCP	1896 bytes	
Thu Nov 27 02:49:05 2003	25 s	81.168.168.127:4043 -> 128.10.9.127:4444	krb524/TCP	671 bytes	Enbiei Worm [sid]
Thu Nov 27 02:49:05 2003	21 s	128.10.9.127:1036 <-> 81.168.168.127:69	tftp/UDP	12134 bytes	
Thu Nov 27 04:58:24 2003	10 s	204.188.17.242:42948 -> 128.10.9.127:135	135/TCP	1908 bytes	MSBlaster/Enbiei Worm? [sid]
Thu Nov 27 04:58:35 2003	25 s	204.188.17.242:43362 -> 128.10.9.127:4444	krb524/TCP	349 bytes	
Thu Nov 27 06:08:24 2003	7 s	208.29.230.14:56429 -> 128.10.9.127:135	135/TCP	1572 bytes	MSBlaster/Enbiei Worm? [sid]
Thu Nov 27 06:08:30 2003	0 s	208.29.230.14:57021 -> 128.10.9.127:4444	krb524/TCP	78 bytes	
Thu Nov 27 07:23:46 2003	2 s	66.66.221.210:4581 -> 128.10.9.127:135	135/TCP	1896 bytes	
Thu Nov 27 07:23:49 2003	17 s	66.66.221.210:4599 -> 128.10.9.127:4444	krb524/TCP	666 bytes	MSBlaster Worm [sid]
Thu Nov 27 07:23:49 2003	11 s	128.10.9.127:1552 <-> 66.66.221.210:69	tftp/UDP	6372 bytes	
Thu Nov 27 08:03:55 2003	0 s	128.10.16.220 -> 128.10.9.127	ICMP	64 bytes	
Thu Nov 27 08:03:55 2003	0 s	128.10.9.127 -> 128.10.16.220	ICMP	64 bytes	
Thu Nov 27 08:03:55 2003	0 s	128.10.16.220:1567 -> 128.10.9.127:135	135/TCP	436 bytes	
Thu Nov 27 08:03:55 2003	4 s	128.10.9.127:3912 -> 128.10.16.220:707	707/TCP	1686 bytes	Welchia Worm [sid]
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3953 <-> 128.10.16.220:69	tftp/UDP	40 bytes	
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3953 <-> 128.10.16.220:1605	1605/UDP	20196 bytes	
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3954 <-> 128.10.16.220:69	tftp/UDP	40 bytes	
Thu Nov 27 08:03:56 2003	0 s	128.10.9.127:3954 <-> 128.10.16.220:1606	1606/UDP	10488 bytes	

Figure 5.8. Worm detection and identification with behavioral footprints

infections generated a wrong address (*192.168.1.59*) to download the worm replica while the other unsuccessful infection incurred a flawed exploitation in binding the command shell service. Since the *tftp* protocol is used by all these worms, Table 5.2 reports four alerts with message "TFTP GET" while Figure 5.8 further associates one tftp with the Enbiei worm, one tftp with the MSBlast worm, and the other two tftps with the Welchia worm, in which one tftp session is used to download the file $DLLHOST.exe$ (the worm payload) and the other tftp session is for $SVCHOST.exe$ (a tftpd daemon). The comparison demonstrates the advantage of behavioral footprinting for worm identification.

5.4.4 Robustness of Behavioral Footprinting

In this section, we further compare the robustness of behavioral footprinting when facing three different types of mutation attacks.

5.4.4.1 A Content-Mutation Attack

In this experiment, we examine the robustness against a simple content-mutation attack using the Slapper worm.

In snort, there are two Slapper-related signatures shown in Table 5.3. A vGround with 100 virtual nodes is created and one instance of the original Slapper worm is unleashed in the vGround. A *tcpdump* trace is generated by the vGround. From the trace, snort reports two "MISC OpenSSL Worm Traffic" alerts and five "WEB-MISC Bad HTTP/1.1 Request" alerts.

Table 5.3
Snort signatures for the Slapper worm

	Snort Signature	*Alert Message*
1	TERM=xterm	MISC OpenSSL Worm Traffic
2	GET / HTTP/1.1	WEB-MISC Bad HTTP/1.1 Request, Potentially Worm Attack

We conduct another experiment by performing a simple mutation of the Slapper worm content: replacing the string "TERM=xterm" with "TERM=linux" and "GET / HTTP/1.1" (the banner grabbing routine) with "GET / HTTP/10." The same vGround is used to experiment with the modified Slapper worm. Once the content is mutated, no alert is generated by snort. Others' recent work [97] has also confirmed the ineffectiveness of content-based signatures under content mutation attacks.

Behavioral footprinting is not affected by this attack. In both cases, sneeze is able to identify the same infection sequence of the Slapper worm. As shown in Figure 5.9, the Slapper worm first opens a normal TCP connection ($C_1 F_1 \overleftarrow{F_1}$) against port 80 checking the reachability of remote host; It then issues an invalid HTTP GET request ($C_2 \overleftarrow{F_2}$, *half-close*; containing the second content signature used in snort) to grab the server banner and query the version of the web server. It establishes 20 simultaneous plain TCP connections ($\prod_{i=3}^{22} C_i$, opened without any payload and never shutdown) on port 443 to prepare for the two following exploitations (C_{23}, C_{24}). Finally, a flurry of short packets ($> 10,000$) with each containing only 1 byte in its payload can be observed in the C_{24} TCP connection.

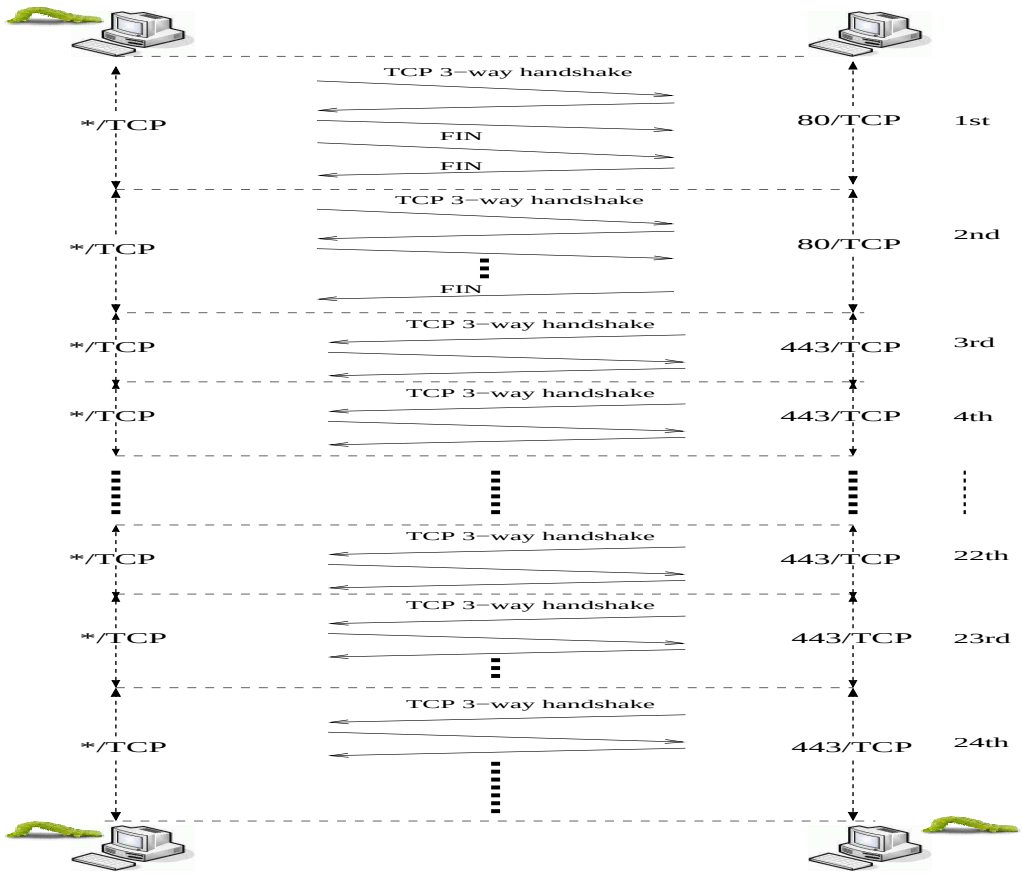


Figure 5.9. The behavioral footprint of the Slapper worm

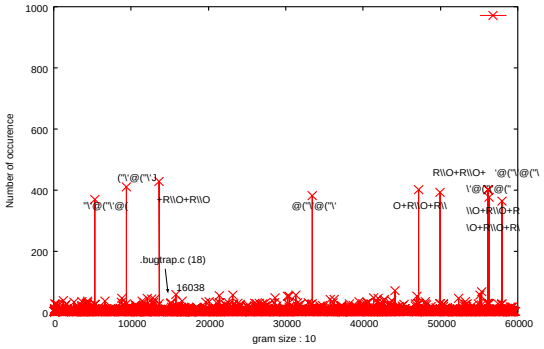


Figure 5.10. N-Gram analysis of the original Slapper worm

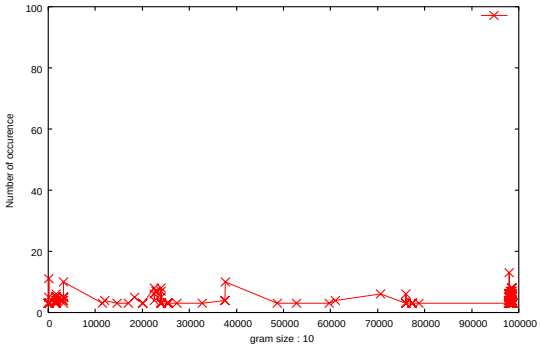


Figure 5.11. N-Gram analysis of a Slapper worm variant with encrypted transmission

5.4.4.2 A Traffic-Encryption Attack

In this experiment, we evaluate the robustness when worm traffic is encrypted. As pointed out in [10], the original Slapper worm is propagated through the transmission of a *uuencoded* version of the unencrypted Slapper source code. In this experiment, a synthesized Slapper variant is first instructed to encrypt the worm source file before propagation and later instructed to decrypt the file before compiling and executing the worm code in the victim. N-Gram analysis (counting the frequency of n-length combinations of bytes) is performed over two infection traces: one for the original Slapper worm with unencrypted worm source (Figure 5.10) and the other for the Slapper worm variant with encrypted source (Figure 5.11).

The N-gram analysis on the original Slapper worm trace shows several common strings with much higher frequency than other strings. However, these strings are not the same as the signature adopted in snort to detect Slapper worms. In fact, the signature used in snort "TERM=xterm" only happens twice according to the N-Gram analysis. It suggests that the most recurring content strings are not necessarily suitable for the signature. Once the transmission is encrypted, every string has equal probability of occurrence. On the other hand, the sequence $C_1 F_1 \overleftarrow{F_1} C_2 \overleftarrow{F_2} \prod_{i=3}^{22} C_i C_{23} C_{24}$ is exhibited in both the original and the synthesized Slapper worm infection sessions, which demonstrates the robustness of behavioral footprinting in the face of worms that encrypt their traffic.

5.4.4.3 A Behavior-Polymorphism Attack

The previous two experiments demonstrate the robustness of worm behavioral footprints against content-mutation and traffic encryption attacks. In this experiment, we further examine the robustness of behavioral footprints against a behavior-polymorphism attack.

Instead of following the behavior sequence shown in its original footprint, we craft a behavior-polymorphic Slapper worm variant, which is capable of (1) intentionally introducing an arbitrary number of irrelevant or miscellaneous sequences during an infection

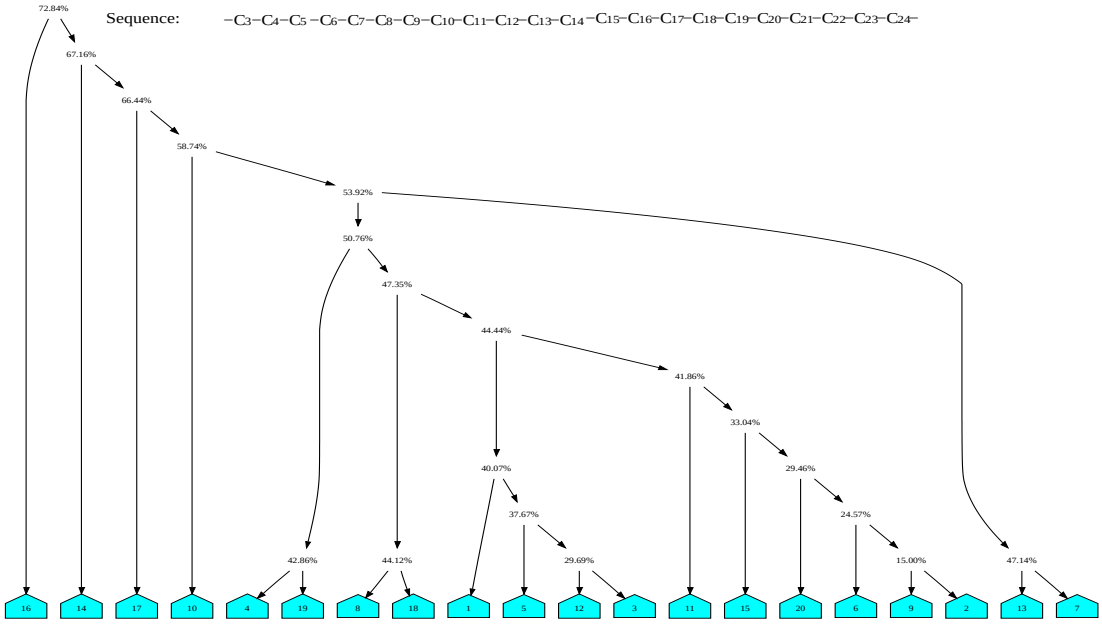


Figure 5.12. A phylogenetic tree built from 20 polymorphic behavioral sequences of the Slapper worm variant

session [11]; (2) intentionally introducing a random delay between two consecutive infection steps; and (3) intelligently changing the IP address to download the attack payload, including the worm replica. However, as restricted by the specific way to exploit the OpenSSL heap vulnerability (Section 5.3.3), the temporal order in the original behavior sequence has to be maintained to ensure successful infection.

A vGround with 1500 virtual nodes is created and all successful infection sessions are recorded for analysis. For brevity and readability, Figure 5.12 only shows the phylogenetic tree built from traces collected from 20 infection sessions. The numbers in the leaf nodes are session index numbers (from 1 to 20). The values in intermediate nodes indicate normalized similarity ($[0, 1]$) based on Smith-Waterman Algorithm (Section 5.3.2). A lower value indicates higher similarity between the two sub-clusters. The penalty used for

[11]Though the worm is able to initiate the connections (e.g., ICMP/TCP/UDP flows) to the victim node, it can not control the *reverse* direction as the victim is not under its control before a successful exploitation.

each gap through the algorithm is $p = -2$ and the scoring matrix used in Smith-Waterman Algorithm is

$$s(i,j) = \begin{cases} 2, & x_i = y_j \\ -1, & otherwise. \end{cases} \tag{5.4}$$

As we observe, the phylogenetic tree algorithm is still able to extract the most critical part of the original behavior sequence: $\prod_{i=3}^{22} C_i C_{23} C_{24}$, demonstrating the resilience of behavioral footprinting against behavior-polymorphism attacks.

5.5 Limitations

As a new dimension to characterize self-propagating worms, behavioral footprinting shows promise by identifying all worm infection incidents in our experiments. Behavioral footprinting is proposed to enrich a worm's profile along other dimensions such as content-based signature. In the following, we discuss limitations of behavioral footprinting. These limitations also call for further improvement along this new dimension as well as the integration of all dimensions for a multi-facet worm profile.

Behavior substitution attacks Our current pairwise alignment algorithm leverages a basic sequence alignment technique, or more specifically, a simple predefined scoring matrix (Section 5.3.2), to align worm infection sequences. An attacker might intentionally introduce a *substitutable* subsequence, which attempts to corrupt the alignment process while still achieving the goal of infection or propagation. For example, during the replication phase (Figure 5.1), different transport channels or even tunnels can be leveraged to retrieve the worm replica.

However, if we consider each behavior substitution as a possible mutation, such an attack is reminiscent of the classic challenge faced by biologists on how to optimally align gene sequences under possible mutations. Two popular scoring matrices used in gene sequence alignment, i.e., PAM [108] and BLOSSOM [108], have been constantly evolved to reflect newly-discovered mutations. Similarly, considering that the scoring matrix in

our algorithm is primitive as it simply returns 1 if two flows are fully matched, additional efforts are necessary to refine the scoring matrix. Fortunately, our application domain is different from the biological domain in that a worm usually can not evolve itself at runtime and has a relatively limited number of possible substitutions. In addition, a worm capable of substituting its infection steps is likely to be more bloated (reflected by its replica size) than a simple one. An over-bloated worm is more likely to be detected.

Behavior camouflaging attacks A worm author may attempt to inject fake steps into the infection sequences. After these fake steps have been included in the worm's behavioral footprint, the worm will stop exhibiting these fake steps. As a result, the behavioral footprint will experience a sudden increase in false negatives because a full match against the footprint will fail from now on. The fundamental solution is to identify and remove those fake steps using techniques such as semantic-level analysis [111, 112], a challenging on-going research topic. Another possible approach is to mitigate such attack by adopting partial instead of full footprint matching. However, a trade-off will have to be made to determine the confidence level of the partial matching to avoid the opposite, namely high false positives. Other dimensions (e.g., content-based signature) may provide complementary capability in this case.

5.6 Related work

Security researchers have explored various dimensions to capture worms' characteristics and apply them to worm identification. Among the most notable, content-based fingerprinting [67–69,96] has been extensively investigated and utilized to derive the most representative content sequences. To address the inconvenience of manually extracting worms' signatures, several systems such as Honeycomb [96], Autograph [68], EarlyBird [67], and Polygraph [69] have recently been proposed to automate the content-based signature extraction process. A content sequence is only able to detect worm activity within one infection step – most likely, the exploitation stage (Figure 5.1). Behavioral footprinting instead is proposed to capture a worm's infection steps during its entire infection session.

Another dimension, anomaly detection [104, 113–118], is based on the insight that worms are likely to exhibit anomalous behavior such as port scanning [115] and failed connection attempts [104, 113, 114], which are fundamentally different from normal behavior. Though such an approach has been demonstrated effective in detecting worm activities, it is not intended to *identify* worms. In other words, it mainly answers the question "is there a worm infection?", not the question "which worm is this?"

Other promising dimensions include vulnerability-specific characterization [104–106] and semantic-aware taintedness tracking [111,112,119–121]. Shield [106], Worm Vaccine [104], and Generic Exploit Blocking [105] propose the notion of vulnerability-specific signature and use it to accurately filter attack flows. TaintCheck [111], Minos [120], Vigilante [119], and other related systems [112, 121] enable the detection of unknown attacks by associating a tag to untrusted information sources and reporting an alert if a tainted instruction is executed. These schemes are generally applicable to detecting unknown attacks or intrusions. While capable of detecting the occurrence of an exploitation, they do not attempt to characterize the entire worm infection session where exploitation is only one of the infection phases.

Another related behavior-oriented approach [122] has been proposed. It focuses more on the inter-machine propagation pattern (tree) exhibited by worms as well as on the similar payload from one machine to another. It *assumes* the existence of worms' behavioral footprints, without justifying their existence and proposing the extraction methods, which is the focus of our work.

5.7 Summary

We have presented a worm defense mechanism, behavioral footprinting, to enrich the profile of a worm. Orthogonal and complementary to existing dimensions, behavioral footprinting characterizes a worm's infection steps and their temporal order. Robust algorithms are proposed to extract worm behavioral footprints. Our experiments with

real-world worms, in comparison with the content-based fingerprinting approach, clearly demonstrate the existence, uniqueness, and robustness of behavioral footprinting.

6 TRACKING MALWARE BREAK-INS AND CONTAMINATIONS WITH PROVENANCE-AWARE PROCESS COLORING

In this chapter, we present another malware defense mechanism, process coloring, which enables efficient, tamper-resistant tracing of malware break-ins and contaminations. This mechanism is based on a general framework for operating system level information flows [129], which we apply to malware defense using virtualization-based logging.

6.1 Introduction

In combat against Internet malware, the following tasks are critical to the understanding of malware exploitation details and to the recovery of an infected host from its contamination: (1) identifying the *break-in point*, namely the vulnerable, remotely accessible service via which the malware infects the victim machine and (2) determining all contaminations and damages inflicted by the malware during its residence on the victim machine. To perform these tasks, various intrusion analysis tools can be used [123–126]. For example, BackTracker [126] is an advanced forensic tool that traces back an intrusion starting from a "detection point" and identifies files and processes that could have affected that detection point. The tool takes the entire log file of the host as input for the back-tracking.

Log-based intrusion analysis tools face the following challenges: (1) Many tools [126–128] rely on an *externally-determined* detection point, from which a forensic investigation will be initiated towards the break-in point of the intrusion. However, a malware's possibly long "infection-to-detection" duration may mean days or even weeks later when such a detection point is identified. It is therefore desirable that the log data carry more information and provide "leads" to initiate more timely investigations. (2) Current operating systems lack a *provenance-aware* mechanism to pre-classify the log data before log analysis. Log data generated by the system may be of large volume. As reported in [126], log

data as large as $1.2GB$ can be generated within one day and need to be examined for an intrusion trace-back. The uncategorized bulk log data are likely to result in long duration and high overhead in malware investigation. Although human investigators can provide heuristics to reduce the log space to be examined, such heuristics may lead to inaccuracy or incompleteness in worm investigation results. (3) Many log-based tools do not address *tamper-resistant* log collection, which is essential in dealing with advanced malware. As shown in Section 6.2.3, a commonly adopted mechanism, i.e., syscall-wrapping for collecting system call traces, can be easily circumvented during an attack.

In this chapter, we present the design, implementation, and evaluation of *process coloring*, an efficient provenance-aware approach to worm break-in and contamination investigation. More specifically, process coloring associates a "color," a unique system-wide identifier, to each remotely-accessible server or process – a potential worm break-in point. The color will be either *inherited* directly by any spawned child process, or *diffused* indirectly through the processes' actions (e.g., *read* or *write* operations). As a result, any process or object (e.g., a file or directory) affected by a colored process will be tainted with the same color, as recorded in the corresponding log entry. Process coloring is based on Buchholz's general framework of *process labeling*, which pioneered in modeling and reasoning about operating system level information flows [129]. Process coloring naturally leads to two key advantages:

Color-based identification of a worm's break-in point All worm-infected processes and contaminated objects will be tainted with the same color as the original vulnerable service, which is exploited by the worm as the break-in point. By simply examining the color of any worm-related log entry or any worm-affected object, the break-in point of the corresponding worm can be immediately identified before detailed log analysis.

Natural partition of log data The colors of log entries provide a natural way to partition the log. To reveal the contamination caused by a worm, it is no longer necessary to examine the entire log file. Instead, only log entries with the same color as the worm's entry point will need to be inspected. Such partition can substantially reduce the volume of relevant log data, and thereby improve the efficiency of worm investigation.

The practicality and effectiveness of process coloring are demonstrated using a number of real-world self-propagating worms and their variants. For each of these worms, we are able to quickly identify the vulnerable networked service exploited by the worm. Moreover, reduction of inspected log data is achieved in every worm experiment. For example, for a detailed SARS worm [100] break-in and contamination investigation, only 12.1% of the entire log data needs to be inspected. Our prototype also addresses the important requirement of tamper-resistant log data recording. Virtualization techniques provide a better instrumentation facility than the system call hooking mechanism to safely obtain and collect internal states, including worm exploitation and contamination information. We adopt a technique similar to Livewire [130] and develop an extension to the UML virtual machine monitor (VMM) for tamper-resistant logging.

In this chapter, we mainly focus on the application of process coloring to the investigation of Internet worms. However, process coloring has potential in analyzing general malware intrusions and contaminations (Section 6.5.1). The rest of this chapter is organized as follows: Section 6.2 provides an overview of process coloring. Section 6.3 presents implementation details. Experimental evaluation results are presented in Section 6.4. Other applications and possible attacks are addressed in Section 6.5. Section 6.6 discusses related work. Finally, Section 6.7 summarizes this chapter.

6.2 Process Coloring Approach

6.2.1 Initial Coloring

Figure 6.1 shows a process coloring view of a networked host system running multiple servers. A unique system-wide identifier called *color* is assigned to each server process. The color assignment takes place after the server processes have started but before serving client requests. A worm breaking into the system will need to exploit a certain vulnerability of a (colored) server process. Because any action performed by the exploited process will lead to a corresponding *color diffusion* in the host (Section 6.2.2), the break-in and

contamination by the worm will be evidenced by the color of the affected processes and system resources and by the color of the corresponding log entries.

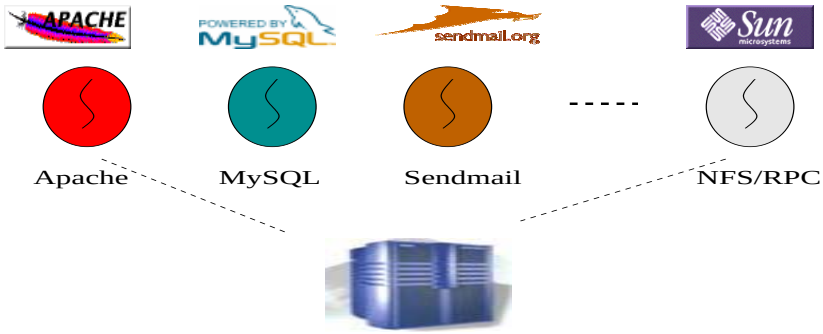


Figure 6.1. Process coloring view of a system running multiple servers

Each remotely-accessible service is performed by one or more active processes in the host. For example, the Samba service will start with two different processes *smbd* and *nmbd*; and both *portmap* and *rpc.statd* processes belong to the NFS/RPC service. Such processes can be assigned the same color. However, if we need to further differentiate each individual process (e.g., "which Apache process is exploited by the Slapper worm?"), different colors can be assigned to processes belonging to the same service. One benefit of such assignment is that it provides a finer granularity in log data partition. Alternatively, it is possible to define a color with two fields: a *major* field indicating the service and a *minor* field differentiating between individual working processes of the same service. For simplicity, we consider each color as having only one single field.

Although the process identifier (PID) uniquely identifies a process, it is *not* suitable for coloring purpose. First, PIDs are generated without any awareness of break-in points. Consider a zombie process, it is not possible to tell its break-in point simply by its PID or parent's PID. Second, it is possible that a process dynamically injects customized code (e.g., a whole library) into the code space of another active process. In this case, the PID is not capable of reflecting the impact of the former process on the latter. Such an attack has become popular on Windows platforms (e.g., the hxdef rootkit [131]) and there exist open-source libraries (e.g., Injectso [132]) that provide similar functionality for Linux and

Solaris platforms. In our design, a new field is defined in the operating system kernel to record the current colors of active processes.

6.2.2 Color Diffusion Model

After the service processes are initially colored, the colors will be diffused to other processes according to the operations performed by the processes. To reveal worm contamination, we are especially interested in process color diffusion via system-wide shared resources, such as files, directories, and sockets. For a worm to inflict contamination (e.g., backdoor installation), it needs to go through a number of system calls. Hence the process colors are diffused to the affected system resources via the operations performed by the system calls. Table 6.1 shows a simplified color diffusion model with respect to several abstract operations. A worm contamination example will be described later in this section.

Color diffusion follows Buchholz's *process labeling* framework [129], where audit information (defined as process label) is propagated and preserved in a system. Process color diffusion reflects the classic information flow models [133–136] in many aspects such as explicit/implicit information flows [136]. In this dissertation, we only consider the information flow through syscall interfaces, with processes as subjects and intermediate resources as objects. Other means such as using CPU utilization or disk space availability to convey information are beyond the scope of this dissertation. In the following, we describe two types of syscall-based color diffusion:

Direct diffusion involves one process directly affecting the color of another process. It can happen in a number of ways: (1) *Process spawning:* If a process issues the *fork*, *vfork*, or *clone* system call, a new child process will usually be spawned and it will inherit the color of the parent process. (2) *Code injection:* A process may use code injection (e.g. via *ptrace* system call) to modify the memory space of another process to change its functionality. The color of the injected process will be updated accordingly. (3) *Signal processing:* A process may send a special signal (e.g., the *kill* command) to another process. If received

Table 6.1
A simplified color diffusion model

Abstract Operation	*Color Diffusion*	*Description*	*Example Events/Actions*
$create < s_1, o >$	$color(o) = color(s_1)$	Subject s_1 creates a new object o	create, mkdir, link, mknod, pipe, symlink
$create < s_1, s_2 >$	$color(s_2) = color(s_1)$	Subject s_1 creates a new subject s_2	fork, vfork, clone, execve
$read < s_1, o >$	$color(s_1) \cup = color(o)$	Subject s_1 reads from object o	read, readv, recv, access, stat, fstat
$read < s_1, s_2 >$	$color(s_1) \cup = color(s_2)$	Subject s_1 reads from subject s_2	ptrace
$write < s_1, o >$	$color(o) \cup = color(s_1)$	Subject s_1 writes into object o	write, writev, truncate, chmod, chown, fchown, send, sendfile
$write < s_1, s_2 >$	$color(s_2) \cup = color(s_1)$	Subject s_1 writes into subject s_2	ptrace, kill,
$destroy < s_1, o >$	-	Subject s_1 destroys the object o	unlink, rmdir, close
$destroy < s_1, s_2 >$	-	Subject s_1 destroys the subject s_2	kill, exit

and authorized, the signal will invoke corresponding signal handling and thus affect the execution flow of the signaled process.

Indirect diffusion from process s_1 to s_2 can be represented as $s_1 \Rightarrow o \Rightarrow s_2$, where o is an intermediate resource (object). Various types of intermediate resources exist: some resources are dynamically created and will not exist after the process is terminated (e.g., UNIX sockets); other resources such as files can persistently exist and may later affect another process if that process acquires some input from these resources. To support indirect diffusion, the system data structure for an intermediate resource will be enhanced to record the influence of a process (i.e. its color). Later, when another process gets input

from the "tainted" resource, the process will be tainted the same color [1]. Common resource types supported in current Linux systems include files, directories, network sockets (including UNIX sockets), named pipes (FIFO), and IPC (messages, semaphores, and shared memory). We also note the existence of special system-wide control resources such as the system timer/clock, which could be used to indirectly influence another process. However, as the information flow through the influence is usually limited (i.e., a *low-bandwidth* channel) and we are not aware of any worm utilizing these special resources to affect other processes, we do not explicitly address them in this dissertation.

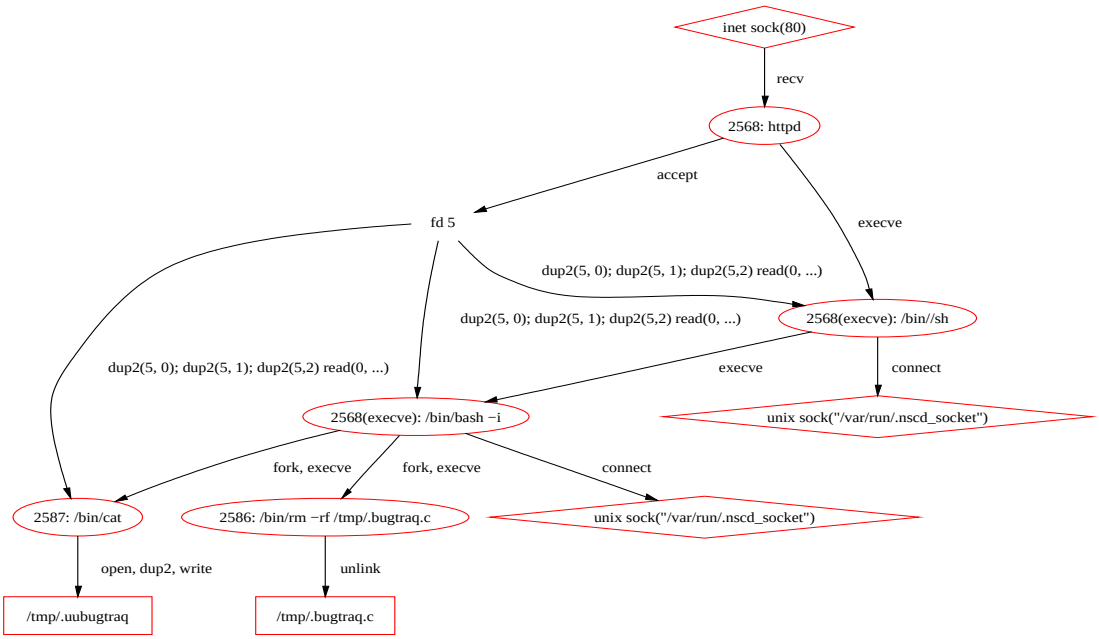


Figure 6.2. A coloring diffusion view showing the initial break-in by the Slapper worm

A worm example Figure 6.2 illustrates process color diffusion during the break-in of the Slapper worm [10], which exploits a vulnerable Apache service as its break-in point. In Figure 6.2, an oval represents a running process, a rectangle represents a file, and a diamond represents a network socket. The number inside the oval is the PID while the following string is the name of the process. Initially, all Apache "httpd" processes are

[1]To determine which input actually leads to an output, it is shown in [137] that such a problem is equivalent to solving the Halting problem [138, 139]. To be conservative, we consider that once a process reads from a tainted source, it will also be tainted.

colored “RED.” Right after the successful exploitation, the exploited “httpd” process (PID: 2568, color: RED) executes (*sys_execve* syscall) the program “/bin//sh” (2568, RED), which then executes (*sys_execve* syscall) the program “/bin/bash -i” (2568, RED). The “/bin/bash -i” process further spawns (by *sys_fork*) two child processes: process “/bin/rm -rf /tmp/.bugtraq.c” (2586, RED) and process “/bin/cat” (2587, RED) - their colors are inherited from their parent process via direct diffusion. Later on, the WRITE operation (*sys_write*) of process “/bin/cat” (2587, RED) updates the file (/tmp/.uubugtraq), which is thus tainted “RED.” As we will show in Section 6.4, this file will be used to generate (*sys_read* syscall) the worm process to infect other vulnerable hosts. Via indirect diffusion, the worm process will also be colored “RED.”

6.2.3 Log Collection

Process coloring employs system call (syscall) interception to generate log entries and tag them with process colors. As demonstrated in [24, 42, 126, 140–142], syscall interception is effective in revealing and understanding intrusion steps and actions. However, a simple syscall-based hooking mechanism may be vulnerable to the *re-hooking* attack, where attackers can easily avoid or even subvert [143] the log collection function. Figure 6.3 compares various hooking points for syscall intercepting. Figure 6.3(a) shows the original implementation in the current Linux kernel. Figure 6.3(b) demonstrates the popular *syscall wrapping* technique to intercept system calls. Syscall wrapping modifies the system call table and redirects the corresponding calls to its own implementation. Unfortunately, if the system call table is later modified, previous interception and redirection will be invalid. This type of syscall interception is used in [42, 124, 126], which are therefore vulnerable to this re-hooking attack. Figure 6.3(c) shows a more advanced technique, which intercepts system calls before or while invoking the system call dispatcher. Systrace [142] implements this type of interception by modifying the system call dispatcher and thus achieves better tamper-resistance. However, it is still possible [124] for an ad-

vanced intruder to avoid the interception if the corresponding syscall interrupt handler (e.g., "int $0x80$" in Linux) is hooked in the first place.

Our design is based on the virtual machine introspection technique [130]. Though similar to Figure 6.3(c), the interception happens *not* in the syscall dispatcher, but *on the virtual machine virtualization path*. As such, the interceptor is an integral part of the underlying virtual machine implementation (Section 6.3) achieving stronger tamper-resistance. Information about each intercepted system call (e.g. current process, syscall number, parameters, return value, and return address) forms a log entry, which is tagged with the color of the current process.

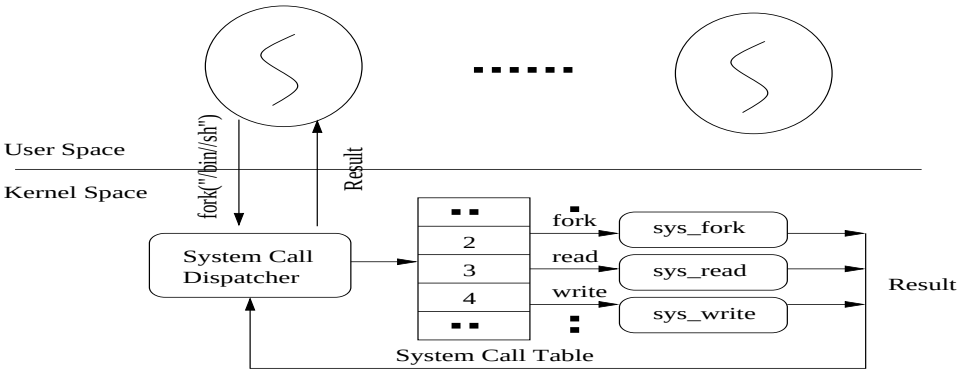
6.3 Implementation

In this section, we present key aspects of process coloring implementation. Our prototype leverages User-Mode Linux (UML), an open-source VM implementation where the guest OS runs directly in the unmodified user space of the host OS, and only considers the *ext2* file system[2]. To support process coloring, a number of key data structures (e.g., $task\_struct$, $ext2\_inode\_info$) are modified to accommodate the color information.

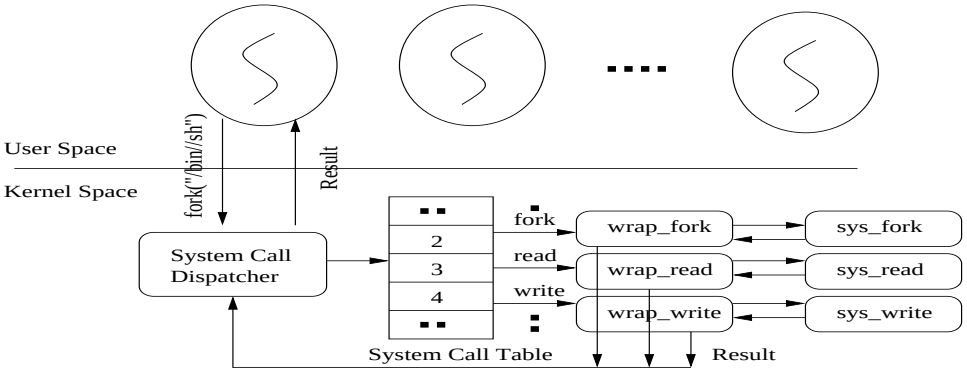
6.3.1 Process Color Setting

In our prototype, a new field *color* is added to the process control block (PCB) structure, i.e., $task\_struct$, in Linux kernel. To facilitate the setting and retrieval of the *color* field, two additional system calls (*sys_setcolor* and *sys_getcolor*) are implemented. There exists a possibility that these two new syscalls might be abused to undermine process coloring. Suppose their syscall interfaces are exposed, it would be easy for worm authors to add additional code to corrupt the color assignment. Though a strong authentication scheme may be used to restrict the usage of these two syscalls, it is not desirable as it essentially achieves security by obscurity. Our solution to this problem is to create and

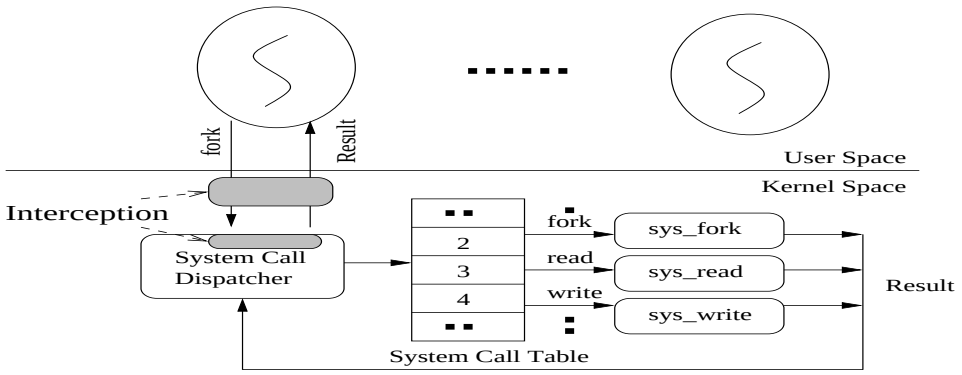
[2]We are currently implementing process coloring on another VM platform Xen [29] and we expect even better performance than our UML-based prototype due to Xen's para-virtualization approach.



(a) Original system calls



(b) Wrapped system calls



(c) Intercepted system calls

Figure 6.3. Different hooking points to intercept system calls

maintain a separate color mapping table within the syscall interceptor, which allows process color setting only within a certain time period after a service starts.

6.3.2 Color Diffusion

Direct diffusion If a new process is created by the *fork/vfork/clone* system call, it will inherit the color of its parent process. When a process is being manipulated via the *ptrace* system call, the diffusion of color will depend on the system call parameter. If the call has parameter *PTRACE_PEEKTEXT, PTRACE_PEEKDATA*, or *PTRACE_PEEKUSER*, the color(s) of the ptraced process will be diffused to the ptracing process. Conversely, if the call has parameter *PTRACE_POKETEXT, PTRACE_POKEDATA*, or *PTRACE_POKEUSER*, the color(s) of the ptracing process will be diffused to the ptraced process. For signal processing, the color(s) of the signaling process will be diffused to the signaled process. Finally, there are system calls (*sys_waitpid* and *sys_wait4*) that will lead to color diffusion from the child process to the parent process.

Indirect diffusion Indirection diffusion involves an intermediate resource (object). In principle, it is feasible that the system data structure for the corresponding resource be extended to record the color information. Among all possible intermediate resources, files and directories are the two most exploited by worms. As they are persistent resources, their colors also need to be persistently recorded. Intuitively, we can extend the corresponding *inode* data structure to accommodate the color attribute. However, adding a color field may essentially change the implementation of reading/writing files from/to a hard disk or even corrupt the underlying file system. After carefully examining all fields in current inode data structure, i.e., $ext2\_inode\_info$, we find that the field *i_file_acl* is intended to record the corresponding access control flags (ACL) but is *not* used in the *ext2* file system. In our current prototype, this field is leveraged to save the color value (represented as bitmap) of the corresponding file or directory. There is another possible field, i.e., *i_dir_acl*, which is intended to record the access control flags for the corresponding directory. However, this field has already been borrowed to serve as an additional 32-bit field for a 64-bit file

size representation for files larger than $4GB$. For non-persistent resources (e.g., IPC and network sockets), our current prototype only supports sockets, shared memory, and pipes. However, for other non-persistent resources, adding a new field is not too challenging.

6.3.3 Log Collection

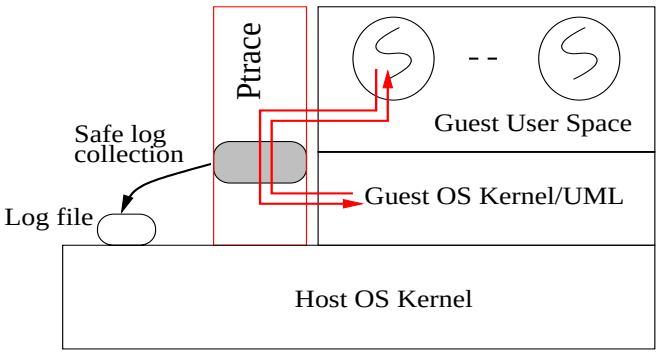


Figure 6.4. Tamper-resistant log collection by positioning the interceptor on the system call virtualization path

The log collection mechanism is based on the underlying virtual machine implementation, i.e. UML, as shown in Figure 6.4. UML adopts a system call-based virtualization approach and supports VMs in the user space of the host OS. Leveraging the capability of *ptrace*, a special thread is created to intercept the system calls made by any process in the VM, and to redirect them to the guest OS kernel. The interceptor for system call log collection is located on the system call virtualization path. Therefore, it is tamper-resistant from malicious processes running inside the VM. Moreover, once the interceptor has collected a certain amount of log data (e.g., $16K$), the log data will be pushed down to the host domain. One important benefit is that the analysis on the log file within the host domain will not interrupt the normal execution of the VM. This creates the possibility of external *runtime* system monitoring based on *colored* log data. Applications benefiting from this opportunity will be discussed in Section 6.5.1.

6.4 Evaluation

6.4.1 Evaluation of Run-Time Overhead

To measure the overhead introduced by process coloring, we perform a number of experiments using McVoy's LMbench [144], a suite of benchmarks targeting various subsystems of UNIX platforms. The experiments are conducted using a Dell PowerEdge 2650 server running Linux 2.4.18 with a 2.6GHz Intel Xeon processor and 2GB RAM. Three sets of experiments are performed: running LMbench on the original Linux kernel (Linux), on the unmodified UML kernel (UML), and on the modified UML kernel with process coloring capability (COLORING). The results are shown in Table 6.2.

Table 6.3(a) shows process operation overhead. Table 6.3(b) shows context switch times under varying number of processes and working set sizes. File system and virtual memory latency results are shown in Table 6.3(c). The results show that UML suffers a significant performance penalty caused by its user-level implementation. However, process coloring only incurs a small extra performance degradation beyond the original UML. The reason lies in the interceptor placement. By positioning the interceptor *within* the system call virtualization path, our prototype is able to avoid an additional context switch per system call, which is needed in other syscall interception schemes [141]. Also, the log data push-down is not performed upon every invocation of system call. Instead, an internal cache ($16K$) is maintained to amortize the overall disk write operations. Finally, we note that process coloring is not dependent on a specific VM platform. Moreover, we expect that the performance penalty caused by virtualization (not by the design of process coloring) be significantly reduced with more efficient VM platforms (e.g., Xen [29] with para-virtualization) and the upcoming architecture support for VMs (e.g., Intel's Vanderpool technology [89]).

Table 6.2
LMBench results showing low process coloring overhead

Confi guration	null cal	open close	signal handler	fork	exec
Linux	0.47	2.11	2.47	117	363
UML	11.0	146	28.5	4707	8016
COLORING	11.0	147	29.0	4910	8221

(a) Process-related times in μs

Confi guration	2p/0K	2p/16K	2p/64K	16p/16K	16p/64K
Linux	0.81	1.17	1.19	3.48	22.2
UML	9.11	8.75	9.67	16.7	46.7
COLORING	10.9	11.5	10.7	19.1	47.2

(b) Context switching times in μs

Confi guration	create (10K)	delete (10K)	mmap	page fault	select (100fd)
Linux	58.8	10.5	141.0	1.35	3.197
UML	226.2	90.2	772.0	15.0	21.9
COLORING	228.6	90.2	792.0	15.1	21.9

(c) File and VM system latencies in μs

6.4.2 Experiments with Real-World Worms

We evaluate the effectiveness of process coloring using a number of real-world Internet worms: Lion [2], Slapper [10], SARS [100], and their variants. Each worm experiment is conducted in vGround, our virtual malware playground (Chapter 4).

Table 6.3 shows key statistics of their respective log data. Each log file contains log entries collected during a 24-hour period, including both worm-related and normal service access entries. During each experiment, process coloring demonstrates its key benefits:

Table 6.3
Statistics of process coloring log in three worm experiments

	Lion Worm	*Slapper Worm*	*SARS Worm*
Exploited Service (CVE references)	BIND (8.2.2_P5-9) (CVE-2001-0010)	Apache (1.3.19-5) (CAN-2002-0656)	Samba (2.2.5-10) (CAN-2003-0201)
Time period being analyzed	24 hours	24 hours	24 hours
Number of log entries	129,386	293,759	166,646
Size of log data	8.0M	18.5MB	10.7MB
Number of worm-relevant log entries	66,504	195,884	19,494
Size of worm-relevant log data	3.9MB	12.2MB	1.3MB
Number of files "touched" by the worm	120,342	62	200
Percentage of worm-relevant logs	48.7%	65.9%	12.1%

(1) We are able to identify the worms' break-in points before performing detailed log analysis. The break-in points are the BIND server (bind-8.2.2_P5-9) for Lion worm, the Apache server (apache-1.3.19-5 with openssl-0.9.6b-8 package) for Slapper worm, and the Samba server (samba-2.2.5-10) for SARS worm. (2) The log data that need to be inspected for detailed worm investigation is only 48.7% (Lion worm), 65.9% (Slapper worm), and 12.1% (SARS worm) of the total logged events, respectively. We note that, because log entries are naturally partitioned by their colors, increasing background service accesses (i.e. accesses to unrelated services) in the experiments will further *reduce* the percentage of worm-related log. (3) As the worm break-in point (vulnerable service) is identified *before* log analysis, it is possible to further filter the log entries that record normal accesses

to the vulnerable service, which have *known and different* footprint from that of a worm infection.

6.4.2.1 Lion Worm Contamination Investigation

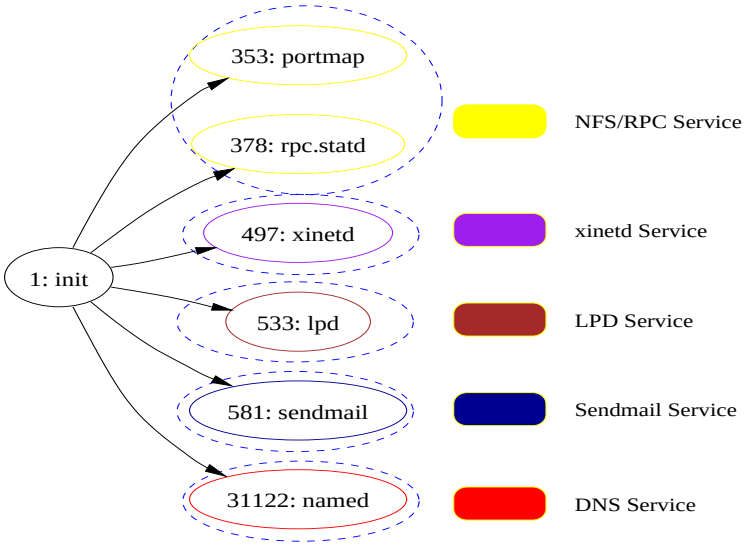


Figure 6.5. A process coloring view of a vulnerable system *before* Lion infection

Figure 6.5 shows a process coloring view of an *uninfected* system running a BIND server vulnerable to the Lion worm. There are also a number of other services hosted at the same system: NFS/RPC service (*portmap* and *rpc.statd*), printer service (*lpd*), and mail service (*sendmail*). A different color is assigned to each service. Process *named* has the color "RED." The Lion worm is unleashed from a different VM in the vGround[3]. After the experiment, we obtain a log file whose entries are conveniently partitioned by their colors. Among the "RED" entries whose provenance is the *named* process, we observe an abnormal event that a *shell* process was spawned. This is *one* of the contamination inflicted by the Lion worm. To further reduce the inspected log volume, entries generated by normal accesses to the BIND server from other legitimate VM clients in the vGround

[3]This "seed" worm is instrumented to target the vulnerable VM for infection. However, the transferred worm replica is unmodified.

are filtered. We then use the remaining "RED" log entries to derive a Lion worm contamination graph as shown in Figure 6.6.

We confirm that Figure 6.6 reveals *all* Lion worm contamination by comparing our results with a detailed Lion worm report [2]. The leftmost oval is the vulnerable *named* daemon (PID: 31122). After a successful exploitation of the *named* process, a worm replica is downloaded (Circle 2 in Figure 6.6). The worm then overwrites all HTML files named *index.html* in the system with a self-carried HTML file for web defacement (Circle 3). We observe from the log that the worm attempts to execute the file replacement *twice* - a detail *not* reported in [2]. The first attempt to replace files is within the shell code (PID: 31181) after executing the malicious buffer overrun code (Circle 2 and Circle 3). The second attempt happens when the driving script *./1i0n.sh* (PID: 31347) is executed (Circle 4). The worm then tries to initiate the next round of infection (Circle 4). In the thick dotted circle inside Circle 4, we find two "RED" *dangling* files *bind* and *bindx.sh*, which are introduced by the worm but never accessed by any worm-related process. Such an anomaly deserves a further investigation. A forensic analysis of the VM reveals that these two files contain the exploitation code for the BIND vulnerability. As there is only one VM running the vulnerable BIND service in the vGround, the worm cannot find another host to infect and the file *bindname.log* storing IP addresses of possible victims is empty. As a result, the exploitation code is never launched.

6.4.2.2 Slapper Worm Contamination Investigation

The Slapper worm experiment is conducted in a different vGround. We initially assign colors to service processes in an uninfected VM. The vulnerable Apache service is assigned "RED." Through direct diffusion, all spawned httpd worker processes are also colored "RED." A process coloring view of the system *before* the Slapper infection is shown in Figure 6.7. The experiment involves accesses to the other services as well as normal web accesses requesting a 2890-byte *index.html* file.

After the experiment, an examination on the log file shows a flurry of "RED" log entries (> 10000) within a very short period (1 minute) - an anomaly indicating a possible

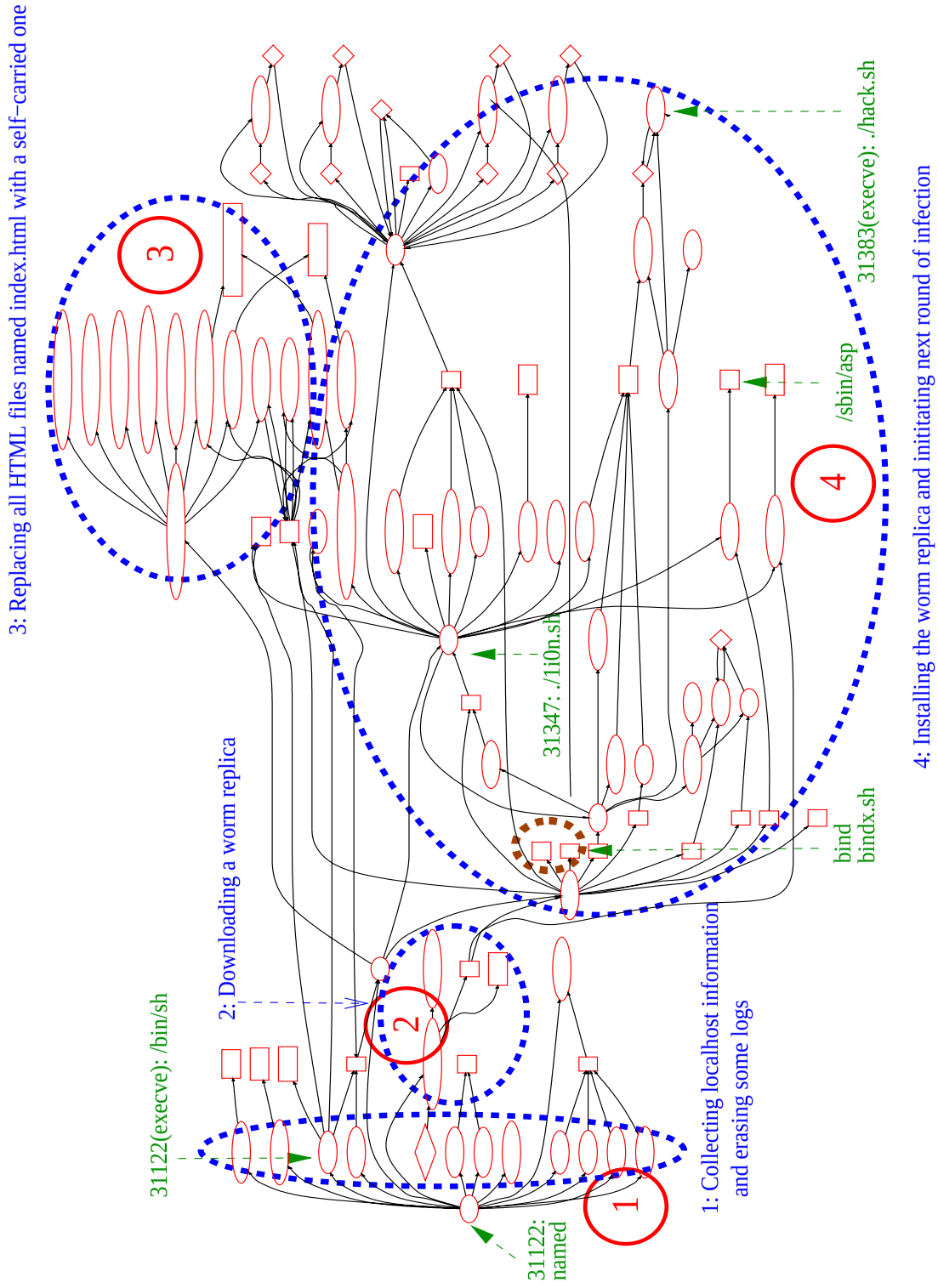


Figure 6.6. Lion worm contamination reconstructed from “RED” log entries

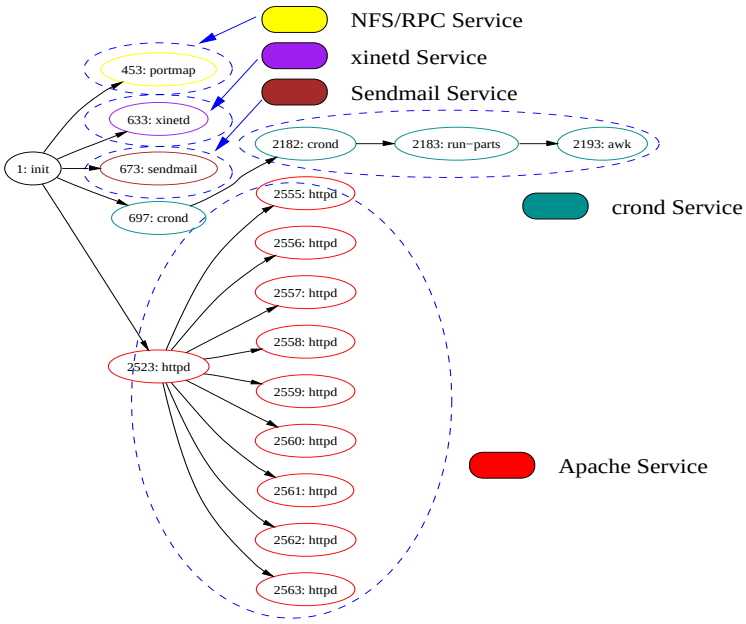


Figure 6.7. A process coloring view of a Slapper-vulnerable system *before* infection

infection. As the "RED" color is associated with the Apache web server, we select all "RED" log entries, which constitute 65.9% of the entire log file. A quick review of these log entries shows that the Slapper worm infection has a large and distinct footprint in the infected host. During the transmission of a Slapper worm, an *uuencoded* source file is sent from the infecter to the victim. More specifically, the sender issues a *sendch* call for *each byte* of the uuencoded file. Correspondingly, the receiver uses a *sys_read* for each byte received (total 94320 calls). Moreover, each encoded byte is then written (the *cat* command) to a local file named */tmp/.uubugtraq*, leading to another 94320 *sys_write* system calls. In sharp contrast, each normal web access only generates 15 log entries, recording the known normal sequence of Apache server actions. Therefore, we remove these ("RED") entries before constructing the Slapper worm contamination graph (Figure 6.8)[4].

[4]A general intrusion may mimic the normal sequence of service access actions [145]. However, it is more difficult for self-propagating worms to do so because their *outgoing propagation* behavior is semantically different from a normal service access.

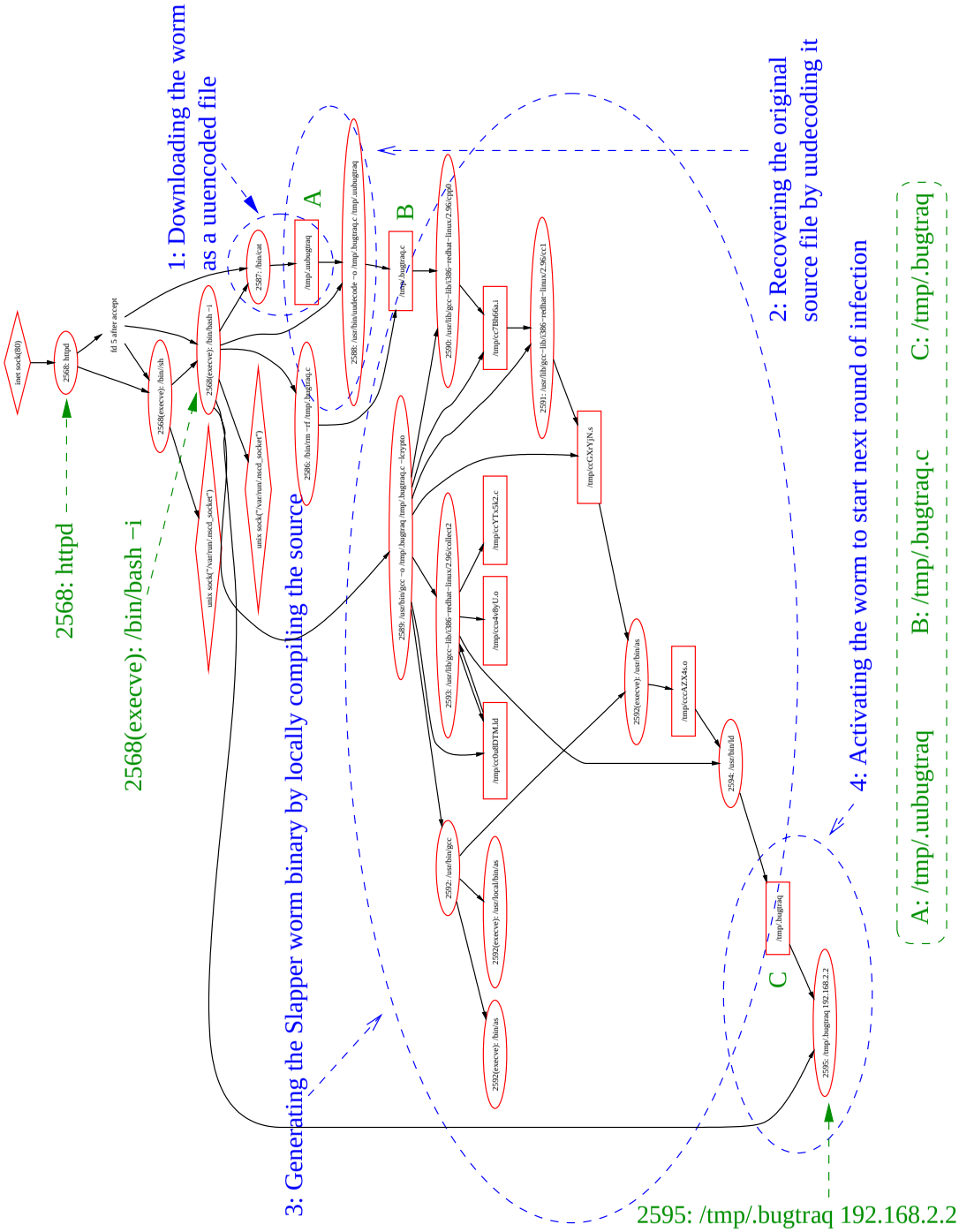


Figure 6.8. Slapper worm contamination reconstructed from “RED” log entries

By comparing our results with a detailed Slapper worm analysis [10], we confirm that Figure 6.8 reveals *all* contamination by the Slapper worm. We first observe that the

```
RED: 2523["httpd"]: 2_fork(void) = 2567 (rule 2)
RED: 2567["httpd"]: 214_setgid(48) = 0 (rule 16)
RED: 2567["httpd"]: 5_open("/etc/group", 0, 438) = 5 (rule 3)
...
RED: 2567["httpd"]: 5_open("/var/nis/NIS_COL...", 0, 438) = −2 (rule 3)
RED: 2567["httpd"]: 206_setgroups(1, 081eb4c0) = 0 (rule 49)
RED: 2567["httpd"]: 213_setuid(48) = 0 (rule 15)
...
BROWN: 673["sendmail"]: 5_open("/proc/loadavg", 0, 438) = 5 (rule 3)
BROWN: 673["sendmail"]: 192_mmap2(0, 4096, 3, 34, 4294967295, 0) = 1073868800 (rule 44)
BROWN: 673["sendmail"]: 3_read(5, "0.26 0.10 0.03 2...", 4096) = 25 (rule 4)
BROWN: 673["sendmail"]: 6_close(5) = 0 (rule 6)
BROWN: 673["sendmail"]: 91_munmap(1073868800, 4096) = 0 (rule 34)
...
RED: 2567["httpd"]: 102_accept(16, sockaddr{2, cac91f3a}, cac91f38) = 5 (rule 55)
RED: 2567["httpd"]: 3_read(5, "\1281\1\0\2\0\24...", 11) = 11 (rule 4)
RED: 2567["httpd"]: 3_read(5, "\7\0À\5\0\128\3\...", 40) = 40 (rule 4)
RED: 2567["httpd"]: 4_write(5, "\132@\4\0\1\0\2\...", 1090) = 1090 (rule 5)
RED: 2567["httpd"]: 3_read(5, "\128Ê", 2) = 2 (rule 4)
RED: 2567["httpd"]: 3_read(5, "\2\1\0\128\0\0\0...", 202) = 202 (rule 4)
RED: 2567["httpd"]: 4_write(5, "\128!\132ýFÞ\7B| ...", 35) = 35 (rule 5)
RED: 2567["httpd"]: 3_read(5, "\128!", 2) = 2 (rule 4)
RED: 2567["httpd"]: 3_read(5, "\0RØÔþn-A¸÷?(\1\...", 33) = 33 (rule 4)
RED: 2567["httpd"]: 4_write(5, "\128\129ôh¸\132«...", 131) = 131 (rule 5)
RED: 2567["httpd"]: 3_read(5, "(nil", 32769) = 0 (rule 4)
RED: 2567["httpd"]: 6_close(5) = 0 (rule 6)
```

Figure 6.9. Log excerpt showing the first exploitation of the Slapper worm attempting to get the overwritable heap address in the vulnerable Apache server. BROWN log entries are not related.

worm exploits an httpd worker process (PID:2568) to gain system access. After that, an uuencoded version of the worm source code is downloaded (Circle 1 in Figure 6.8) and *uudecoded* (Circle 2) to reconstruct the original code, which is then compiled (Circle 3) to generate the worm binary. The binary is executed (Circle 4) to attempt to infect other hosts. The collected log data further reveal that the exploitation of the Slapper worm is complex. Before the httpd worker process (PID: 2568) is exploited, 23 TCP connections have already been established with different http worker processes between the infecter and the victim. Among them, 21 connections have *no* payload; one connection is an invalid HTTP request, which turns out to be a request to obtain the Apache server version; the last connection has a short interaction as shown in the log excerpt in Figure 6.9. From [10], we know that one of the 21 plain connections is used to validate the reachability of the Apache server, while the other 20 connections are made for depleting the Apache server pool to make sure that the two subsequent exploitations will have the same heap layout. The

first exploitation aims at reliably deriving the over-writable heap address in the vulnerable Apache server. This heap address is then reused in the second exploitation. All these connections and interactions are recorded by “RED” log entries.

6.4.2.3 SARS Worm Contamination Investigation

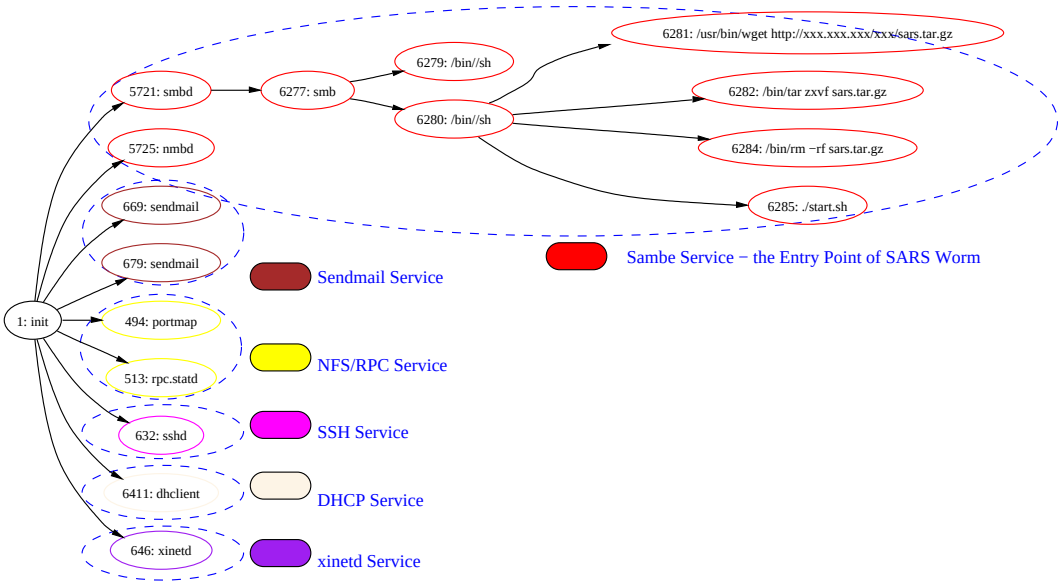


Figure 6.10. A process coloring view of a Redhat 8.0 system running multiple servers *after* it is infected by the SARS worm

The SARS worm is a multi-platform worm, which is able to propagate across all major distributions of Linux platforms (Redhat, Debian, SuSE, Mandrake, and Gentoo) and BSD platforms (FreeBSD, OpenBSD, and NetBSD). Since our current prototype is based on UML virtual machines, our experiment is conducted in a Linux-based vGround. The vulnerable Samba service is assigned “RED.”

After the experiment, only 12.1% of the entire log data are “RED,” because of the large number of log entries generated by other background services (e.g. *sendmail*, *sshd*, and *dhclient*) running in the Redhat 8.0 system. Derived from the “RED” log entries, Figure 6.10 shows the Redhat 8.0-based system *after* the infection of the SARS worm. Process *smbd* (PID: 5721) and process *nmbd* (PID: 5725) have the same color (“RED”) as

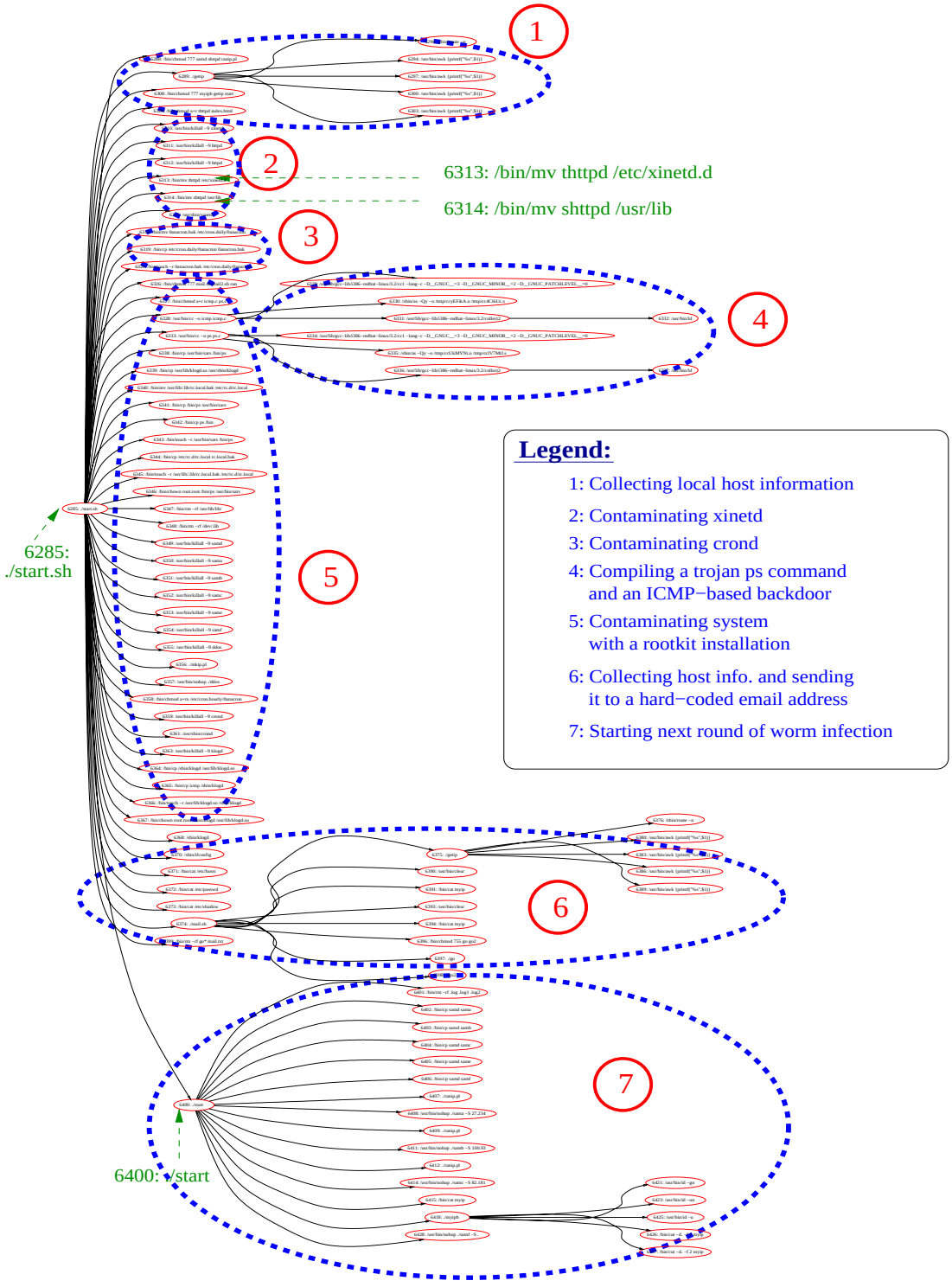


Figure 6.11. SARS worm contamination reconstructed from “RED” log entries

both of them belong to the Samba service. From the figure, it seems that the exploitation code contains some redundancy as two "/bin//sh" processes are executed and one quits immediately after its creation. These two shell processes are spawned when the buffer overrun code is executed.

Continuing from process *start.sh* (PID: 6285, shown in Figure 6.10), Figure 6.11 further reveals the contamination inflicted by the SARS worm. For readability, certain edges and nodes describing intermediate files are omitted. From the figure, we observe that the SARS worm contains a primitive user-level *rootkit* (Circle 4 and Circle 5 in Figure 6.11), whose purpose is to hide the existence of worm-related files, directories, active processes, and network connections. Also, the SARS worm inserts a number of backdoors such as a web server and an ICMP-based backdoor, which allow an attacker to access the infected host later. System-wide information such as host IP address, and configuration files including */etc/hosts* and */etc/passwd* is collected by the worm and sent to a hard-coded mail account (Circle 6). The integration of advanced payloads, such as the rootkit in the SARS worm, indicates a recent trend in the underground evolution of more stealthy self-propagating worms.

6.5 Other Applications and Possible Attacks

6.5.1 Other Applications

Malware investigation Process coloring can be naturally extended to support general malware investigation. The goal is to understand possible malicious actions and their impact on the infected system, which can further guide the recovery from the malware's contamination. In particular, process coloring is highly effective in exposing the following two anomaly points:

- *Color mixing:* Color mixing refers to the situation where a *different* color is diffused to an already-colored process. Based on the rationale of color diffusion, coloring mixing indicates that the process is influenced by another process with a different color. Considering the initial assignment of different colors to mutually *unrelated*

service processes, such cross-service influence is mostly likely an anomaly and warrants further investigation. For example, in one of our experiments, we run BIND and Apache services in one VM and let the Lion worm infect the VM via the BIND vulnerability. The Lion worm then contaminates the system by replacing *index.html* files with its own. We observe that log entries recording subsequent web accesses bear the colors of *both* BIND and Apache.

- *Dangling file:* A dangling file is created by a malware infection, but is not accessed during the same infection session. For example, if we re-examine Figures 6.6 and 6.11, some dangling files belong to rootkits/backdoors installed by worms: */sbin/asp* by the Lion worm (Figure 6.6) and */etc/xinetd.d/thttpd* and */usr/lib/shttpd* by the SARS worm (Figure 6.11). Though these rootkits are usually installed by stealthy worms/malware to hide their presence, identification of dangling files can actually help to reveal the presence of the rootkits.

Run-time monitoring and log visualization As mentioned in Section 6.3.3, the log push-down mechanism and color-based log partition provide a convenient means to externally monitor the running state of a networked server system, without interrupting the operations of the system. Process coloring can be used to identify possible anomalies revealed by log colors (e.g., color mixing, deviating log pattern for a particular color/service) during runtime and thus raise more timely alarms. We are currently extending our prototype with a log visualization tool, which makes it more intuitive for administrators to monitor system states.

6.5.2 Possible Attacks and Countermeasures

Jamming attack A worm could intentionally introduce many noise log entries to hide its actual intention. For example, a worm could invoke a large number of "innocuous" or unrelated syscalls to hide its real infection attempts. However, tactically speaking, these actions still need to be considered as a part of the worm's behavior in the infected system, even though they may not contribute to any real damage. Also, to the worm's

disadvantage, these noise log entries deviate from the normal log pattern of a specific color and will trigger an alarm. Finally, the capability of color-based identification of a worm's entry point is still valid under this attack, though it will take a more careful analysis to uncover the obfuscated intention.

Low-level attack The integrity of colors associated with active processes and intermediate resources are critical to worm investigation. As the current prototype maintains the color information within the kernel of the system under inspection, it is possible that this information may be manipulated through certain low-level attacks. For example, if the process color is associated with the *task_struct* PCB structure, a method called direct kernel object manipulation (DKOM) [146] can be leveraged to explicitly change the color value (e.g., by writing to the special device file */dev/kmem*). Fortunately, solutions such as CoPilot [147], Livewire [130], and Pioneer [148] have been proposed to address the issue of kernel integrity. Another possible counter-measure is to create a *shadow* structure, which is instead maintained by the virtual machine monitor (VMM) and is totally inaccessible from inside the VM. Compared with the current prototype, the shadow solution poses significantly greater challenge in deriving VM operation semantics from low-level information collected via virtual machine introspection, which may affect the accuracy and completeness of worm investigation results.

Diffusion-cutting attack It is possible that a worm might use a hidden channel to undermine the diffusion. For example, a worm could use an initial part of an attack to crack a weak password, which is later used in a *separate* session to gain system access and complete the rest of the worm contamination. Process coloring can track any action performed within each break-in, but it cannot automatically associate the second break-in with the first one. However, any anomaly within the second break-in will immediately expose the responsible login session, which may lead to identification of the cracked password. Based on the log data from the first break-in, the administrator may still be able to correlate those two disjunct break-ins.

Color saturation attack If a worm is aware of the coloring scheme, it might attempt to acquire more colors from different services right after its break-in. As a result, the

associated colors can not uniquely identify the break-in point. However, to the worm's *disadvantage*, the color saturation attack will immediately lead to an alarm of *color mixing* (Section 6.5.1) – an anomaly triggering further investigation. A color saturation attack does expose a weakness of our current prototype, which uses a single color field. Although our prototype is able to accommodate multiple colors (each bit in the color field represents a different color), it is not able to differentiate between an *inherited* color and a *diffused* color. The inherited color of a process can only be inherited from its parent and will not be changed by its own or others' behavior. The diffused colors reflect the color diffusions through its own or others' actions (e.g., *sys_read* and *sys_write*). With this distinction, the inherited colors can be used to partition the log data, while the diffused colors can be used to detect a color saturation attack and naturally identify all color-mixing points for further examination in affected partitions.

6.6 Related Work

The development of process coloring is inspired by the concept of transitive dependency tracking [149–151], which was originally proposed for failure recovery in fault tolerant systems. Process coloring also reflects various information flow models [129, 133–135], especially the operating system level information flows [129]. With these concepts and models as theoretical underpinnings, a spectrum of taint-based techniques have recently been proposed for different aspects of system security: Process coloring operates at the system call level to reveal worm break-ins and contaminations; TaintCheck [111] works at the instruction level to detect overwrite attacks and generate exploit signatures (Section 5.6); TaintBochs [152] focuses on the lifetime tracking of sensitive data (e.g., passwords) in a system. While sharing the same design philosophy, these techniques differ in their goals, design, implementation, and usage.

Process coloring can be integrated into existing log-based intrusion investigation tools [126, 153] so that they become provenance-aware. BackTracker [126] is able to automatically reconstruct the sequences of steps that occurred during an intrusion based on log

data. More specifically, starting with an external detection point (e.g., a corrupted file), BackTracker identifies files and processes that could have affected this detection point and displays chains of events in a dependency graph. The follow-up work [153] of Backtracker proposes a forward tracking capability that identifies all possible damage caused by the intrusion after the back-tracking session. Both BackTracker and its forward tracking extension require the entire log data as input. With process coloring enhancement, the break-in point of a worm can first be identified by the color of the detection point, and the volume of input log data will be reduced by color-based log partition, resulting in more efficient back-tracking and forward-tracking sessions. In addition, the colors and patterns of log entries may provide alerts at runtime, leading to more timely investigations.

Process coloring can also be applied to enhance file and transaction repair/recovery systems. The Repairable File Service [128] aims to identify possible file system level corruption caused by a root process, assuming that the administrator has already identified the root process that starts an attack or a human-involved error. It then uses the log data to identify the files that may have been contaminated by that process. The repairable file service implements a limited version of the forward tracking capability by only tracking file system-level corruption. Meanwhile, a similar technique [127] exists in the database area, which is capable of recording contamination at the transaction level and rolling back the damages if the transaction is later found malicious. This technique also requires external identification of malicious processes or transactions. Process coloring can enhance these techniques by tracking more sophisticated contamination behavior via color diffusion, raising anomaly alarms based on log colors and patterns, and achieving tamper-resistant log collection.

Recent advances in virtual machine technologies have created tremendous opportunities for intrusion monitoring and replay [24, 47, 94, 130], system problem diagnosis [154–156], attack recovery and avoidance [94,157], and data life-time tracking [152,158]. For example, ReVirt [94] is able to replay a system's execution at the instruction level. Time-traveling virtual machines such as [154–156] provide a highly effective means of re-examining and troubleshooting system execution or configuration. Process coloring

complements these efforts by leveraging virtual machine technologies for worm break-in and contamination investigation. In addition, process coloring, as an advanced logging mechanism, can be integrated into other VM-based networked systems to add provenance-awareness to these systems.

6.7 Summary

We have presented the design, implementation, and evaluation of process coloring, a malware defense mechanism that enables provenance-aware tracing of malware break-ins and contaminations. By associating a unique color to each remotely-accessible service and diffusing the color based on actions performed by processes in the system, process coloring achieves two key benefits: (1) color-based identification of a malware's break-in point before detailed log analysis and (2) color-based partitioning of log data. Process coloring improves log-based malware investigation tools by reducing the amount of log entries to be processed and by providing color-related "leads" for more timely investigation. Experiments with a number of real-world Internet worms demonstrate the practicality and effectiveness of process coloring.

7 CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this dissertation, we have presented an integrated, virtualization-based framework for malware capture, investigation, and defense. By creating a layer of indirection between physical resources and software systems, virtualization technology provides unique opportunities to address challenging problems in computer system security. More specifically, our framework has demonstrated the power of virtualization in malware research. Using the virtualization-based Collapsar honeyfarm (Chapter 3) as the front-end, we are able to capture current malware attacks from the Internet. With vGround, the virtual malware "playground" (Chapter 4) as the back-end, we are able to perform destruction-oriented experiments with captured real-world worms or malware. Moreover, our virtualization technique achieves strong tamper-resistance of log generation and collection in the implementation of process coloring (Chapter 6).

Based on the observations and insights obtained from this experiment platform, we gain unique advantages in designing and evaluating advanced malware defense mechanisms. In this dissertation, we present two such mechanisms: behavioral footprinting (Chapter 5) and process coloring (Chapter 6). Behavioral footprinting enriches a worm's profile by characterizing its infection behavior and improves worm identification accuracy and robustness. Process coloring enables malware forensics by improving the efficiency in tracking malware break-in points and contaminations, a capability important to post-attack investigation and recovery.

Meanwhile, our research experience indicates that virtualization is not panacea and could be attacked or abused by malware. For instance, the Agobot backdoor [159] will refuse to reveal its contamination behavior if it detects that it is running inside a virtual machine environment. Moreover, virtualization may be exploited to create stealthy or

even undetectable malware [160, 161]. These possibilities call for further improvement in virtualization technology and reflect the never-ending "arms race" between attackers and defenders.

7.2 Future Work

The integrated malware research framework presented in this dissertation has laid a solid foundation for future work, especially in advanced malware defense. In the following, we propose two future research topics:

- **Automating the malware detection, investigation, and defense workflow** Current vulnerability-patching process is manual and slow and can be easily outpaced by the fast spreading of malware. We believe that an automated workflow of malware detection, investigation, and defense holds great potentials in creating a malware defense infrastructure that is parallel to the cyber-infrastructure. We will investigate and evaluate a malware defense workflow that automates the following tasks: (1) operating a large-scale honeyfarm such as [47, 58] and capturing a new malware during its early stage of outbreak; (2) investigating the malware in a virtual playground environment and extracting its multi-dimensional profile; and (3) distributing the malware profile to end systems and network entities (e.g., gateways and routers) to prevent further propagation of the malware as well as to recover the damages already inflicted by the malware. Currently, it remains a challenge how to automate, optimize, and protect the workflow to make it widely deployable yet robust against advanced attacks.

- **Proactive malware defense** This dissertation only presents two reactive malware defense mechanisms and does not explore the solution space of proactive malware defense. Based on the weaknesses of current system and application software exposed by our experiments, we are motivated to investigate a proactive approach that hardens vulnerable programs and makes malware attacks hard to succeed in the first place. For example, system randomization techniques such as address space

layout randomization [162] and instruction set randomization [163, 164] randomize the execution environment of a program and therefore thwart a variety of malware infection attempts. We will investigate efficient, robust, and non-intrusive runtime system randomization mechanisms to overcome the limitations of current randomization techniques. By combining proactive and reactive techniques, we will be able to bring our malware defense capabilities to the next level.

LIST OF REFERENCES

LIST OF REFERENCES

[1] R. Figueiredo, P. Dinda, and J. Fortes. Resource Virtualization Renaissance. *IEEE Computer, Special Issue on Virtualization, 38(5)*, May, 2005

[2] Lion Worms. *http://www.sans.org/y2k/lion.htm*, 2001.

[3] Nimda Worms. *http://www.cert.org/advisories/CA-2001-26.html*, 2001.

[4] SoBig Worms. *http://www.cert.org/incident_notes/IN-2003-03.htm*, 2003.

[5] Rootkit. *http://en.wikipedia.org/wiki/Rootkit*.

[6] Backdoor. *http://dictionary.reference.com/browse/backdoor*.

[7] Trojans. *http://www3.ca.com/securityadvisor/newsinfo/collateral.aspx?cid=37734*.

[8] McAfee Anti-virus FAQ's. *http://www.hull.ac.uk/comp/FAQs/antivirus.html*.

[9] Slammer Worms. *http://www.cert.org/advisories/CA-2003-04.html*, 2003.

[10] F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf*.

[11] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[12] X. Jiang, D. Xu, and Y. M. Wang. Collapsar: A VM-Based Honeyfarm and Reverse Honeyfarm Architecture for Network Attack Capture and Detention. *Journal of Parallel and Distributed Computing, Special Issue on Security In Grid and Distributed Systems*, 2006.

[13] Y. M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. *Proceedings of the 13th Network and Distributed System Security Symposium*, February 2006.

[14] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual Playgrounds for Worm Behavior Investigation. *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, September 2005.

[15] X. Jiang and D. Xu. Behavioral Footprinting: A New Dimension to Characterize Self-Propagating Worms. *Department of Computer Science Technical Report CSD TR 05-027, Purdue University*, January 2005.

[16] X. Jiang, A. Walters, F. Buchholz, D. Xu, Y. M. Wang, and E. H. Spafford. Provenance-Aware Tracing of Worm Break-in and Contaminations: A Process Coloring Approach. *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, July 2006.

[17] X. Jiang and D. Xu. SODA: A Service-On-Demand Architecture for Application Service Hosting Utility Platforms . *Proceedings of The 12th IEEE International Symposium on High Performance Distributed Computing*, June 2003.

[18] X. Jiang and D. Xu. vBET: A VM-Based Emulation Testbed. *ACM Workshop on Models, Methods and Tools for Reproducible Network Research*, in conjunction with *ACM SIGCOMM 2003*, August 2003.

[19] E. H. Spafford. The Internet Worm Program: An Analysis. *Purdue CS Technical Report TR-CSD-823*, 1988.

[20] L.Spitzner. Honeypots: Tracking Hackers. *Addison-Wesley, 2003 ISBN: 0-321-10895-7*.

[21] VMware. *http://www.vmware.com/*.

[22] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[23] CERT/CC Overview Incident and Vulnerability Trends, CERT Coordination Center. *http://www.cert.org/present/cert-overview-trends/*, May 2006.

[24] The Honeynet Project. *http://www.honeynet.org*.

[25] L. Spitzner. Honeypot Farms. *http://www.securityfocus.com/infocus/1720*, August 2003.

[26] N. Provos. A Virtual Honeypot Framework. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[27] L. Spitzner. Honeytokens: The Other Honeypot. *http://www.securityfocus.com/infocus/1713*, July 2003.

[28] CERT Advisory CA-2002-01 Exploitation of Vulnerability in CDE Subprocess Control Service. *http://www.cert.org/advisories/CA-2002-01.html*, January 2002.

[29] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. Neugebauer A. Ho, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[30] Honeyclient Development Project. *http://www.honeyclient.org/*.

[31] L. Spitzner. Dynamic Honeypots. *http://www.securityfocus.com/infocus/1731*, September 2003.

[32] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[33] Snort-inline. *http://sourceforge.net/projects/snort-inline/*.

[34] D. Moore. Network Telescopes: Observing Small or Distant Security Events. *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[35] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. *Proceedings of the 10th ACM Conference on Computer and Communication Security*, October 2003.

[36] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). *RFC 1701*, October 1994.

[37] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation over IPv4 networks. *RFC 1702*, October 1994.

[38] Virtual PC. *http://www.microsoft.com/windowsxp/virtualpc/*.

[39] H. J. Hoxer, K. Buchacker, and V. Sieh. Implementing a User-Mode Linux with Minimal Changes from Original Kernel. *Linux-Kongress 2002, Koln, Germany*, September 2002.

[40] Tcpdump. *http://www.tcpdump.org*.

[41] Snort. *http://www.snort.org*.

[42] Sebek. *http://www.honeynet.org/tools/sebek/*.

[43] X. Jiang, D. Xu, and R. Eigenmann. Protection Mechanisms for Application Service Hosting Platforms. *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid*, April 2004.

[44] Y. Zhang and V. Paxson. Detecting Stepping Stones. *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[45] CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability. *http://www.cert.org/advisories/CA-2002-17.html*, March 2003.

[46] Linux Kernel Ptrace Privilege Escalation Vulnerability. *http://www.secunia.com/advisories/8337/*, March 2003.

[47] Collapsar. *http://www.cs.purdue.edu/homes/jiangx/collapsar*, December 2003.

[48] J. V. Miller. SHV4 Rootkit Analysis. *https://tms.symantec.com/members/AnalystReports/030929-Analysis-SHV4Rootkit.pdf*, October 2003.

[49] Iroffer. *http://iroffer.org/*.

[50] Napster. *http://www.napster.com/*.

[51] CERT/CC Vulnerability Note VU-298233. *http://www.kb.cert.org/vuls/id/298233*, March 2003.

[52] psyBNC. *http://www.psychoid.net/psybnc.html*.

[53] Microsoft Security Bulletin MS03-026. *http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS03-026.asp*, July 2003.

[54] MSBlast Worms. *http://www.cert.org/advisories/CA-2003-20.html*, 2003.

[55] Nachi Worms. *http://www.mycert.org.my/advisory/MA-055.082003.html*, 2003.

[56] CVE-2005-2087. *http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2005-2087*.

[57] Microsoft Security Bulletin MS05-014. *http://www.microsoft.com/technet/security/bulletin/ms05-014.mspx*.

[58] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity and Containment in the Potemkin Virtual Honeyfarm. *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[59] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Utility of Internet Sinks for Network Abuse Monitoring. *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[60] B. N. Chun, J. Lee, and H. Weatherspoon. Netbait: A Distributed Worm Detection Service. *Intel Research Berkeley Technical Report IRB-TR-03-033*, September 2003.

[61] Internet Storm Center. *http://isc.sans.org*.

[62] T. Ptacek and J. Nazario. Exploit Virulence: Deriving Worm Trends From Vulnerability Data. *CanSecWest/Core04 Conference, Vancouver*, April 2004.

[63] K2. ADMmutate. *CanSecWest/Core01 Conference, Vancouver* *http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz*, March 2001.

[64] Adore Worms. *http://securityresponse.symantec.com/avcenter/venc/data/linux.adore.worm.html*, 2001.

[65] Witty Worms. *http://securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html*, 2004.

[66] MyDoom Worms. *http://securityresponse.symantec.com/avcenter/venc/data/w32.mydoom.b@mm.html*, 2004.

[67] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design & Implementation*, December 2004.

[68] H. A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. *Proceedings of the 13th Usenix Security Symposium*, August 2004.

[69] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *Proceedings of the 26th IEEE Symposium on Security and Privacy*, May 2005.

[70] PlanetLab. *http://www.planet-lab.org*.

[71] K. S. Perumalla and S. Sundaragopalan. High-Fidelity Modeling of Computer Network Worms. *Proceedings of the 20th Annual Computer Security Applications Conference*, December 2004.

[72] Bro. *http://bro-ids.org*.

[73] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *Proceedings of the 5th ACM/USENIX Symposium on Operating Systems Design & Implementation*, December 2002.

[74] J. Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Proceedings of the 8th IEEE International Conference on Network Protocols*, November 2000.

[75] A. Sundararaj and P. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. *Proceedings of the 3rd USENIX Virtual Machine Technology Symposium*, August 2004.

[76] X. Jiang and D. Xu. VIOLIN: Virtual Internetworking on Overlay Infrastructure. *Technical Report CSD-TR-03-027, Purdue University* (also in *LNCS Vol. 3358, Springer)*, July 2003.

[77] Ramen Worms. *http://www.sans.org/y2k/ramen.htm*, February 2001.

[78] Internet Protocol V4 Address Space. *http://www.iana.org/assignments/ipv4-address-space*.

[79] Objdump. *http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html*.

[80] J. Nazario. *Defense and Detection Strategies against Internet Worms*. Artech House Publishers, ISBN: 1-58053-537-2, 2004.

[81] ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability. *http://www.securityfocus.com/bid/2302*, 2001.

[82] P. Szor. Fighting Computer Virus Attacks. *Invited Talk, the 13th Usenix Security Symposium (Security 2004), San Diego, CA*, August 2004.

[83] P. Craveiro. SANS Malware FAQ: What is t0rn rootkit? *http://www.sans.org/resources/malwarefaq/t0rn_rootkit.php*.

[84] PUD: Peer-To-Peer UDP Distributed Denial of Service. *http://www.packetstormsecurity.org/distributed/pud.tgz*, 2002.

[85] C. C. Zou, D. Towsley, W. Gong, and S. Cai. Routing Worm: A Fast, Selective Attack Worm Based on IP Address Information. *Technical Report, TR-03-CSE-06, University of Massachusetts, Amherst*, November 2003.

[86] C. Carella, J. Dike, N. Fox, and M. Ryan. UML Extensions for Honeypots in the ISTS Distributed Honeypot Project. *Proceedings of the 5th Annual IEEE Information Assurance Workshop*, June 2004.

[87] Santy Worms. `http://www.f-secure.com/v-descs/santy_a.shtml`, December 2004.

[88] Google Smacks Down Santy Worm. `http://www.pcworld.com/news/article/0,aid,119029,00.asp`, December 2004.

[89] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C.M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer, Special Issue on Virtualization Technology*, May 2005.

[90] The DETER Project. `http://www.isi.edu/deter/`.

[91] 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 the 5th ACM/USENIX Symposium on Operating Systems Design & Implementation*, December 2002.

[92] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *Proceedings of the 5th ACM/USENIX Symposium on Operating Systems Design & Implementation*, December 2002.

[93] I. Whalley, B. Arnold, D. Chess, J. Morar, and A. Segal. An Environment for Controlled Worm Replication & Analysis (Internet-inna-Box). *Proceedings of the 10th Annual Virus Bulletin Conference*, September 2000.

[94] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *Proceedings of the 5th ACM/USENIX Symposium on Operating Systems Design & Implementation*, December 2002.

[95] 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`, December 2004.

[96] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Communication Review*, January 2004.

[97] G. Vigna, W. Robertson, and D. Balzarotti. Testing Intrusion Detection Signatures Using Mutant Exploits. *Proceedings of the 11th ACM Conference on Computer and Communication Security*, October 2004.

[98] O. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. `http://www.cc.gatech.edu/~ok/w/ok_pw.pdf`.

[99] Enbiei Worms. *http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.f.worm.html*, 2003.

[100] SARS Worms. *http://www.patching.net/bbs/viewgooddoc.asp?id=25669\&bordid=4*, 2003.

[101] Welchia Worms. *http://securityresponse.symantec.com/avcenter/venc/data/w32.welchia.worm.html*, 2003.

[102] Sasser Worms. *http://www.microsoft.com/security/incident/sasser.asp*, 2004.

[103] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. *Proceedings of the 2nd ACM Internet Measurement Workshop*, November 2002.

[104] Arbor Networks: PeakFlow X. *http://www.arbornetworks.com/products_x.php*.

[105] C. Nachenberg. From AntiVirus to AntiWorm: A New Strategy for a New Threat Landscape. *Invited talk in the 2004 ACM Workshop on Rapid Malcode*, October 2004.

[106] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. *Proceedings of ACM SIGCOMM 2004*, September 2004.

[107] W. Nyhan. Behavioral Phenotypes in Organic Genetic Disease. *Pediatric Research 6:1-9*, 1972.

[108] R. Durbin, S. Eddy, and A. Krogh. *Biological Sequence Analysis*. Cambridge University Press, ISBN: 0521629713, 1998.

[109] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack Issue 0x3d*, 2003.

[110] M. Sedalo. Jempiscodes: Polymorphic shellcode generator. *http://securitylab.ru/tools/services/download/?ID=36712*, 2003.

[111] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[112] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[113] CounterStorm. *http://www.sysd.com*.

[114] Mazu Networks. *http://www.mazunetworks.com/*.

[115] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. *Proceedings of the 25th IEEE Symposium on Security and Privacy*, May 2004.

[116] M. E. Locasto, J. J. Parekh, A. D. Keromytis, and S. J. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. *Proccedings of the 6th Annual IEEE Information Assurance Workshop*, June 2005.

[117] S. Radosavac. Detection and Classification of Network Intrusions using Hidden Markov Models. *Master's Thesis, Institute for System Research, University of Maryland*, `http://techreports.isr.umd.edu/reports/2003/MS_2003-1.pdf`, May 2002.

[118] K. Wang and S. J. Stolfo. Anomalous Payload-Based Network Intrusion Detection. *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[119] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[120] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. *Proceedings of 37th International Symposium on Microarchitecture*, October 2004.

[121] M. Rinard, C. Cadar, D. Dumitran, D. Roy, and T. Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). *Proceedings of the 20th Annual Computer Security Applications Conference*, December 2004.

[122] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A Behavioral Approach To Worm Detection. *Proceedings of the 2004 ACM workshop on Rapid Malcode*, October 2004.

[123] E. Alata, M. Dacier, Y. Deswarte, M. Kaaniche, K. Kortchinsky, V. Nicomette, V.H. Pham, and F. Pouget. CADHo: Collection and Analysis of Data from Honeypots. *Proceedings of the 5th European Dependable Computing Conference*, April 2005.

[124] J. Grizzard, J. Levine, and Henry Owen. Re-Establishing Trust in Compromised Systems: Recovering from Rootkits that Trojan the System Call Table. *Proceedings of the 9th European Symposium on Research in Computer Security*, September 2004.

[125] G. H. Kim and E. H. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. *In Systems Administration, Networking and Security Conference III, USENIX*, 1994.

[126] S. T. King and P. M. Chen. Backtracking Intrusions. *Proceedings of the 19th Symposium on Operating Systems Principles*, October 2003.

[127] P. Ammann, S. Jajodia, and P. Liu. Recovery from Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering, Volume 14, Issue 5, 1167-1185*, September 2002.

[128] N. Zhu and T. Chiueh. Design, Implementation and Evaluation of Repairable File Service. *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, June 2003.

[129] F. Buchholz. Pervasive Binding of Labels to System Processes. *Ph.D. Thesis*, also as *CERIAS Technical Report 2005-54, Purdue University*, 2005.

[130] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, February 2003.

[131] hxdef. `http://hxdef.czweb.org`.

[132] Injectso: Modifying and Spying on Running Processes under Linux and Solaris. `http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt`.

[133] D. Bell and L. LaPadula. MITRE Technical Report 2547 (Secure Computer System): Volume II. *Journal of Computer Security, vol. 4, no. 2/3, pages 239-263*, 1996.

[134] D. R. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. *Proceedings of the 1987 IEEE Symposium on Security and Privacy, pages 184-194*, 1987.

[135] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM 19, 5 (May), 236-243*, 1976.

[136] J. A. Goguen and J. Meseguer. Security Policies and Security Models. *Proceedings of the 1982 IEEE Symposium on Security and Privacy, pages 11-20*, 1982.

[137] F. Buchholz and E. H. Spafford. On the Role of File System Metadata in Digital Forensics. *Journal of Digital Investigation*, December 2004.

[138] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungs Problem. *Proceedings of London Mathematical Society, Series 2, 42:230-265*, 1937.

[139] A. M. Turing. Correction to: On Computable Numbers, with an Application to the Entscheidungs Problem. *Proceedings of London Mathematical Society, Series 2, 43:544-546*, 1938.

[140] A. Goel, M. Shea, S. Ahuja, W. C. Feng, W. C. Feng, D. Maier, and J. Walpole. Forensix: A Robust, High-Performance Reconstruction System. *Proceedings of the 19th Symposium on Operating Systems Principles (poster session)*, October 2003.

[141] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. *Proceedings of the 19th Annual Computer Security Applications Conference*, December 2003.

[142] N. Provos. Improving Host Security with System Call Policies. *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[143] M. Dornseif, T. Holz, and C. Klein. NoSEBrEaK - Attacking Honeynets. *Proccedings of the 5th Annual IEEE Information Assurance Workshop*, June 2004.

[144] L. McVoy and C. Staelin. LMbench: Portable Tools for Performance Analysis. *Proccedings of the 1996 USENIX Annual Technical Conference*, January 1996.

[145] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. *Proceedings of the 9th ACM Conference on Computer and Communication Security*, November 2002.

[146] J. Butler. Direct Kernel Object Manipulation (DKOM). `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-butler.pdf`, 2004.

[147] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A Coprocessor-Based Kernel Runtime Integrity Monitor. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[148] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. *Proceedings of the 20th Symposium on Operating Systems Principles*, October 2005.

[149] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. *ACM Computing Survey*, 34(3), 2002.

[150] A. Sistla and J. Welch. Efficient Distributed Recovery Using Message Logging. *Proceedings of the 1989 ACM Symposium on Principles of Distributed Computing*, 1989.

[151] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 3(3), 1985.

[152] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[153] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching Intrusion Alerts Through Multi-Host Causality. *Proceedings of the 12th Network and Distributed System Security Symposium*, February 2005.

[154] S. T. King, George W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[155] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design & Implementation*, December 2004.

[156] A. Whitaker, R. S. Cox, and S. D. Gribble. Using Time Travel to Diagnose Computer Problems. *Proceedings of the 11th SIGOPS European Workshop*, September 2004.

[157] A. Stavrou, A. D. Keromytis, J. Nieh, V. Misra, and D. Rubenstein. MOVE: An End-to-End Solution To Network Denial of Service. *Proceedings of 12th Symposium on Network and Distributed System Security*, February 2005.

[158] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[159] Agobot Backdoor. *http://www.viruslist.com/en/viruses/encyclopedia?virusid=42101*.

[160] S. T. King, P. M. Chen, Y. M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. *Proceedings of the 27th IEEE Symposium on Security and Privacy*, May 2006.

[161] 'Blue Pill' Prototype Creates 100% Undetectable Malware. *http://www.eweek.com/article2/0,1895,1983037,00.asp*.

[162] PaX Team: PaX address space layout randomization (ASLR). *http://pax.grsecurity.net/docs/aslr.txt*.

[163] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *Proceedings of the 10th ACM Conference on Computer and Communication Security*, October 2003.

[164] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. *Proceedings of the 10th ACM Conference on Computer and Communication Security*, October 2003.

VITA

VITA

Xuxian Jiang was born in Yi Wu, China in 1976. He graduated from Yi Wu High School in the Spring of 1994 and then attended Xi'an Jiaotong University, where he received his B.S. in Computer Science in July 1998 and his M.S. in Computer Science in March 2001. He then enrolled in the graduate program at Purdue University where he completed his Ph.D. in Computer Science in August 2006. In the Fall of 2006, he will join the faculty of the Department of Information and Software Engineering at George Mason University, Fairfax, Virginia as an assistant professor.