Automating Malware Detection by Inferring Intent

Weidong Cui

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-115
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-115.html

September 14, 2006

Copyright © 2006, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Automating Malware Detection by Inferring Intent

by

Weidong Cui

B.E. (Tsinghua University) 1998
M.E. (Tsinghua University) 2000
M.S. (University of California, Berkeley) 2003

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Dr. Vern Paxson
Professor David A. Wagner
Professor John Chuang

Fall 2006

Automating Malware Detection by Inferring Intent

Copyright © 2006

by

Weidong Cui

Abstract

Automating Malware Detection by Inferring Intent

by

Weidong Cui

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Randy H. Katz, Chair

An increasing variety of malware like worms, spyware and adware threatens both personal and business computing. Modern malware has two features: (1) malware evolves rapidly; (2) self-propagating malware can spread very fast. These features lead to a strong need for *automatic* actions against new unknown malware. In this thesis, we aim to develop new techniques and systems to automate the detection of new unknown malware because detection is the first step for any reaction. Since there is no single panacea that could be used to detect all malware in every environment, we focus on one important environment, *personal computers*, and one important type of malware, *computer worms*.

To tackle the problem of automatic malware detection, we face two fundamental challenges: *false alarms* and *scalability*. We take two new approaches to solve these challenges. To minimize false alarms, our approach is to *infer the intent* of user or adversary (the malware author) because most benign software running on personal computers is user driven, and authors behind different kinds of malware have distinct intent. To achieve early detection of fast spreading Internet worms, we must monitor the Internet from a large number of vantage points, which leads to the scalability problem—how to filter repeated probes. Our approach is to leverage *protocol-independent replay* of application dialog, a new technology which, given examples of an application session, can mimic both the initiator and responder sides of the session for a wide variety of application protocols

without requiring any specifics about the particular application it mimics. We use replay to filter frequent multi-stage attacks by replaying the server side responses.

To evaluate the effectiveness of our new approaches, we develop the following systems: (1) *BINDER*, a host-based detection system that can detect a wide class of malware on personal computers by identifying extrusions, malicious outbound network requests which the user did not intend; (2) *GQ*, a large-scale, high-fidelity honeyfarm system that can capture Internet worms by analyzing in real-time the scanning probes seen on a quarter million Internet addresses, with emphases on isolation, stringent control, and wide coverage.

Professor Randy H. Katz
Dissertation Committee Chair

To my parents and Li.

Contents

List of Figures

List of Tables

Acknowledgements

When I am about to wrap up my work and life in Berkeley, I look back at these years and realize that I was really fortunate to receive help and support from many people. Without them, I wouldn't be able to finish my Ph.D. dissertation.

First and foremost, I would like to thank my advisor, Professor Randy Katz, for giving me a chance to work with him and guiding me through these years. Randy's advice and support was critical for my Ph.D. study. His technical advice teaches me how to conduct cutting-edge research; his patience and trust gives me confidence to overcome the hurdles in my research; his unrestricted support leads me to my choice on network security research; his care and responsibility to students tells me what's beyond being a researcher.

I would like to thank my de facto co-advisor, Dr. Vern Paxson, for taking me on to work with him in the CCIED project. I have learned tremendously from him: his high standard for research, exceptional diligence, efficient and effective communication, curiosity for new problems, and understanding and consideration for students. My work experience with Randy and Vern would always inspire me to do my best in my career.

I would also like to thank Professors David Wagner and John Chuang for serving on my qualifying exam and dissertation committee. Their early feedback to my thesis proposal was very helpful for me to formulate my research plan. I am also very grateful to them for reading and commenting on my thesis.

I was also very fortunate to work with Professor Ion Stoica, Dr. Wai-tian (Dan) Tan, and Dr. Nicholas Weaver. Ion helped me start my first research project at Berkeley and has always been available to offer his advice on both my technical and career questions. I did my industrial internship under the mentoring of Dan at HP Labs, who led me to enjoy the fun of system hacking and has been a great mentor and friend. Nick has generously helped me with my quals and research at ICSI. I want to thank all of them for their advice and help. I would also like to thank Professor Anthony Joseph for his technical and career advice.

I would like to thank my colleagues in CCIED for their insightful comments and intriguing discussion: Professors Stefan Savage, Geoffrey Voelker, and Alex Snoeren, Dr. Mark Allman,

Dr. Robin Sommer, Martin Casado, Jay Chen, Christian Kreibich, Justin Ma, Michael Vrable, and Vinod Yegneswaran.

My colleagues in Berkeley have always been there to give me their help. Helen Wang generously shared with me her Ph.D. research experience in my first year. Adam Costello's feedback intrigued me to rethink my backup-path allocation algorithm in my first networking research paper. Sridhar Machiraju and Matthew Caesar were fantastic partners in our collaboration. I want to thank all of them for their help. I would also like to thank others who have served to comment on my ideas and review my paper drafts: Sharad Agarwal, Mike Chen, Yan Chen, Chen-Nee Chuah, Yanlei Diao, Yitao Duan, Rodrigo Fonseca, Ling Huang, Jayanthkumar Kannan, Chris Karlof, Karthik Lakshminarayanan, Yaping Li, Morley Mao, Ana Sanz Merino, David Oppenheimer, George Porter, Bhaskaran Raman, Ananth Rao, Mukund Seshadri, Jimmy Shih, Lakshminarayanan Subramanian, Mel Tsai, Kai Wei, Alec Woo, Fang Yu, and Shelley Zhuang.

My life in Berkeley would have been much less fun without my Chinese friends. Zhanfeng Jia has given me tremendous help since my first day in Berkeley. Fei Lin gave me her generous help in my first year. Wei Liu shared with me his love for sports and beer through his years in Berkeley. Jinwen Xiao has always been available to offer her advice and help whenever I asked. I want to thank all of them for their help. I am also very grateful to Minghua Chen, Yanmei Li, Jinhui Pan, Na Pi, Xiaofeng Ren, Chen Shen, Xiaotian Sun, Wei Wei, Rui Xu, Guang Yang, Jing Yang, Lei Yuan, Min Yue, Jianhui Zhang, Haibo Zeng, Wei Zheng, and Xu Zou, for all the joy they have brought to me in these years.

Finally, I am extremely grateful to my parents and wife. Without their selfless love and support, I wouldn't be able to reach this point in my life. My father and mother taught me the importance of being honest, optimistic, confident, and hard-working. They not only gave me a life but also led me to the path to explore and enjoy all the adventures in life. My wife, Li Yin, is my best colleague and friend. Her love, care, support, and encouragement have been and will be the source of my strength.

Chapter 1

Introduction

1.1 Motivation

An increasing variety of malware like worms, spyware and adware threatens both personal and business computing [32, 86]. Since 2001, large-scale Internet worm outbreaks have not only compromised hundreds of thousands of computer systems [56, 54, 97] but also slowed down many parts of the Internet [54]. The Code Red worm [56] exploited a single Microsoft IIS server vulnerability, took 14 hours to infect the 360,000 vulnerable hosts, and launched a flooding attack against the White House web server. Using only a single UDP packet for infection, Slammer [54] took about 10 minutes to infect its vulnerable population and caused severe congestion in both local and global networks. Spyware [73] jeopardizes computer users by disclosing their personal information to third parties. Remotely controlled "bot" (short for robot, an automated software program that can execute certain commands when it receives a specific input) networks of compromised systems are growing quickly [13]. The bots have been used for spam forwarding, distributed denial of service attacks, and spyware. In short, modern threats are causing severe damage to today's computer world.

Before we describe the problem this thesis tackles, we first look at the features of modern malware.

- **Malware evolves rapidly.** In a recent study [86], Symantec reported that it detected more than 21,000 new Win32 malware variants in 2005, which is one order of magnitude larger

than the number of variants seen in 2003. In addition to the volume increase, we also see more attacks against personal computers that exploit not only buffer overflow vulnerabilities but also user misbehavior in using web browser, instant messaging, and peer-to-peer software.

- **Self-propagating malware can spread very rapidly.** While it took 14 hours for the Code Red worm to infect its vulnerable hosts, it took only 10 minutes for the Slammer worm to do so. Staniford *et al.* presented several techniques to accelerate a worm's spread in [83, 82]. Their simulations showed that flash worms, which propagate over a pre-constructed tree for efficient spreading, can complete their propagation in less than a second for single packet UDP worms and only a few seconds for small TCP worms. In another study [57], Moore *et al.* demonstrated that effective worm containment requires a reaction time under 60 seconds.

These features lead to a strong need for *fast*, *automatic* action against new unknown malware. In the next section (Section 1.2), we describe the research problems we attempt to tackle in this thesis and discuss the research challenges we face. In Section 1.3, we present our contributions and the structure of this thesis.

1.2 Problem Definition and Challenges

To fight against new unknown malware, there are two potential approaches: *proactive* and *reactive*. Proactive approaches focus on eliminating vulnerabilities to prevent any malware. Reactive approaches concern about actions after malware is unleashed. We believe that there is a long way to go before we could achieve zero attacks. Thus we follow reactive approaches in this thesis. To react to a new malicious attack, the *first* step is to *detect* the new malware. Since there are more and more malware variants and self-propagating malware can spread very rapidly, we need *fast, automatic* detection. In this thesis, we aim to *develop new techniques and systems to automate the detection of new unknown malware*.

To tackle this problem, we face two fundamental challenges.

- **False Alarms**: To detect new unknown malware means we cannot use signatures. Instead, we must detect it by recognizing certain patterns possessed solely by some malware. In this

thesis, we focus on patterns one can infer by monitoring its anomalous behavior. Anomaly-based intrusion detection techniques have been studied for many years [4, 18], but they are not widely deployed in practice due to their high false alarms. Axelsson [5] used the base-rate fallacy phenomenon to show that, when intrusions are rare, the false alarm rate (i.e., the probability that a benign event causes an alarm) has to be very low to achieve an acceptable Bayesian detection rate (i.e., the probability that an alarm implies an intrusion). In automatic detection and reaction, we tolerate fewer false alarms because actions on false alarms would interrupt normal communication and computation.

To minimize false alarms, the approach in this thesis is to *infer the intent* of user or adversary (the malware author).

- *User Intent*: It is a unique characteristic on personal computers. Most benign software running on personal computers shares a common feature, that is, its activity is user driven. Thus, if we can infer user intent, we can detect break-ins of new unknown malware by identifying activity that the user did not intend. In other words, instead of attempting to model anomalous behavior, we model the normal behavior of benign software: their activity follows user intent. Since users of personal computers usually use input devices such as keyboards and mice to control their computers, we assume that user intent can be inferred from user-driven activity such as key strokes and mouse clicks. Software activity may include network requests, file access, or system calls. We focus on network activity in this thesis. The idea is that an outbound network connection that is not requested by a user or cannot be traced back to some user request is treated as likely to be malicious. A large class of malware makes such malicious outbound network connections either for self-propagation (worms) or to disclose user information (spyware/adware). We refer to these malicious, user-unintended, outbound network connections as *extrusions*. Therefore, we can detect new unknown malware on personal computers by identifying extrusions. In Chapter 3, we present the details of the extrusion detection algorithm and a prototype system we built.
- *Adversary Intent*: Authors of malware have unique intent. For example, worm authors intend to spread their malicious code to vulnerable systems via self-propagation; spy-

ware authors intend to collect private information on victim machines; bot authors intend to control their bots and execute various malicious tasks on them. In our work, we focus on computer worms. Given the intent of worm authors, we detect new worms by identifying *self-propagation*. Our basic approach is to first let a machine become infected, and then let the first machine infect another one, and so on. By doing so, we can observe a chain of self-propagation if the first machine is compromised by a computer worm. We have never seen a false alarm by using this approach (see Section 5.4.3).

- **Scalability**: We must achieve early detection of fast spreading Internet worms for any effective defense. To do so, we must monitor the Internet from a large number of vantage points. This leads to the scalability problem—how to analyze the large amount of data efficiently in real-time.

  A central tool that has emerged recently for detecting Internet worm outbreaks is the *honeyfarm* [80], a large collection of honeypots fed Internet traffic by a *network telescope* [6, 34, 58, 96]. In our work, we use high-fidelity honeypots to analyze in real-time the thousands of scanning probes per minute seen on the large address space of our network telescope. To do so by using a small number of honeypots, we leverage extensive filtering and engage honeypots dynamically. In extensive filtering, we *filter* scanning probes in multiple stages: (1) the scan filter limits the total number of distinct telescope addresses that engage a remote source; (2) the first-packet filter drops a connection if the first data packet unambiguously identifies the attack as a known threat; (3) the replay proxy filters frequent multi-stage attacks by replaying the server side responses until the point where it can identify the attack.

  The replay proxy leverages a new technology—protocol-independent replay of application dialog—we develop in this thesis. In replay, we use a *script* automatically derived from one or two examples of a previous application session as the basis for mimicking either the client or the server side of the session in a subsequent instance of the same transaction. The idea is that we can represent each previously seen attack using a script that describes the network-level dialog (both client and server messages) corresponding to the attack. A key property of such scripts is that we can automatically extract them from samples of two dialogs corresponding to a given attack, *without* any knowledge of the semantics of the application protocol used in

the attack. In Chapter 4, we present the details of protocol-independent replay. In Chapter 5, we describe the design and implementation of our honeyfarm system.

In next section, we highlight our contributions and describe the structure of this thesis.

1.3 Contributions and Structure of This Thesis

In this thesis, we tackle the problem of automating detection of new unknown malware. Because there are many different kinds of malware and they infect computer systems in many different environments, there is no single panacea that can detect all malware in every environment. We recognize this and focus on one important environment (personal computers) and one important type of malware (computer worms). We develop the following algorithms and systems:

- **Extrusion Detection**: Our algorithm is based on the idea of inferring user intent. It detects new unknown malware on personal computers by identifying extrusions, malicious outbound network requests which the user did not intend. We implement BINDER, a host-based detection system for Windows that realizes this algorithm and can detect a wide class of malware on personal computers, including worms, spyware, and adware, with few false alarms.

- **Protocol-Independent Replay**: We develop RolePlayer, a system which, given examples of an application session, can mimic both the initiator and responder sides of the session for a wide variety of application protocols. A key property of RolePlayer is that it operates in an application-independent fashion: the system does not require any specifics about the particular application it mimics. We can potentially use such replay for recognizing malware variants, determining the range of system versions vulnerable to a given attack, testing defense mechanisms, and filtering multi-stage attacks.

- **GQ Honeyfarm System**: We develop GQ, a large-scale honeyfarm system that ensures high-fidelity honeypot operation, efficiently discards the incessant Internet "background radiation" that has only nuisance value when looking for new forms of activity, and devises and enforces an effective "containment" policy to ensure that the detected malware does not inflict external

damage or skew internal analyses. GQ leverages aggressive filtering, including a technique based on protocol-independent replay.

The rest of this thesis is organized as follows. In Chapter 2, we discuss the related work in the areas of intrusion detection and Internet epidemiology and defense. It provides the background from which this thesis is developed.

In Chapter 3, we describe the design of the extrusion detection algorithm and the implementation and evaluation of the BINDER prototype system. We present the results of both real-world and control-testbed experiments to demonstrate how effectively the system detects a wide class of malware and controls false alarms. Our limited user study indicates that BINDER controls the number of false alarms to at most five over four weeks on each computer and the false positive rate is less than 0.03%. Our control-testbed experiments show that BINDER can successfully detect the Blaster worm and 22 different email worms (collected on a departmental email server over one week).

In Chapter 4, we describe the design and implementation of the RolePlayer system that can mimic both the client and server sides of an application session for a wide variety of application protocols without knowing any specifics of the application it mimics. We present the evaluation of RolePlayer with a variety of network applications as well as the multi-stage infection processes of some Windows worms. Our evaluations show that RolePlayer successfully replays the client and server sides for NFS, FTP, and CIFS/SMB file transfers as well as the infection processes of the Blaster and W32.Randex.D worms.

In Chapter 5, we describe the design and implementation of the GQ honeyfarm system which we built to detect new worm outbreaks by analyzing in real-time the scanning probes seen on a quarter million Internet addresses. We report on preliminary experiences with operating the system at scale. We monitor more than a quarter million Internet addresses and have detected 66 distinct worms in four months of operations, which far exceeds that previously achieved for a high-fidelity honeyfarm.

Finally, in Chapter 6, we summarize our work and contributions, and discuss directions for future work.

Chapter 2

Related Work

In this chapter, we discuss the related work in the areas of intrusion detection and Internet epidemiology and defense. Instead of covering every piece of work in these areas, we focus on the most relevant work and aim to provide the background from which we developed this thesis. We start with intrusion detection (see Section 2.1), describing techniques proposed in the areas of host-based intrusion detection (see Section 2.1.1) and network-based intrusion detection (see Section 2.1.2) Then we describe the research efforts made in the area of Internet epidemiology and defense (see Section 2.2), which motivate our work on developing a large-scale, high-fidelity honeyfarm system.

2.1 Intrusion Detection

Research on intrusion detection has a long history since Anderson's [4] and Denning's [18] seminal work. Prior work on intrusion detection can be roughly classified along two dimensions: detection method (misuse-based vs. anomaly-based) and audit data source (host-based vs. network-based). Misuse-based (also known as signature-based or rule-based) intrusion detection systems use a set of rules or signatures to check if intrusion happens, while anomaly-based intrusion detection systems attempt to detect anomalous behavior by observing a deviation from the previously learned normal behavior of the system or the users. Host-based intrusion detection systems leverage audit

data collected from end hosts, while network-based intrusion detection systems monitor and analyze network traffic.

2.1.1 Host-based Intrusion Detection

Early work on anomaly-based host intrusion detection was focused on analyzing system calls. In [23], Forrest *et al.* modeled processes as normal patterns of short sequences of system calls and used the percentage of mismatches to detect anomalies. Their work was based on two assumptions: (1) the sequence of system calls executed by a program is locally consistent during normal operation, and (2) some unusual short sequence of system calls will be executed when a security hole in a program is exploited. Ko *et al.* [38] proposed to use a specification language to define the intended behavior of every program. The adoption of their scheme is limited because of the low degree of automation. To detect intrusions, Wagner and Dean [100] used static analysis to learn "program intent" and then compared the runtime behavior of a program with its intent. They proposed three models to describe program intent: call graph, stack and digraph. The advantages of their techniques are: a high degree of automation; protection against a large class of attacks; the elimination of false alarms. The limitation is that program source code is required. Sekar *et al.* [75] proposed to train a finite-state automaton using a trace of system calls, program counter and stack information and monitor system calls to check if they follow the automaton in real-time. Compared with Wagner and Dean's work, this method does not require program source code but may have false alarms due to the incompleteness of system call traces. Liao and Vemuri [48] applied text categorization techniques to intrusion detection by making an analogy between processes and documents and between system calls and words. They used a k-Nearest Neighbor classification algorithm to classify processes based on system calls. In [101], Wagner and Soto showed that it is possible for attackers to evade host-based anomaly intrusion detection systems that use system calls. Besides system calls, the behavior of storage [65] and file [113] systems has also been used for intrusion detection. Pennington *et al.* [65] proposed to monitor storage activity to detect intrusions using a set of predetermined rules. Their approach has limited ability to detect new attacks due to the usage of rules. In [113], Xie *et al.* proposed to detect intrusions by identifying "simultaneous" creations of

new files on multiple machines in a homogeneous cluster. They correlated the file system activity on a coarse granularity of one day, which slows the detection.

Anomaly-based host intrusion detection techniques are not widely deployed in practice. A key obstacle is their high false alarm rate. Axelsson [5] used the base-rate fallacy phenomenon to show that, when intrusions are rare, the false alarm rate has to be very low to achieve an acceptable Bayesian detection rate, i.e., the probability that alarms imply intrusions. In contrast, signature-based host intrusion detection systems are well adopted in practice. Commercial products like Symantec [87] and ZoneAlarm [120] protect hosts against intrusions by filtering known attacks, but they cannot detect new ones.

Recently, dynamic data flow analysis [62, 15] has been used to detect attacks that overwrite a buffer with data received from untrusted sources (e.g., buffer overflow attacks). These approaches have very good detection accuracy, but high performance overhead limits their deployment.

In summary, previous host-based intrusion detection systems have the following limitations. Signature-based solutions have low false alarm rates but do not perform well on detecting new attacks. Anomaly-based solutions can detect new attacks but have high false alarm rates. Moreover, anomaly-based solutions that use system calls are likely to fail to detect new kinds of attacks like spyware and email viruses because these attacks usually run as a new program and its program behavior in terms of system calls may be similar to other benign programs.

2.1.2 Network-based Intrusion Detection

The first network intrusion detection system (NIDS) was developed by Snapp *et al.* in [78]. Lee and Stolfo [44, 45] proposed a framework for constructing features and models for intrusion detection. Their key idea is to use data mining techniques to compute frequent patterns, select features, and then apply classification algorithms to learn detection models. To detect network intrusions, Sekar *et al.* [76] proposed a specification language to combine traffic statistics with state-machine specifications of network protocols. In [92], Thottan and Ji used a statistical signal processing technique based on abrupt change detection to detect network anomalies. Snort [71] is a signature-based lightweight network intrusion detection system. Paxson [64] developed Bro, a system for detecting

network intrusions in real-time. Bro applies the design philosophy of separating mechanisms from policies by separating the event engine and the policy script interpreter. The former is for processing packets at the network/application level, while the latter is for checking events according to policies and generating real-time alarms. Ptacek and Newsham [68] presented three classes of attacks against network intrusion detection systems: insertion (attackers send packets that are only seen by the NIDS but not the end system), evasion (attackers send packets that create ambiguities for the NIDS), and denial of service attacks. Handley *et al.* [26] studied the evasion problem and introduced a traffic *normalizer* that patches up the packet stream to eliminate potential ambiguities. To detect new unknown attacks exploiting known vulnerabilities, Wang *et al.* [102] developed Shield, which prevents vulnerability-specific, exploit-generic attacks by using network filters built based on known vulnerabilities.

In summary, anomaly-based network intrusion detection suffers from false alarms while signature-based solutions do not reliably detect new unknown attacks.

2.2 Internet Epidemiology and Defenses

2.2.1 Overview

Since 2001, Internet worm outbreaks have caused severe damage that affected tens of millions of individuals and hundreds of thousands of organizations. The Code Red worm of 2001 [56] was the first *outbreak* in today's global Internet after the Morris worm [20, 79] in 1988. It not only compromised 360,000 hosts with a Web server vulnerability but also attempted to launch a distributed denial of service attack against a government web server from those hosts. The Blaster worm [97] exploited a vulnerability of a service running on millions of personal computers. The Slammer worm [54] used only a single UDP packet for infection and took just 10 minutes to infect its vulnerable population. The excessive scan traffic generated by Slammer slowed down many parts of the Internet. Also using a single UDP packet for infection, the Witty worm [55] exploited a vulnerability, which was publicized only one day before, in a commercial intrusion detection software. It was the first to carry a destructive payload that overwrote random disk blocks. It

also started in an organized manner with an order of magnitude more ground-zero hosts than any previous worm, and apparently was targeted at military bases. Weaver *et al.* provided a taxonomy of computer worms in [104].

Staniford *et al.* presented several techniques to accelerate a worm's spread in [83, 82]. They described the design of flash worms, in which the worm author collects a list of vulnerable hosts, constructs an efficient spread tree and encode it in the worm before propagating the worm. Their simulations showed that flash worms can complete their spread in less than a second for single packet UDP worms and only a few seconds for small TCP worms. In another study [57], Moore *et al.* demonstrated that effective worm containment requires a reaction time under 60 seconds. These results suggest that *automated early* detection of new worm outbreaks is essential for any effective defense.

2.2.2 Network Telescopes

Understanding the behavior of Internet-scale worms requires broad visibility into their workings. *Network telescopes*, which work by monitoring traffic sent to unallocated portions of the IP address space, have emerged as a powerful tool for this purpose [58], enabling analysis of remote denial-of-service flooding attacks [59], botnet probing [116], and worm outbreaks [42, 54, 55, 56].

However, the passive nature of network telescopes limits the richness of analysis we can perform with them; often, all we can tell of a source is that it is probing for a particular type of server, but not what it will do if it finds such a server. We can gain much richer information by interacting with the sources. One line of research in this regard has been the use of lightweight mechanisms that establish connections with remote sources but process the data received from them syntactically rather than with full semantics. For example, *iSink* [117] uses a stateless active responder to generate response packets to incoming traffic, enabling it to discriminate between different types of attacks by checking the response payloads. The lightweight responder of the *Internet Motion Sensor* [6] bases its analysis on payload signatures. The responder acknowledges TCP SYN packets to elicit the first data segment the source will send, using a cache of packet payload checksums to detect matches to previously seen activity. This matching, however, lacks sufficient power to identify

new probing that deviates from previously seen activity only in subsequent data packets, nor can it detect activity that is semantically equivalent to previous activity but differs in checksum due to the presence of message fields that do not affect the semantics of the probe (e.g., IP addresses embedded in data). Pang *et al.* used a detailed application-level responder to study the characteristics of Internet "background radiation" [63]. The work shows that a large proportion of such probing cannot in fact be distinguished using lightweight first-packet techniques such as those in [6, 117] due to the prevalence of protocols (such as Windows NetBIOS and CIFS) that engage in extensive setup exchanges.

In summary, network telescopes provide early and broad visibility into a worm's spread, but have no or limited capability of actively responding to probes. This makes them prone to false alarms when they are attempted for fast *detection* of novel worms.

2.2.3 Honeypots

A *honeypot* is a vulnerable network decoy whose value lies in being probed, attacked, or compromised for the purposes of detecting the presence, techniques, and motivations of an attacker [81]. Honeypots can run directly on a host machine ("bare metal") or within virtual machine environments that provide isolation and control over the execution of processes within the honeypot. Honeypots can also be classified as low-interaction, medium-interaction or high-interaction (alternatively, low-fidelity or high-fidelity). Low-interaction honeypots can emulate a variety of services that the attackers can interact with. Medium-interaction honeypots provide more functionalities than low-interaction honeypots but still do not have a full, productive operating system. One example is the use of `chroot` in a UNIX environment. High-interaction honeypots give the attackers access to a real operating system with few restrictions.

In [66], Provos proposed Honeyd, a scalable virtual honeypot framework. Honeyd can personalize TCP response to deceive Nmap [51] (a free open source utility for network exploration or security auditing, which can be used to determine what operating systems and OS versions a host is running) and create arbitrary virtual routing topologies. Honeyd can be used for detecting worms and distracting adversaries. However, low-interaction honeypots' ability to detect novel worms is

limited to their emulation capability in terms of the number of services they can emulate and their incapability of emulating unknown vulnerabilities. High-interaction honeypots behave exactly as a real host and therefore allow accurate, *in situ* examinations of a worm's behavior. Moreover, they can be used to detect new attacks against even unknown vulnerabilities in real operating systems [29]. However, they also require more protection in containment and isolation.

Advances in Virtual Machine Monitors such as VMware [95], Xen [7], and User-Mode Linux (UML) [19] have made it tractable to deploy and manage high-interaction virtual honeypots. The Honeynet project [29] proposed an architecture for high-interaction honeypots to contain and analyze attackers in the wild, focusing on two main components: data control and data capture. The data control component prevents attackers from using the Honeynet to attack or harm other systems. It limits outbound connections based on configurable policies and uses *Snort Inline* [53] to block known attacks. The data capture component (also known as Sebek [30]) logs all of the attacker's activity and transmits data to a safe machine without the attacker knowing it. In [17], Dagon and colleagues presented HoneyStat, a system that uses high-interaction honeypots running on VMware GSX servers to monitor memory, disk and network activity on the guest OS. HoneyStat uses logit regression [106] for causality analysis to detect a worm when discovering that a single network event causes all subsequent memory or disk events.

By themselves, honeypots serve as poor "early warning" detection of new worm outbreaks because of the low probability that any particular honeypot will be infected early in a worm's growth. Conceptually, one might scatter a large number of well-monitored honeypots across the Internet to address this limitation, but this raises major issues of management, cost, and liability. Spitzner presented the idea of "Honeypot Farms" to address these issues [80]. These work by deploying a collection of honeypots in a single, centralized location, where they receive suspicious traffic redirected from distributed network locations. Jiang and Xu presented a VM-based honeyfarm system, *Collapsar*, which uses UML to capture and forward packets via Generic Routing Encapsulation (GRE) tunnels and *Snort-Inline* to contain outbound traffic. However, since in Collapsar each honeypot has a single IP address, the probability of early detection is still low due to the limited "cross section" presented by the honeypots.

In summary, we still lack solutions that can automatically detect and analyze Internet worm

outbreaks in seconds. In this thesis, we develop a large-scale high-fidelity honeyfarm system to tackle this challenge.

2.2.4 Defenses

There have been many research efforts on detecting scanning worms. When a host is compromised by a scanning worm, it will try to spread by scanning other hosts while a large fraction of them may not exist. Williamson [107] presented a *virus throttle*, a rate-limiting solution that throttles the rate of new connections made by a compromised host. The virus throttle can detect and contain *fast* scanning worms. Zou *et al.* [121] proposed a new architecture for detecting scanning worms. Their basic idea is to detect the trend, not the rate of monitored illegitimate scan traffic because at the beginning of its spreading, a scanning worm propagates almost exponentially with a constant, positive rate [122]. They used a Kalman filter to compute the parameter modeling the victim increasing rate. If that parameter stabilizes and oscillates slightly around a positive constant value, it means a new worm outbreak. Based on the theory of sequential hypothesis testing, Jung *et al.* [35] proposed an on-line port scan detection algorithm called Threshold Random Walk (TRW). In practice, TRW requires only four or five connection attempts to detect a malicious remote hosts. In [74], Schechter *et al.* proposed a hybrid approach that integrates reverse sequential hypothesis testing and credit-based connection rate limiting. Weaver *et al.* [105] proposed a solution based on TRW to detect and contain scanning worms in an enterprise environment at very high speed by leveraging hardware acceleration. They showed that their techniques can stop a scanning host after fewer than ten scans with a very low false positive rate.

Email worms like SoBig and MyDoom have caused severe damage [112]. Several solutions have been proposed to defend against email worms. Gupta and Sekar [25] proposed an anomaly-based approach that detects increases in traffic volume over what was observed during the training period. Their assumption was that an email virus attack would overwhelm mail servers and clients with a large volume of email traffic. In [31], Hu and Mok proposed a framework based on behavior skewing and cordoning. Particularly, they set up bait email addresses (behavior skewing), intercept SMTP messages to them (behavior monitoring), and forward suspicious emails to their own SMTP server (cordoning). Based on conventional epidemiology, Xiong [114] proposed an at-

tachment chain tracing scheme that detects email worm propagation by identifying the existence of transmission chains in the network.

It is important to generate attack signatures of previously unknown worms so that signature-based intrusion detection systems can use the signatures to protect against these worms. Several automated solutions have been proposed in the past. Kreibich and Crowcroft [41] proposed Honeycomb, which uses Honeyd as the frontend. After merging connections, Honeycomb compares them horizontally (the nth incoming messages of all connections) and vertically (all incoming messages of two connections) and uses a Longest Common Substring algorithm to find attack signatures. In [77], Singh *et al.* developed Earlybird, a worm fingerprinting system that uses *content sifting* to generate worm signatures. Content sifting is based on two anomalies of worm traffic: (1) some content in a worm's exploit is invariant; (2) the invariant content appear in flows from many source to many destinations. By using multi-resolution bitmaps and multistage filters, Earlybird can maintain the necessary data structures at very high speed and with low memory overhead. Kim and Karp [36] proposed Autograph, which automatically generates worm signatures by analyzing the prevalence of portions of flow payloads. Autograph uses content-based payload partitioning to divide payloads of suspicious flows from scanning sources into variable-length content blocks, then applies a greedy algorithm to find prevalent content blocks as signatures. All these solutions do not work with polymorphic worms whose content may be encoded into successive, different byte strings. In [61], Newsome *et al.* proposed Polygraph, which automatically generates signatures for polymorphic worms. Polygraph leverages a key assumption that multiple invariant substrings must often be present in all variants of a worm payload for the worm to function properly. These substrings typically correspond to protocol framing, return addresses, and poorly obfuscated code.

Some worm behavior signatures [21] can be used to detect unknown worms. Network traffic patterns of worm behaviors include (1) sending similar data from one machine to the next, (2) tree-like propagation and reconnaissance, and (3) changing a server into a client. However, it is not clear how the behavioral approach can be deployed in practice because it is difficult to collect necessary information.

2.3 Summary

In this chapter, we reviewed the related work in the areas of intrusion detection and Internet epidemiology and defense. We observe that anomaly detection techniques are not widely deployed in practice due to their high false alarms. We also see that, in the war against Internet worms, it is critical to achieve fast automatic detection of new outbreaks. The previous work described in this chapter sets the stage for our work. In the next chapter, we present our work on detecting new unknown malware on personal computers by capturing user-unintended malicious outbound network connections.

Chapter 3

Extrusion-based Malware Detection for Personal Computers

The main goal of our work is to develop new techniques and system architectures to automate the process of malware detection. In this chapter, we describe how we can infer user intent to automatically detect a wide class of new unknown malware on personal computers without any prior knowledge. In Chapter 4 and Chapter 5, we will describe how we can infer adversary intent to detect fast spreading Internet worms automatically and quickly.

3.1 Motivation

Many research efforts [64, 71, 102] and commercial products [87, 120] have focused on preventing break-ins by filtering either known malware or unknown malware exploiting known vulnerabilities. To protect computer systems from rapidly evolving malware, these solutions have two requirements. First, a central entity must rapidly generate signatures of new malware after it is detected. Second, distributed computer systems must download and apply these signatures to their local databases before they are attacked. However, these can leave computer systems temporarily unprotected from newly emerging malware. [86] estimated that 49 days elapsed on average between the publication of a vulnerability and the release of an associated patch in 2005. On the other hand,

the average time for exploit development was less than 7 days in 2005. In particular, worms can propagate much more rapidly than humans can respond in terms of generation and distribution of signatures [83]. Therefore, we need schemes to detect new unknown malware when the signatures are *not* available.

Anomaly-based intrusion detection have been studied for detecting new unknown malware. [28] used short sequences of system calls executed by running processes to detect anomalies caused by intrusions. [45] proposed a data mining framework for constructing features and training models of network traffic for intrusion detection. [119] proposed a scheme based on the distinctive timing characteristics of interactive traffic for detecting compromised stepping stones. Recent work of [47] correlated simultaneous anomalous behaviors from different perspectives to improve the accuracy of intrusion detection. The performance of anomaly-based approaches is limited in practice due to their high false positive rate [5].

In short, it is highly desirable to develop new anomaly-based malware detection techniques that can achieve low false positive rate to be useful in practice. We tackle this problem on a specific but important platform—personal computers (i.e., a computer that is used locally by a single user at any time).

A unique characteristic of personal computers is *user intent*. Most benign software running on personal computers shares a common feature, that is, its activity is user driven. Thus, if we can infer user intent, we can detect break-ins of unknown malware by identifying activity which the user did not intend. Such activity can be network requests, file access, or system calls. In this thesis, we focus on network activity because a large class of malware makes malicious outbound network connections either for self-propagation (worms) or to disclose user information (spyware/adware). Such outbound connections are suspicious when they are uncorrelated with user activity. We refer to these malicious user unintended outbound connections as *extrusions*[1]. Thus we can detect new unknown malware on personal computers by identifying extrusions.

Prior to our work, user intent has been used to set the access control policy (also known as "designating authority") in a GUI (Graphic User Interface) system [85, 115, 84]. Based on the Principle of Least Authority (or Privilege), the basic idea of these research efforts is to designate

[1]Extrusion is also defined as unauthorized transfer of digital assets in some other context [9].

the necessary authority to a running program based on user actions it receives directly or indirectly. Unlike the prior work, our goal is to detect malware on personal computers that run a GUI, and our basic idea is that a running program that receives user input directly or indirectly may generate outbound network requests.

In the remainder of this chapter, we first describe the extrusion detection algorithm (see Section 3.2). Then, we present the architecture and implementation of BINDER (Break-IN DEtectoR), a host-based malware detection system that realizes the extrusion detection algorithm (see Section 3.3). Next, we demonstrate our evaluation methodology and show that BINDER can detect a wide class of malware with few false alarms (see Section 3.4). In the end, we discuss the limitations caused by potential attack techniques (see Section 3.5) and summarize this chapter (see Section 3.6).

3.2 Design

In this section, we first present the design goals and rationale for our approach. Then, we use an example to demonstrate how to infer user intent. Next, we describe the extrusion detection algorithm in detail. We conclude this section by discussing why outbound connections of a wide class of malware can be detected as extrusions.

3.2.1 Overview

Our goal is to automatically detect break-ins of new unknown malware on personal computers. Our design should achieve:

- **Minimal False Alarms**: This is critical for any automatic intrusion detection system.
- **Generality**: It should work for a large class of malware without the need for signatures, and regardless of how the malware infects the system.
- **Small Overhead**: It must *not* use intrusive probing or adversely affect the performance of the host computer.

We achieve these design goals by leveraging a unique characteristic on personal computers, that is, user intent. Our key observation is that outbound network connections from a compromised personal computer can be classified into three categories: *user-intended*, *user-unintended benign*, and *user-unintended malicious* (referred to as *extrusions*). Since many kinds of malware such as worms, spyware and adware make malicious outbound network connections either for self-propagation or to disclose user information, we can detect break-ins of these kinds of malware by identifying their extrusions. In the rest of this chapter, we use connections and outbound connections interchangeably unless otherwise specified. We do not consider inbound connections (initiated by a remote computer) because firewalls are designed to filter such traffic. In Chapter 5, we will describe how we use honeypots to inspect incoming malicious traffic to detect worms.

Since users of personal computers usually use input devices such as keyboards and mice to control their computers, we assume that *user intent* can be inferred from user-driven activities. We also assume malware runs as standalone processes. For those that hide in other processes by exploiting techniques proposed in [70], our current design is unable to detect them. We look at this as one of the limitations of today's operating systems. We will discuss in detail attack techniques and potential solutions in Section 3.5.

To detect extrusions, we need to determine if an outbound connection is user-intended. We infer user intent by correlating outbound network connections with user-driven input at the process level. Our key assumption is that outbound network connections made by a process that receives user input a short time ago is user-intended. We also treat repeated connections (from the same process to the same destination host or IP address within a given time window; see Section 3.2.3 for more details) as user-intended as long as the first one is user-intended. By doing this, we can handle the case of automatically refreshing web pages and polling emails. Among user-unintended outbound connections, we use a small whitelist to differentiate benign traffic from malicious traffic. The whitelist covers three kinds of programs: system daemons, applications automatically checking updates, and network applications automatically started by the operating system. Actual rules are specific to each operating system and may become user specific.

In the rest of this section, we first use an example to demonstrate how user intended connections may be initiated. Next, we describe the extrusion detection algorithm. Finally, we discuss how malware can be detected by this algorithm.

3.2.2 Inferring User Intent

The following example demonstrates how user-intended connections are initiated. Let us assume that a user opens an Internet Explorer (IE) window, goes to a news web site, then leaves the window idle to answer a phone call. We consider three kinds of events: user events (user input), process events (process start and process finish), and network events (connection request, data arrival and domain name lookup). A list of events generated by these user actions is shown in Figure 3.1. In this example, new connections are generated in the following four cases.

- Case I: When the user opens IE by double-clicking its icon on My Desktop in Windows, the shell process explorer.exe (PID=1664) of Windows receives the user input (Event 1-2), and then starts the IE process (Event 3). After the domain name (www.cs.berkeley.edu) of the default homepage is resolved (Event 4), the IE process makes a connection to it to download the homepage (Event 5). This connection of IE is triggered by the user input of its parent process of explorer.exe.

- Case II: After the user clicks a bookmark of news.yahoo.com in the IE window (Event 8), the domain name is resolved as 66.218.75.230 (Event 9). Then the IE process makes a connection to it to download the HTML file (Event 10). This connection is triggered by the user input of the same process.

- Case III: After receiving the HTML file in 4 packets (Event 11-14), IE goes to retrieve two image files from us.ard.yahoo.com and us.ent4.yimg.com. IE makes connections to them (Event 16,18) after the domain names are resolved (Event 15,17). These two connections are triggered by the HTML data arrivals of the same process.

- Case IV: According to a setting in the web page, IE starts refreshing the web page for updated

```
 1 09:32:33, PID=1664 (user input)
 2 09:32:33, PID=1664 (user input)
 3 09:32:35, PID=2573, PPID=1664, NAME="C:\...\iexplore.exe" (process start)
 4 09:32:39, HOST=www.cs.berkeley.edu, IP=169.229.60.105 (DNS lookup)
 5 09:32:39, PID=2573, LPORT=5354, RIP=169.229.60.105, RPORT=80 (connection request)
 6 09:32:40, PID=2573, LPORT=5354, RIP=169.229.60.105, RPORT=80 (data arrival)
 7 09:32:40, PID=2573, LPORT=5354, RIP=169.229.60.105, RPORT=80 (data arrival)
 8 09:32:43, PID=2573 (user input)
 9 09:32:45, HOST=news.yahoo.com, IP=66.218.75.230 (DNS lookup)
10 09:32:45, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (connection request)
11 09:32:47, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (data arrival)
12 09:32:47, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (data arrival)
13 09:32:48, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (data arrival)
14 09:32:48, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (data arrival)
15 09:32:50, HOST=us.ard.yahoo.com, IP=216.136.232.142,216.136.232.143 (DNS lookup)
16 09:32:50, PID=2573, LPORT=5359, RIP=216.136.232.142, RPORT=80 (connection request)
17 09:32:51, HOST=us.ent4.yimg.com, IP=192.35.210.205,192.35.210.199 (DNS lookup)
18 09:32:51, PID=2573, LPORT=5360, RIP=192.35.210.205, RPORT=80 (connection request)
19 09:32:52, PID=2573, LPORT=5359, RIP=216.136.232.142, RPORT=80 (data arrival)
20 09:32:52, PID=2573, LPORT=5360, RIP=192.35.210.205, RPORT=80 (data arrival)
21 09:32:53, PID=2573, LPORT=5359, RIP=216.136.232.142, RPORT=80 (data arrival)
22 09:32:53, PID=2573, LPORT=5360, RIP=192.35.210.205, RPORT=80 (data arrival)
23 09:43:01, HOST=news.yahoo.com, IP=66.218.75.230 (DNS lookup)
24 09:43:01, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (connection request)
25 09:43:02, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (data arrival)
26 09:43:02, PID=2573, LPORT=5357, RIP=66.218.75.230, RPORT=80 (data arrival)
   ......
```

Figure 3.1. An example of events generated by browsing a news web site.

news 10 minutes later. This connection (Event 24) repeats the previous connection (Event 10) by connecting to the same host or Internet address.

It is natural for a user input or data arrival event to trigger a new connection in the same process (Case II and III). It is also normal to repeat a recent connection in the same process (Case IV). Note that connections of email clients repeatedly pulling emails falls into this case. However, Case I implies that it is necessary to correlate events among processes. In general, a user-intended connection must be triggered by one of the rules below:

- *Intra-process* rule: A connection of a process may be triggered by a user input, data arrival or connection request event of the same process.

- *Inter-process* rule: A connection of a process may be triggered by a user input or data arrival event of another process.

To verify if a connection is triggered by the *intra-process* rule, we just need to monitor all user and network activities of each single process. However, we need to monitor all possible communications among processes to verify if a connection is triggered by the *inter-process* rule. In our current design, we only consider communications from a parent process to its *direct* child process and use the following *parent-process* rule to approximate the *inter-process* rule. In other words, we do not allow cascading in applying the parent-process rule.

- *Parent-process* rule: A connection of a process may be triggered by a user input or data arrival event received by its *direct* parent process before it is created.

3.2.3 Extrusion Detection Algorithm

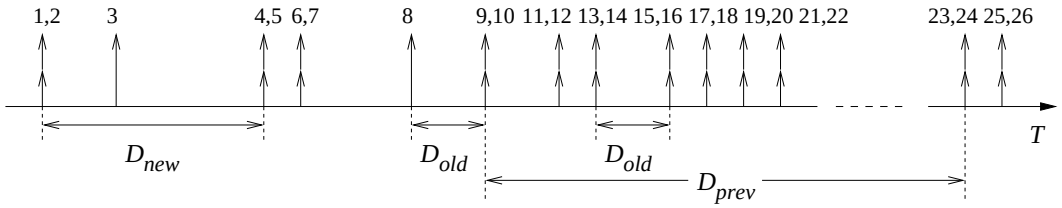


Figure 3.2. A time diagram of events in Figure 3.1.

The extrusion detection algorithm must decide if a connection is user-intended and if it is in the whitelist. Its main idea is to limit the delay from a triggering event to a connection request event. Note that, for data arrival events, we only consider those of user-intended connections. There are three possible delays for a connection request. In Figure 3.2, we show them in a time diagram of events in Figure 3.1. For a connection request made by process P, we define the three delays as follows:

- D_{new}: The delay since the last user input or data arrival event received by the parent process of P before P is created. It is the delay from Event 1 to 5 in Figure 3.2.

- D_{old}: The delay since the last user input or data arrival event received by P. It is the delay from Event 8 to 10 and from Event 14 to 16 in Figure 3.2.

- D_{prev}: The delay since the last connection request to the same host or IP address made by P. It is the delay from Event 10 to 24 in Figure 3.2.

As D_{old} is the reaction time of a process, D_{new} includes the loading time of a process as well. For user-intended connections, D_{old} and D_{new} are on the order of seconds while D_{prev} is on the order of minutes. Depending on how a user-intended connection is initiated, it must have at least one of the three delays less than its upper bound (defined as D_{new}^{upper}, D_{old}^{upper}, and D_{prev}^{upper}). Note that these delays are dependent on the hardware, operating system, and user behavior. Given a host computer system, we must train the BINDER system to find the correct upper bounds. In Section 3.4.2 we will discuss how to choose these upper bounds.

In the design of the extrusion detection algorithm, we assume that we can learn rules from previous false alarms. Each rule includes an application name (the image file name of a process with the complete path) and a remote host (host name and/or IP address). The existence of a rule means that any connection to that host made by a process of that application is not considered to be an extrusion.

Given a connection request, the detection algorithm works as follows:

- If it is in the rule set of previous false alarms, then return;

- If it is in the whitelist, then return;

- If D_{prev} exists and is less than D_{prev}^{upper}, then return;

- If D_{new} exists and is less than D_{new}^{upper}, then return;

- If D_{old} exists and is less than D_{old}^{upper}, then return;

- Otherwise, it is an extrusion.

After detecting an extrusion, we can either drop the connection or raise an alarm with related information such as the process ID, the image file name, and the connection information. Studying the tradeoff between different reactions is beyond the scope of this chapter.

3.2.4 Detecting Break-Ins

We just described the extrusion detection algorithm. Next, we discuss why outbound connections of worms, spyware and adware can be detected as extrusions. Unlike worms, spyware and adware cannot propagate themselves and thus require user input to infect a computer system. Worms can be classified as self-activated like Blaster [97] or user-activated like email worms. The latter also requires user input to infect a personal computer.

When the malware receives user input for its break-in, its connections shortly after the break-in may be masked by user activity. Thus we may not be able to detect these initial extrusions. However, we can detect a break-in as long as we can capture the first non-user triggered connection. In Section 3.4, we will show that we successfully detected the adware Spydeleter [3] and 22 email worms shortly after their break-ins.

When the malware runs without user input, we can easily detect its first outbound connection as an extrusion. This is because the malware runs as a background process and does not receive any user input. So D_{old}, D_{new} and D_{prev} of the first connection do not exist. Additionally, we clear the user input history after a personal computer is restarted. So even for the malware that received user input for its break-in, it is guaranteed to detect its first connection as an extrusion after the victim system is restarted.

In summary, BINDER can detect a large class of malware such as worms, spyware and adware that (1) run as background processes, (2) do not receive any user-driven input, and (3) make outbound network connections.

3.3 Implementation

In the previous section, we described the extrusion detection algorithm. In this section, we describe the architecture and implementation of BINDER (Break-IN DEtectoR), a host-based detection system which, realizing the extrusion detection algorithm, can detect break-ins of new unknown malware on personal computers.

3.3.1 BINDER Architecture

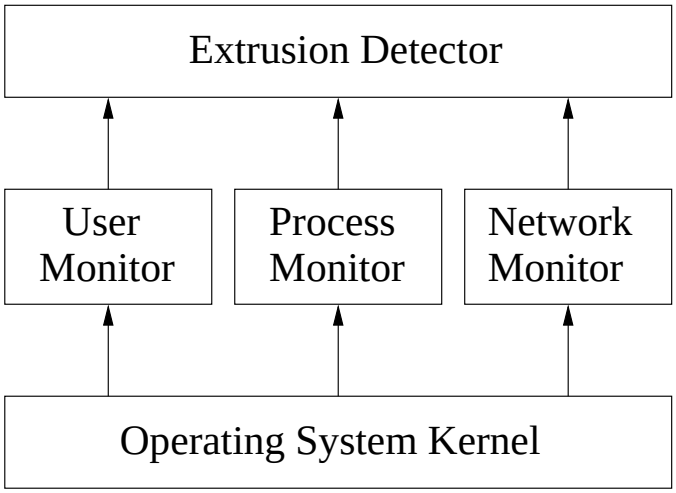


Figure 3.3. BINDER's architecture.

As a host-based system, BINDER must have small overhead so that it does not affect the performance of the host computer. Therefore, BINDER takes advantage of passive monitoring. To detect extrusions, BINDER correlates information across three sources: user-driven input, processes, and network traffic. Therefore, there are four components in a BINDER system: *User Monitor*, *Process Monitor*, *Network Monitor*, and *Extrusion Detector*. The architecture of BINDER is shown in Figure 3.3. The first three components *independently* collect information from the operating system (OS) *passively* in real time and report user, process, and network events to the Extrusion Detector. APIs for real-time monitoring are specific to each operating system. Since BINDER is built to demonstrate the feasibility and effectiveness of the extrusion detection technique, we implement BINDER in the application domain rather than in the kernel for ease of implementation and practical limitation (i.e., no access to Windows source code). We describe the implementation on Windows operating system in the second part of this section. In the following, we explain the functionality and interface of these components that are general to all operating systems.

The User Monitor is responsible for monitoring user input and reporting user events to the Extrusion Detector. It reports a user input event when observing a user clicks the mouse or hits a key. A user input event has two components: the time when it happens and the ID of the process that receives this user input. This mapping between a user input and a process is provided by the

operating system. So the User Monitor does not rely on the Process Monitor for such information. Since a user input event has only the time information and the Extrusion Detector only stores the last user input event, BINDER avoids storing or leaking privacy-sensitive information.

When a process is created or stopped, the Process Monitor correspondingly reports to the Extrusion Detector two types of process events: process start and process finish. A process start event includes the time, the ID of the process itself, its image file name, and the ID of the parent process. A process finish event has only the time and the process ID.

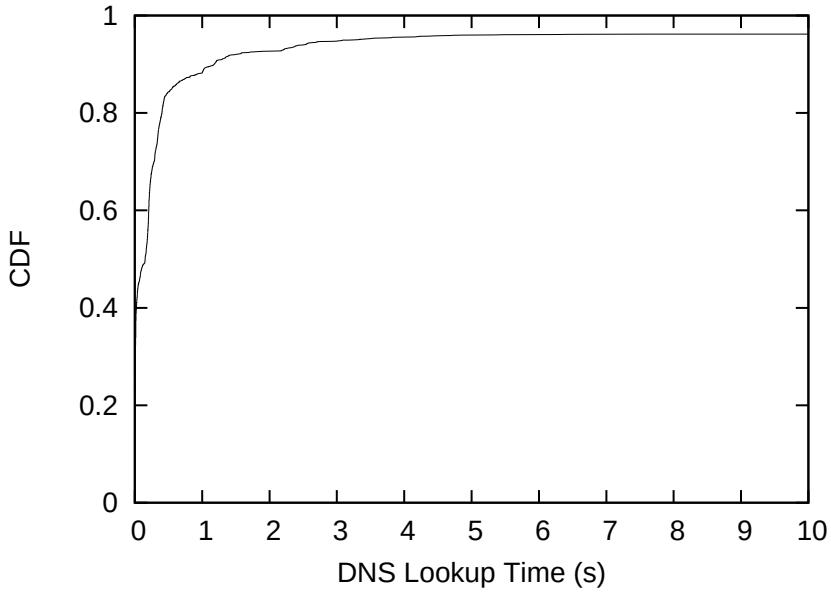


Figure 3.4. CDF of the DNS lookup time on an experimental computer.

The Network Monitor audits network traffic and reports network events. For the sake of detecting extrusions, it reports three types of network events: connection request, data arrival and domain name lookup. For connection request events, the Network Monitor checks TCP SYN packets and UDP packets. A data arrival event is reported when an inbound TCP or UDP packet with non-empty payload is received from a normal outbound connections. Note that the direction of a connection is determined by the direction of the first TCP SYN or UDP packet of this connection. The *Network Monitor* also parses DNS lookup packets. It associates a successful DNS lookup with a following connection request to the same remote IP address as returned in the lookup. This is important because DNS lookup may take significant time between a user input and the corresponding connection

request. The Cumulative Distribution Function (CDF) of 2,644 DNS lookup times on one of the experimental computers (see Section 3.4.2 for details of the experiments) is shown in Figure 3.4. We can see that about 8% DNS lookups take more than 2 seconds. A connection request event has five components: the time, the process ID, the local transport port number, the remote IP address and the remote transport port number. Note that the time is the starting time of its DNS lookup if it has any or the connection itself. The mapping between network traffic and processes is provided by the operating system. A data arrival event has the same components as a connection request event except that its time is the time when the data packet is received. A domain name lookup event has the time, the domain name for lookup, and a list of IP addresses mapping to it.

Except for domain name lookup results that are shared among all processes, the Extrusion Detector organizes events based on processes and maintains a data record for each process. A process data record has the following members: the process ID, the image file name, the parent process ID, the time of the last user input event, the time of the last data arrival event, and all the previous normal connections. When a process start event is received, a process data record is created with the process ID, the image file name and the parent process ID. The time of the last user input event is updated when a user input event of the process is reported. Similarly, the time of the last data arrival is updated when a data arrival event is received. A process data record is closed when its corresponding process finish event is received. All process records are cleared when the system is shutdown. The size of the event database is small because the number of simultaneous processes on a personal computer is usually less than 100. Based on all the information of user, process and network events, the Extrusion Detector implements the extrusion detection algorithm.

3.3.2 Windows Prototype

BINDER's general design (see Figure 3.3) can be implemented in all modern operating systems. Since computers running Windows operating systems are the largest community targeted by malware [86], we implemented a prototype of BINDER for Windows 2000/XP. Given current Windows systems' limitations [70], our Windows prototype does *not* provide a bulletproof solution for break-in detection although it does demonstrate effectiveness of the extrusion detection technique on detecting a large class of existing malware. Though this prototype is implemented in the applica-

tion space, a BINDER system can run in the kernel space if it is adopted in practice. A widespread availability of Trusted Computing-related Virtual Machine-based protection [33] or similar isolation techniques are necessary to turn BINDER into robust production.

The User Monitor is implemented with the Windows Hooks API [108]. It uses three hook procedures, KeyboardProc, MouseProc and CBTProc. KeyboardProc is used to monitor keyboard events while MouseProc is used to monitor mouse events. MouseProc can provide the information of which window will receive a mouse event. Since KeyboardProc cannot provide the same information for a keyboard event, we use CBTProc to monitor events when a window is about to receive the keyboard focus. After determining which window will receive a user input event, the User Monitor uses the procedure GetWindowThreadProcessId to get the process ID of the window.

The Process Monitor is implemented based on the built-in Security Auditing on Windows 2000/XP [109]. By turning on the local security policy of auditing process tracking (Computer Configuration/Windows Settings/Security Settings/Local Policies/Audit Policy/Audit process tracking), the Windows operating system can audit detailed tracking information for process start and finish events. The Process Monitor uses psloglist [67] to parse the security event log and generates process start and process finish events.

The Network Monitor is implemented based on TDIMon [91] and WinDump [110] which requires WinPcap [111]. TDIMon monitors activity at the Transport Driver Interface (TDI) level of networking operations in the operating system kernel. It can provide all network events associated with the process information. Since TDIMon does not save complete DNS packets, the Network Monitor uses WinDump for this purpose. Based on the information collected by TDIMon and DNS packets stored by WinDump, the Network Monitor reports network events to the Extrusion Detector.

It is straightforward to implement the extrusion detection algorithm based on the information stored in the process data record in the Extrusion Detector. Here we focus on the whitelisting mechanism in our Windows implementation. The whitelist in our current implementation has 15 rules. These cover three kinds of programs: system daemons, software updates and network applications automatically started by Windows. A rule for system daemons has only a program name (with a

full path). Processes of the program are allowed to make connections at any time. In our current implementation, we have five system daemons including System, spoolsv.exe, svchost.exe, services.exe and lsass.exe. Note that the traffic of network protocols such as DHCP, ARP, and NTP is initiated by the system daemons. A rule for software updates has both a program name and an update web site. Processes of the program are allowed to connect to the update web site at any time. In this category, we now have six rules that cover Symantec, Sygate, ZoneAlarm, Real Player, Microsoft Office, and Mozilla. For network applications automatically started by Windows when it starts, we currently have four rules for messenger programs of MSN, Yahoo!, AOL, and ICQ. These programs are allowed to make connections at any time. In the future, we need to include a special rule regarding wireless network status change. For example, an email client on a laptop computer may start sending pre-written emails right after the laptop is connected to the wireless network in a hot spot. In addition, we need to study the behavior of peer-to-peer software applications to decide if it is necessary to treat them as system daemons to avoid false positives.

Managing the whitelist for an average user is very important. Rules for system daemons usually do not change until the operating systems are upgraded. Since the number of software applications that require regular updates is small and do not change very often, the rules for software updates can be updated by some central entity that adopts BINDER. Though rules in the last category have to be configured individually for each system, we believe some central entity can provide help by maintaining a list of applications that fall into this category. A mechanism similar to *PeerPressure* [103] may be used to help an average user configure her own whitelist.

3.4 Evaluation

We evaluated BINDER on false positives and false negatives in two environments. First, we installed it on six Windows computers used by different volunteers for their daily work, and collected traces over five weeks since September 7th, 2004. Second, in a controlled testbed based on the Click modular router [39] and VMware Workstation [95], we tested BINDER with the Blaster worm and 22 different email worms collected on a departmental email server over one week since October 7th, 2004.

3.4.1 Methodology

The most important design objective of BINDER is to minimize false alarms while maximizing detected extrusions. In our experiments, we use the number of false alarms rather than the false positive rate to evaluate BINDER. This is because users who respond to alarms are more sensitive to the absolute number than a relative rate. When BINDER detects extrusions, it is based on connections. However, when we count the number of false alarms, we do not use the number of misclassified normal connections directly. This is because a false alarm covers a series of consecutive connection requests (e.g., after the first misclassified normal connection is corrected, consecutive connection requests that repeat it by connecting to the same host or IP address will be treated as normal). Therefore, for misclassified normal connections, we merge them into groups, in which the first false alarm covers the rest, and count each group as one false alarm. When we evaluate BINDER on false negatives, we check if and how fast it can detect a break-in. The real world experiments are used to evaluate BINDER for both false positives and false negatives, while the experiments in the controlled testbed are only for false negatives.

To evaluate BINDER with different values for the three parameters D_{old}^{upper}, D_{new}^{upper} and D_{prev}^{upper}, we use offline, trace-based analysis in all experiments.

3.4.2 Real World Experiments

Table 3.1. Summary of collected traces in real world experiments.

User	Machine	OS	Days	User Events	Process Events	Network Apps	TCP Conns
A	Desktop	WinXP	27	35,270	5,048	33	33,480
B	Desktop	WinXP	26	80,497	12,502	35	15,450
C	Desktop	WinXP	23	24,781	7,487	55	36,077
D	Laptop	Win2K	23	99,928	8,345	28	9,784
E	Laptop	WinXP	13	8,630	2,448	21	10,210
F	Laptop	WinXP	12	20,490	5,402	20	7,592

To evaluate the performance of BINDER in a real world environment, we installed it on six Windows computers used by different volunteers for their daily work, and collected traces over five weeks. We collected traces of user input, process information, and network traffic from the six

computers. A summary of the collected traces is shown in Table 3.1. On one hand, these computers were used for daily work, so the traces are realistic. On the other hand, our experimental population is small because it is difficult to convince users to submit their working environment to experimental software. However, from the summary of the collected traces in Table 3.1, we see that they have several distinct combinations of hardware, operating system, and user behavior. For real world experiments, we discuss parameter selection and then analyze the performance of our approach on false positives and false negatives. Due to the limitation of data collection, we train BINDER and evaluate it on the same data set.

Parameter Selection

Table 3.2. Parameter selection for D_{old}, D_{new} and D_{prev}.

	90% (sec)				95% (sec)				99% (sec)			
User	D_{old}	D_{new}	D_{prev}	# of FAs	D_{old}	D_{new}	D_{prev}	# of FAs	D_{old}	D_{new}	D_{prev}	# of FAs
A	18	11	142	15	33	15	752	5	79	21	4973	3
B	15	12	64	23	28	21	260	7	79	22	3329	5
C	14	14	28	20	25	15	134	5	74	33	2272	1
D	16	81	213	3	33	81	715	3	85	81	4611	2
E	19	12	539	5	32	14	541	4	93	90	4216	3
F	14	8	80	10	27	13	265	5	79	31	3633	2

In this section, we discuss how to choose values for the three parameters D_{old}^{upper}, D_{new}^{upper} and D_{prev}^{upper}. The goal of parameter selection is to make the tightest possible upper bounds under the condition that the number of false alarms is acceptable. We assume the rules of whitelisting described in Section 3.3.2 are fixed. The performance metric is the number of false alarms. Based on the real-world traces, we calculate D_{old}, D_{new} and D_{prev} for all connection request events for every user. Then we take the 90th, 95th and 99th percentile for all three parameters and calculate the number of false alarms for each percentile. The results are shown in Table 3.2.

From Table 3.2 we can see that D_{old}^{upper}, D_{new}^{upper} and D_{prev}^{upper} must be different for different users because they are dependent on computer speed and user patterns. Thus, they should be selected on a per-user basis. We can also see that the performance of taking the 90th percentile is not acceptable and the improvement from taking the 95th percentile to the 99th percentile is small. Therefore, the parameters can be selected by choosing some value in the 95th percentile or according to user's preference. For conservative users, we should choose smaller values. The percentiles can

be obtained by training BINDER over a period of virus-free time. Without training, these parameters can also be chosen (in some range) based on user's preference. D_{old}^{upper} and D_{new}^{upper} can take values between 10 and 60 seconds, while D_{prev}^{upper} can take values between 600 and 3600 seconds. Note that D_{old}^{upper} can be greater than D_{new}^{upper} because the reaction time from a user input or data arrival event to a connection request event is dependent on the instantaneous state of a computer. It remains a challenge for future research to investigate effective solutions for parameter selection.

False Positives

Table 3.3. Break-down of false alarms according to causes.

User	Inter-Process	Whitelist	Collection	Total
A	2	1	0	3
B	4	1	0	5
C	1	0	0	1
D	0	1	1	2
E	1	1	1	3
F	0	1	1	2

By choosing parameters correctly, we expect to achieve few false alarms. From Table 3.2 we can see that there were at most five false alarms (over 26 days) for each computer by choosing the 99th percentile. The false positive rate (i.e., the probability that a benign event causes an alarm) is 0.03%. We manually checked these remaining false alarms and found that they were caused by one of three reasons:

- Incomplete information about inter-process event sharing. For example, four of the five false alarms of User B were caused by this. We observe that PowerPoint followed IE to connect to the same IP address while the parent process of the PowerPoint process was the Windows shell. We hypothesize this is due to the usage of Windows API ShellExecute (which was called by IE when the user clicked on a link of a PowerPoint file).

- Incomplete whitelisting. For example, connections made by Windows Application Layer Gateway Service were (incorrectly) treated as extrusions.

- Incomplete trace collection. BINDER was accidentally turned off by a user in the middle of

trace collection. We gave users the control of turning on/off BINDER in case they thought BINDER was responsible for bad performance, etc.

A break-down of the false alarms is shown in Table 3.3. We can see that a better-engineered BINDER can eliminate false alarms in the Whitelist and Collection columns in Table 3.3, resulting in much lower false positives. By extending BINDER to consider more inter-process communications than those between parent-child processes, we can decrease the false alarms in the Inter-Process column.

False Negatives

```
 1 14:40:10, PID=2368, PPID=240, NAME="C:\...\iexplore.exe" (process start)
 2 14:40:15, PID=2368 (user input)
 3 14:40:24, PID=2368 (user input)
 4 14:40:24, PID=2368, LPORT=1054, RIP=12.34.56.78, RPORT=80 (connection request)
 5 14:40:24, PID=2368, LPORT=1054, RIP=12.34.56.78, RPORT=80 (data arrival)
   ......
 6 14:40:28, PID=2552, PPID=960, NAME="C:\...\mshta.exe" (process start)
 7 14:40:29, PID=2552, LPORT=1066, RIP=87.65.43.21, RPORT=80 (connection request)
 7 14:40:29, PID=2552, LPORT=1066, RIP=87.65.43.21, RPORT=80 (data arrival)
   ......
 8 14:40:34, PID=2896, PPID=2552, NAME="C:\...\ntvdm.exe" (process start)
 9 14:40:35, PID=2988, PPID=2896, NAME="C:\...\ ftp.exe" (process start)
10 14:40:35, PID=2988, LPORT=1068, RIP=44.33.22.11, RPORT=21 (connection request)
   ......
```

Figure 3.5. A stripped list of events logged during the break-in of adware Spydeleter.

In the real world experiments, among the six computers, one was infected by the adware Gator [2] and CNSMIN [1] and another one was infected by the adware Gator and Spydeleter [3]. In particular, the second computer was compromised by Spydeleter when BINDER was running. BINDER can successfully detect the adware Gator and CNSMIN because they do not have any user input in history. In the following, we demonstrate how BINDER can detect the break-in of Spydeleter right after it compromised the victim computer.

In Figure 3.5, we show a stripped list of events logged during the break-in of the adware Spydeleter. Note that all IP addresses are anonymized. Two related processes not shown in the list are

a process of explorer.exe with PID 240 and a process of svchost.exe with PID 960. After IE is opened, a user connects to a site with IP 12.34.56.78. The web page has code to exploit a vulnerability in mshta.exe that processes .HTA files. After mshta.exe is infected by the malicious .HTA file that is downloaded from 87.65.43.21, it starts a series of processes of ntvdm.exe that provides an environment for a 16-bit process to execute on a 32-bit platform. Then, a process of ntvdm.exe starts a process of ftp.exe that makes a connection request to 44.33.22.11. Since the prototype of BINDER does not have complete information for verifying if a connection is triggered according to the inter-process rule (see Section 3.2), the connection made by mshta.exe is detected as an extrusion. This is because its parent process is svchost.exe rather than iexplore.exe, though it is the latter process that triggers its creation. If BINDER had complete information for inter-process event sharing, it would detect the connection request made by ftp.exe as an extrusion. This is because we do not allow cascading in applying the parent-process rule (this may cause false positives, but we leave it for future research to study this tradeoff) and both the process of ftp.exe and its parent process of ntvdm.exe does not have any user input or data arrival event in its history. So D_{old}, D_{new} and D_{prev} for the connection request made by ftp.exe do not exist. This connection is used to download malicious code. Therefore, BINDER's detection with some appropriate actions could have stopped the adware from infecting the computer. Note that the three parameters of D_{old}^{upper}, D_{new}^{upper} and D_{prev}^{upper} do not affect BINDER's detection of Spydeleter here.

3.4.3 Controlled Testbed Experiments

Our real world experiments show that BINDER has a small false positive rate. Since the number of break-ins in our real world experiments is very limited, to evaluate BINDER's performance on false negatives, we need to test BINDER with more real world malware. To do so, we face a challenge of repeating break-ins of real world malware without unwanted damage. We tackle this problem by building a controlled testbed. Next, we describe our controlled testbed and present experimental results on 22 email worms and the Blaster worm.

Controlled Testbed

We developed a controlled testbed using the Click modular router [39] and VMware Worksta-

tion [95]. We chose Click for its powerful modules [40] for packet filtering and Network Address and Port Translation (NAPT). In addition, we implemented a containment module in Click that can pass, redirect or drop outbound connections according to predefined policies. The advantage of using VMware is that we can discard an infected system and get a new one just by copying a few files. VMware also provides virtual private networks on the host system. The testbed is shown in Figure 3.6.

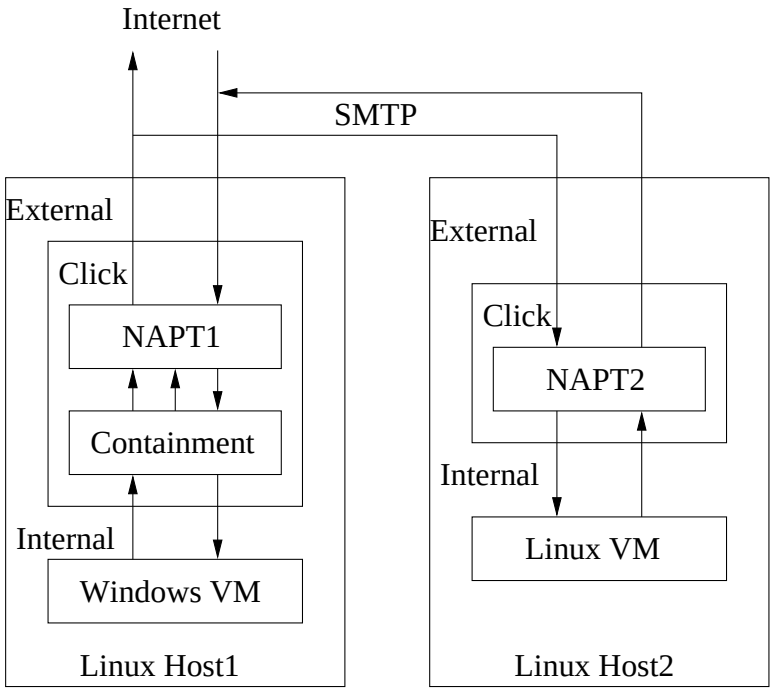


Figure 3.6. The controlled testbed for evaluating BINDER with real world malware.

In the testbed, we have two Linux hosts running VMware Workstation. On the first host, we have a Windows virtual machine (VM) in a host-only private network. This VM is used for executing malicious code attached in email worms. The Click router on this host includes a containment module and a NAPT module. The containment policy on this router allows DNS traffic pass through and redirects all SMTP traffic (to port 25) to another Linux host. The second Linux host has a Linux VM running the eXtremail server [22] in a host-only private network. The email server is configured to accept all relay requests. The Click router on this host also has a NAPT module that guarantees the email server can only receive inbound SMTP connections. Thus, all malicious emails are con-

tained in the email server. This controlled testbed enables us to repeat the whole break-in and propagation process of email worms.

Experiments with Email Worms

Table 3.4. Email worms tested in the controlled testbed.

W32.Beagle.AB@mm	W32.Beagle.AC@mm
W32.Beagle.AG@mm	W32.Beagle.AR@mm
W32.Beagle.M@mm	W32.Bugbear.B@mm
W32.Erkez.B@mm	W32.Mydoom.L@mm
W32.Mydoom.M@mm	W32.Netsky.AB@mm
W32.Netsky.AD@mm	W32.Netsky.B@mm
W32.Netsky.C@mm	W32.Netsky.D@mm
W32.Netsky.F@mm	W32.Netsky.K@mm
W32.Netsky.P@mm	W32.Netsky.Q@mm
W32.Netsky.S@mm	W32.Netsky.W@mm
W32.Netsky.Z@mm	W32.Swen.A@mm

We obtained email worms from two sources. First, we set up our own email server and published an email address to USENET groups. This resulted in the email worm W32.Swen.A@mm (we use Symantec's naming rule [88]) being sent to us. Second, we were fortunate to convince the system administrators of our department email server to give us 1,843 filtered virus email attachments that were collected over the week starting on October 7th, 2004. We used Symantec Norton Antivirus [87] to scan these attachments and recognized 27 unique email worms. Among them, we used 21 email worms because the rest of them were encrypted with a password. Thus, we tested 22 different real world email worms shown in Table 3.4.

For each email worm, we manually start it on a Windows virtual machine that has a file of 10 real email addresses, let it run for 10 minutes (with user input in history), and then restart the virtual machine to run another 10 minutes (without user input in history). We analyze BINDER's performance using the traces collected during the two 10 minute periods. We choose 10 minutes because they are long enough for email worms to scan hard disk, find email addresses, and send malicious emails to them.

Our results show that BINDER successfully detects break-ins of all 22 email worms in the

second 10 minute period by identifying the very first malicious outbound connection. In the rest of this section, we focus on BINDER's performance in the first 10 minute period.

Table 3.5. The impact of D_{old}^{upper}/D_{new}^{upper} on BINDER's performance of false negatives.

$D_{old}^{upper} = D_{new}^{upper}$ (sec)	10	20	30	40	50	60
Num of email worms detected	22	21	21	19	17	15

According to our discussion on parameter selection, D_{old}^{upper} and D_{new}^{upper} usually take values between 10 and 60 seconds, while D_{prev}^{upper} usually takes values between 600 and 3600 seconds. Since our traces are 10 minute long, the parameter D_{prev}^{upper} does not affect BINDER's performance on false negatives. So we study the impact of D_{old}^{upper} and D_{new}^{upper} on BINDER's performance of false negatives. In Table 3.5, we show the number of email worms are detected by BINDER when D_{old}^{upper} and D_{new}^{upper} take a same given value between 10 and 60 seconds. We have only one email worm (W32.Swen.A@mm) *missed* when we take 30 seconds for D_{old}^{upper} and D_{new}^{upper}. This is because the first connection is detected as user-intended due to the user input and all following connections repeat the first one by connecting to the same Internet address.

Experiments with Blaster

We test BINDER with the Blaster worm [97]. In this experiment, we run two Windows XP VMs A and B in a private network. We run BINDER on VM B and run msblast.exe on VM A. Blaster on VM A scans the network, finds VM B and infects it. By analyzing the infection trace collected by BINDER, we see that BINDER detects the first outbound connection made by a process tftp.exe as an extrusion. This is because the process itself and its parent process of cmd.exe does not receive any user input. Thus we can successfully detect Blaster in this case even before the worm itself is transferred over by TFTP.

3.5 Limitations

Our limited user study shows that BINDER controls the number of false alarms to at most five over four weeks on each computer. We also show that BINDER successfully detects break-ins

of the adware Gator, CNMIN, and Spydeleter, the Blaster worm, and 22 email worms. However, BINDER is far from a complete system. It is built to verify that user intent can be a simple and effective detector of a large class of malware with a very low false positive rate. We devote this section to discussions of potential attacks against BINDER if its scheme is known to adversaries. Though we try to investigate all possible attacks against BINDER, we cannot argue that we have considered all of its possible vulnerabilities. Potential attacks include:

- Direct attack: subvert BINDER on the compromised system;
- Hiding inside other processes: inject malicious code into other processes;
- Faking user input: use APIs provided by the operating system to generate synthesized actions.
- Tricking the user to input: trick users to click on pop-up windows or a transparent overlay window that intercepts all user input.
- Exploiting the whitelist: replace the executables of programs in the whitelist with a tweaked one;
- Exploiting user input in history: when a malicious process is allowed to make one outbound connection due to user input (e.g., a user opens a malicious email attachment), it can evade BINDER's detection by making that connection to a collusive remote site to keep receiving data. This would make BINDER think any new connections made by this process are triggered by those data arrivals.
- Covert channels: a very tricky attack is to have a legitimate process make connections and use them as a covert channel to leak information. For example, spyware can have an existing IE process download a web page of a tweaked hyperlink by using some API provided by the Windows shell right after a user clicks on the IE window of the same process. A collusive remote server can get private information from the tweaked hyperlink.

Direct attack is a general limitation of all end-host software (e.g., antivirus, personal firewalls [120], and virus throttles [107] that attempt to limit outgoing attacks). A widespread availability of Trusted Computing-related Virtual Machine-based protection [33] or similar isolation techniques are necessary to turn BINDER or any of these other systems into robust production.

The attacks of hiding inside other processes, faking user input, tricking users to input, and exploiting whitelisting are inherent to the limitations of today's operating systems. The effectiveness of BINDER on malware detection implies pressing requirements for isolation among processes and trustworthy user input in next-generation operating systems. Possible incomplete solutions for these attacks might be to monitor corresponding system APIs or to verify the integrity of programs listed in the whitelist. Even without a bulletproof solution for today's operating system, we believe a deployed BINDER system can raise the bar for adversaries significantly.

For the attacks of exploiting user input in history, a possible solution is to add more constraints on how a user intended connection may be triggered. This requires more research work in the future. For the attack of covert channels, possible solutions are discussed in [10].

3.6 Summary

In this chapter, we presented the design of a novel extrusion detection algorithm and the implementation of a host-based detection system which, realizing the extrusion detection algorithm, can detect break-ins of a wide class of malware, including worms, spyware and adware, on personal computers by identifying their extrusions. Our main contributions are:

- We take advantage of a unique characteristic of personal computers—user intent. Our evaluations suggest that user intent is a simple and effective detector for a large class of malware with few false alarms.
- The controlled testbed based on the Click modular router and VMware Workstation enables us to repeat the whole break-in and propagation process of email worms without worrying about unwanted damage.

BINDER is very effective at detecting break-ins of malware on personal computers, but it also has limitations:

- Leveraging user intent, it only works for personal computers.

- Running solely on a single host, it is poor at providing "early warning" of new worm outbreaks.
- It needs more extensive evaluation for actual deployment.

To address these limitations, in the next two chapters we describe how we can infer adversary intent on a large-scale honeypot system to detect fast spreading Internet worms automatically and quickly.

Chapter 4

Protocol Independent Replay of Application Dialog

In the previous chapter, we described a novel extrusion detection algorithm and the BINDER system for detecting break-ins of new unknown malware on personal computers. BINDER holds promise to be effective on detecting a wide class of malware with few false alarms, but it would be difficult to scale to provide "early warning" of new Internet worm outbreaks. For early automatic detection of new worm outbreaks, we built a large-scale, high-fidelity "honeyfarm" system that analyzes in real-time the scanning probes seen on a quarter million Internet addresses. To achieve such a scale, the honeyfarm system leverages a novel technique of *protocol-independent replay* of application dialog to winnow down the hundreds of probes per second seen by the honeyfarm to a level that it can process. Before describing the honeyfarm system in detail in Chapter 5, we present in this chapter the design of this novel technique and an evaluation over a variety of real-world network applications. This chapter is organized as follows. After motivating our work (see Section 4.1), we first present an overview of design goals, challenges, terminology, and assumptions (see Section 4.2). Then, we describe our replay system in detail and discuss related design issues (see Section 4.3). Next, we describe our implementation, evaluate our system with real-world network applications, and discuss the limitations (see Section 4.4). We demonstrate that our system successfully replays the client and server sides for real-world network applications such as NFS, FTP, and CIFS/SMB

file transfers as well as the multi-stage infection processes of real-world worms such as Blaster and W32.Randex.D. Finally, we summarize in Section 4.5.

4.1 Motivation

In a number of different situations it would be highly useful if we could cheaply "replay" one side of an application session in a slightly different context. For example, consider the problem of receiving probes from a remote host and attempting to determine whether the probes reflect a new type of malware or an already known attack. Different attacks that exploit the same vulnerability often conduct the same application dialog prior to finally exposing their unique malicious intent. Thus, to coax from these attackers their final intent requires engaging them in an initial dialog, either by implementing protocol-specific application-level responders [63, 66] or by deploying high-interaction honeypot systems [29, 89] that run the actual vulnerable services. Both approaches are expensive in terms of development or management overhead.

On the other hand, much of the dialog required to engage with the remote source follows the same "script" as seen in the past, with only very minor variants (different hostnames, IP addresses, port numbers, command strings, or session cookies). If we could cheaply *reproduce* one side of the dialog, we could directly tease out the remote source's intent by efficiently driving it through the routine part of the dialog until it reaches the point where, if it is indeed something new, it will reveal its distinguishing nature by parting from the script.

A powerful example concerns the use of replay to construct *proxies* in our honeyfarm system. Suppose in the first example above that not only do we wish to determine whether an incoming probe reflects a new type of attack or a known type, but we also want to *filter* the attack if it is a known type but allow it through if it is a new type. We could use replay to efficiently do so in two steps (see Figure 5.2 in Section 5.2.4 for an illustration). First, we would replay the targeted server's behavior in response to the probe's initial activity (e.g., setting up a Windows SMB RPC call) until we reach a point where a new attack variant will manifest itself (e.g., by making a new type of SMB call or by sending over a previously unseen payload for an existing type of call). If the remote host at this point proves to lack novelty (we see the same final step in its activity as we have seen before),

then we drop the connection. However, if it reveals a novel next step, then at this point we would like it to engage a high-interaction honeypot server so we can examine the new attack. To do so, though, we must bring the server "up to speed" with respect to the application dialog in which the remote host is already engaged. We can do so by using replay *again*, this time replaying the remote host's previous actions to the new server so that the two systems become synchronized, after which we can allow the remote host to proceed with its probing. If we perform such proxying correctly, the remote host will never know that it was switched from one type of responder (our initial replayer) to another (the high-interaction honeypot).

Another example comes from trying to determine the equivalent for malware of a "toxicology spread" for a biological pathogen. That is, given only an observed instance of a successful attack against a given type of server (i.e., particular OS version and server patch level), how can we determine what other server/OS versions are also susceptible? If we have instances of other possible versions available, then we could test their vulnerability by replaying the original attack against them, providing the replay again takes into account the natural session variants such as differing hostnames, IP addresses, and session cookies. Similarly, we could use replay to feed possible attacks into more sophisticated analysis engines (e.g., [62] and [15]).

We can also use lightweight replay to facilitate testing of network systems. For example, when developing or configuring a new security mechanism it can be very helpful if we can easily repeat attacks against the system to evaluate its response. Historically, this may require a complex testbed to repeatedly run a piece of malware in a safe, restricted fashion. Armed with a replay system, however, we could capture a single instance of an attack and then replay it against refinements of the security mechanism without having to reestablish an environment for the malware to execute within.

We can generalize such repeated replay to tasks of stress-testing, evaluating servers, or conducting large-scale measurements. A replay system could work as a low-cost client by replaying application dialogs with altered parameters in part of the dialog. For example, by dynamically replacing the receiver's email address in a SMTP dialog, we could use replay to create numerous SMTP clients that send the same email to different addresses. We could use such a capability for

both debugging and performance testing, without needing to either create specialized clients or invoke repeated instances of a computationally expensive piece of client software.

The programming and operating systems communities have studied the notion of replaying program execution for a number of years. Debugging was their targeted application for replay. LeBlanc and Mellor-Crummey [43] studied how to leverage replay to debug parallel programs. Russinovich and Cogswell [72] presented the *repeatable scheduling* algorithm for recording and replaying the execution of shared-memory applications. King *et al.* [37] described a time-traveling virtual machine for debugging operating systems. Time travel allows a debugger to go back and forth arbitrarily through the execution history and to replay arbitrary segments of the past execution. Recently, Geels *et al.* [24] developed a replay debugging tool for distributed C/C++ applications. However, we know of little literature discussing general replay of network activity at the application level. The existing work has instead focused on incorporating application-specific semantics. Honeyd [66], a virtual honeypot framework, supports emulating arbitrary services that run as external applications and receive/send data from/to *stdin*/*stdout*.

Libes' *expect* tool includes the same notion as our work of automatically following a "script" of expected interactive dialogs [49]. A significant difference of our work, however, is that we focus on generating the script *automatically* from previous communications.

A number of commercial security products address replaying network traffic [16, 52]. These products however appear limited to replay at the network or transport layer, similar to the *Monkey* and *Tcpreplay* tools [12, 90]. The *Flowreplay* tool uses application-specific plug-ins to support application-level replay [93].

To achieve protocol-independent replay, we develop a new system named RolePlayer. Our work leverages the Needleman-Wunsch algorithm [60], widely used in bioinformatics research, to locate fields that have changed between one example of an application session and another. In this regard, our approach is similar to the recent *Protocol Informatics* project [8], which attempts to identify protocol fields in unknown or poorly documented network protocols by comparing a series of samples using sequence alignment algorithms.

Concurrent with our RolePlayer work, Leita *et al* proposed ScriptGen [46], a system for auto-

matically generating *honeyd* [66] scripts without requiring prior knowledge of application protocols. They presented a novel region analysis algorithm to identify and merge fields in an application protocol. While these results hold promise for a number of applications, it remains a challenge to use the approach for completing interactions with attacks using complex protocols such as SMB.

Before describing the design of RolePlayer, we first present an overview of design goals, challenges, terminology, and assumptions in the next section.

4.2 Overview

In this section, we first describe our design goals. Then, we discuss the challenges in implementing protocol-independent replay. Finally, we present the terminology and design assumptions.

4.2.1 Goals

In developing our replay system, RolePlayer, we aim for the system to achieve several important goals:

- **Protocol independence**. The system should not need any application-specific customization, so that it works transparently for a large class of applications, including both as a client and as a server.
- **Minimal training**. The system should be able to mimic a previously seen type of application dialog given only a small number of examples.
- **Automation**. Given such examples, the system should operate correctly without requiring any manual intervention.

4.2.2 Challenges

In some cases, replaying is trivial to implement. For example, each attack by the Code Red worm sends exactly the same byte stream over the network to a target. However, successfully replaying an application dialog can be much more complicated than simply parroting the stream.

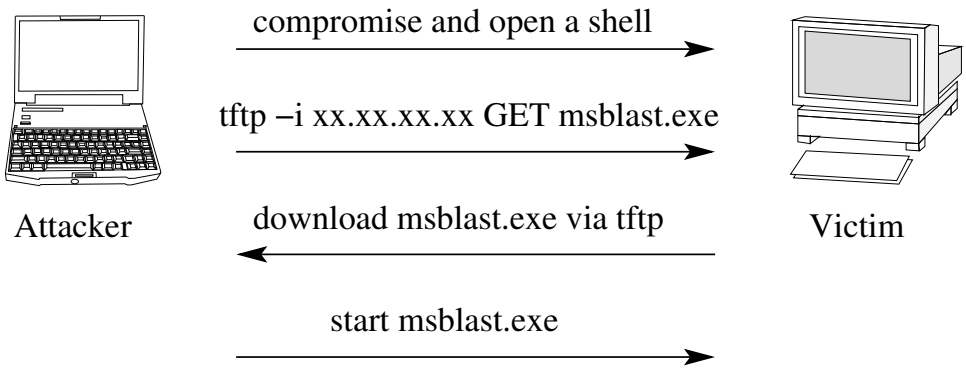


Figure 4.1. The infection process of the Blaster worm.

Consider for example the Blaster worm of August, 2003, which exploited a DCOM RPC vulnerability in Windows by attacking port 135/tcp. For Blaster, if a compromised host (A) finds a new vulnerable host (V) via its random scanning, then the following infection process occurs (see Figure 4.1).

1. A opens a connection to 135/tcp on V and sends three packets with payload sizes of 72, 1460, and 244 bytes. These packets compromise V and open a shell listening on 4444/tcp.

2. A opens a connection to 4444/tcp and issues "tftp -i xx.xx.xx.xx GET msblast.exe" where "xx.xx.xx.xx" is A's IP address.

3. V sends a request back to 69/udp on A to download msblast.exe via TFTP.

4. A issues commands via 4444/tcp to start msblast.exe on host V.

This example illustrates a number of challenges for implementing replay.

1. An application session can involve multiple connections, with the initiator and responder switching roles as both client and server, which means RolePlayer must be able to run as client and server simultaneously.

2. While replaying an application dialog we sometimes cannot coalesce data. For example, if the replayer attempts to send the first Blaster data unit as a single packet, we observe two packets with sizes of 1460 and 316 bytes due to Ethernet framing. For reasons we have been

unable to determine, these packets do not compromise vulnerable hosts. Thus, RolePlayer must consider both application data units and network framing. (It appears that there is a race condition in Blaster's exploitation process—A connects to 4444/tcp before the port is opened on V. If we do not consider the network framing, it increases the likelihood of the race condition. Accommodating such timing issues in replay remains for future work.)

3. Endpoint addresses such as IP addresses, port numbers and hostnames may appear in application data. For example, the IP address of A appears in the TFTP download command. This requires RolePlayer to find and update endpoint addresses dynamically.

4. Endpoint addresses (especially names but also IP addresses, depending on formatting) can have variable lengths, and thus the data size of a packet or application data unit can differ between dialogs. This creates two requirements for RolePlayer: deciding if a received application data unit is complete, particularly when its size is smaller than expected; and changing the value of such length fields when replaying them with different endpoint addresses.

5. Some applications use "cookie" fields to record session state. For example, the process ID of the client program is a cookie field for the Windows CIFS protocol, while the file handle is one in the NFS protocol. Therefore, RolePlayer must locate and update these fields during replay.

6. We observe that non-zero padding up to 3 bytes may be inserted into an application data unit, which we must accommodate without confusion during replay.

4.2.3 Terminology

We will use the term *application session* to mean a fixed series of interactions between two hosts that accomplishes a specific task (e.g., uploading a particular file into a particular location). The term *application dialog* refers to a recorded instance of a particular application session. The host that starts a session is the *initiator*. The initiator contacts the *responder*. (We avoid "client" and "server" here because for some applications the two endpoints assume both roles at different times.) We want RolePlayer to be able to correctly mimic both initiators and responders. In doing

so, it acts as the *replayer*, using previous dialog(s) as a guide in communicating with the remote *live peer* that runs the actual application program.

An application session consists of a set of TCP and/or UDP *connections*, where a UDP "connection" is a pair of unidirectional UDP flows that are matched on source/destination IP address/port number. In a connection, an *application data unit* (ADU) is a consecutive chunk of application-level data sent in one direction, which spans one or more packets. RolePlayer only cares about application-level data, ignoring both network and transport-layer headers when replaying a session.

Table 4.1. Definitions of different types of dynamic fields.

Type	Definition
endpoint-address	hostnames, IP addresses, transport port numbers
length	1 or 2 bytes reflecting the length of either the ADU or a subsequent dynamic field
cookie	session-specific opaque data that appears in ADUs from both sides of the dialog
argument	fields that customize the meaning of a session
don't-care	opaque fields that appear in only one side of the dialog

Within an ADU, a *field* is a byte sequence with semantic meaning. A *dynamic field* is a field that potentially changes between different dialogs. We classify dynamic fields into five types: *endpoint-address*, *length*, *cookie*, *argument*, and *don't-care* (see Table 4.1 for their definitions). Argument fields (e.g., the destination directory for a file transfer or the domain name in a DNS lookup) are only relevant for replay if we want to specifically alter them; for *don't-care* fields, we ignore the difference if the value for them we receive from a live peer differs from the original, and we send them verbatim if communicating them to a live peer.

To replay a session, we need at least one sample dialog—the *primary* application dialog—to use as a reference. We may also need an additional *secondary* dialog for discovering dynamic fields, particularly length fields. Finally, we refer to the dialog generated during replay as the *replay* dialog.

4.2.4 Assumptions

Given the terminology, we can frame our design assumptions as follows.

1. We have available primary and (when needed) secondary dialogs that differ enough to disclose the dynamic fields, but are otherwise the same in terms of the application semantics.

2. We assume that the live peer is configured suitably similar to its counterpart in the dialogs.

3. We assume the knowledge of some standard network protocol representations, such as embedded IP addresses being represented either as a four-byte integer or in dotted or comma-separated format and embedded transport port numbers as two-byte integers. (We currently consider network byte order only, but it is an easy extension to accommodate either endianness.) Note that we do *not* assume the use of a single format, but instead encode into RolePlayer each of the possible formats.

4. We assume the domain names of the participating hosts in sample dialogs can be provided by the user if required for successful replay (see Section 4.4.7 for more discussion on this assumption).

5. We assume that the application session does not include time-related behavior (e.g., wait 30 seconds before sending the next message) and does not require encryption or cryptographic authentication.

We will show that these assumptions do not impede RolePlayer from replaying a wide class of application protocols (see Section 4.4).

4.3 Design

The basic idea of RolePlayer is straightforward: given an example or two of an application session, locate the dynamic fields in the ADUs and adjust them as necessary before sending the ADUs out from the replayer. Since some dynamic fields, such as length fields, can only be found by comparing two existing sample application dialogs, we split the work of RolePlayer into two stages: *preparation* and *replay*. During preparation, RolePlayer first searches for endpoint-address and argument fields in each sample dialog, then searches for length fields and possible cookie fields by comparing the primary and secondary dialogs. During replay, it first searches for new values of dynamic fields by comparing received ADUs with the corresponding ones in the primary dialog, then updates them with the new values. In this section, we describe both stages in detail.

Before proceeding, we note that a particularly important issue concerns dynamic ports. RolePlayer needs to determine when to initiate or accept new connections, and in particular must recognize additional ports that an application protocol dynamically specifies. To do so, RolePlayer detects stand-alone ports and IP address/port pairs during its search process, and matches these with subsequent connection requests. This enables the system to accommodate features such as FTP's use of dynamic ports for data transfers, and portmapping as used by SunRPC.

4.3.1 Preparation Stage

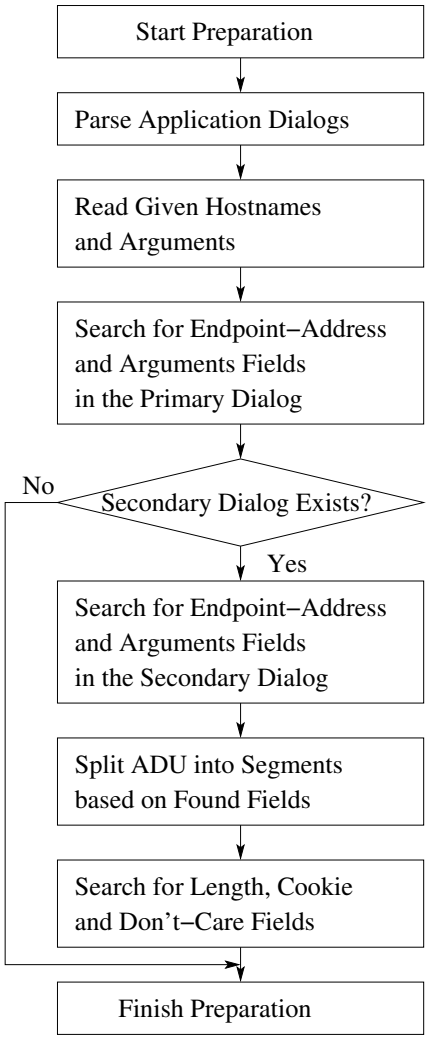


Figure 4.2. Steps in the preparation stage.

In the preparation stage, RolePlayer needs to parse network traces of the application dialogs and

search for the dynamic fields within. For its processing we may also need to inform RolePlayer of the hostnames of both sides of the dialog, and any application arguments of interest, as these cannot be inferred from the dialog. These steps of the preparation stage are shown in Figure 4.2.

Parsing Application Dialogs

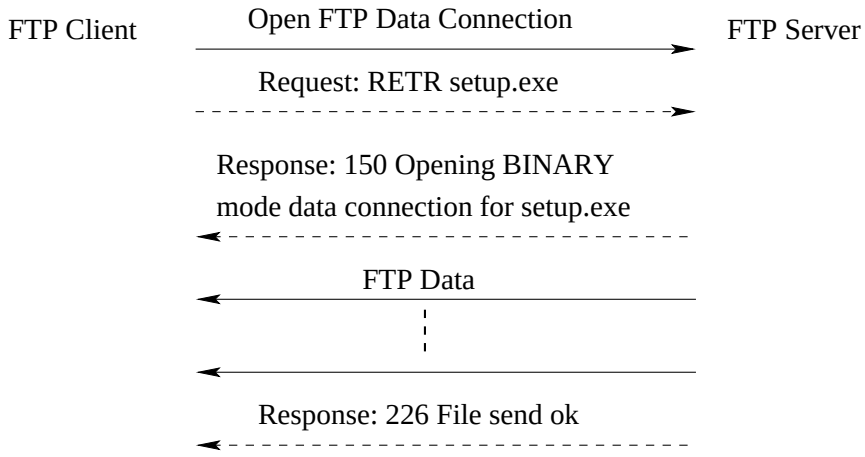


Figure 4.3. A sample dialog for PASV FTP.

RolePlayer organizes dialogs in terms of both ADUs and data packets. It uses data packets as the unit for sending and receiving data and ADUs as the unit for manipulating dynamic fields. Note that data packets may interleave with ADUs. For example, FTP sessions use two concurrent connections, one for data transfer and one for control (see Figure 4.3). RolePlayer needs to honor the ordering between these as revealed by the packet sequencing on the wire, such as ensuring that a file transfer on the data channel completes before sending the "226 Transfer complete" on the control channel. Accordingly, we use the SYN, FIN and RST bits in the TCP header to delimit the beginning and end of each connection, so we know the correct temporal ordering of the connections within a session.

Searching for Dynamic Fields

RolePlayer next searches for dynamic fields in the primary dialog, and also by comparing it with the secondary dialog (if available). The searching process contains a number of subtle steps and considerations (we discuss some design issues in detail in Section 4.3.3). We illustrate these using replay of a fictitious toy protocol, Service Port Discovery (SPD), as a running example.

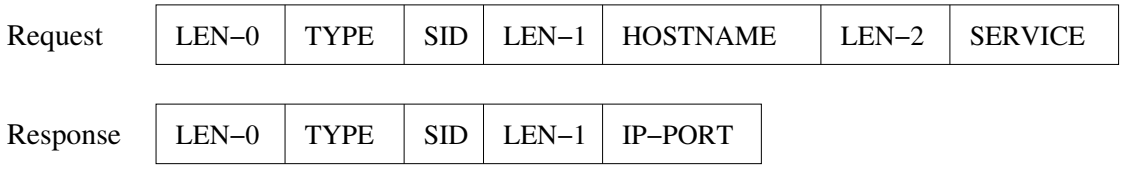


Figure 4.4. The message format for the toy Service Port Discovery protocol.

In SPD, a client sends a request message, carrying the client's hostname and a service name, to a server to ask for the port number of that service. The server replies with an IP address and port number expressed in the comma-separated syntax used by FTP.

These two messages have the formats shown in Figure 4.4. Requests have 7 fields: LEN-0 (1 byte) holds the length of the message. TYPE (1 byte) indicates the message type, with a value of 1 indicating a request and 2 a response message. SID (2 bytes) is session/transaction identifier, which the server must echo in its response. LEN-1 (1 byte) stores the length of the client hostname, HOSTNAME, which the server logs. LEN-2 (1 byte) stores the length of the service name, SERVICE.

Responses have 5 fields. LEN-0, TYPE and SID have the same meanings as in requests. LEN-1 (1 byte) stores the length of the IP-PORT field, which holds the IP address and port number of the requested service in a comma-separated format. For example, 1.2.3.4:567 is expressed as "`0x31 0x2c 0x32 0x2c 0x33 0x2c 0x34 0x2c 0x32 0x2c 0x35 0x35`".

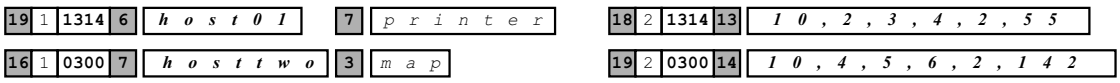


Figure 4.5. The primary and secondary dialogs for requests (left) and responses (right) using the toy SPD protocol. RolePlayer first discovers endpoint-address (bold italic) and argument (italic) fields, then breaks the ADUs into segments (indicated by gaps), and discovers length (gray background) and possible cookie (bold) fields.

We note that this protocol is sufficiently simple that some of the operations we illustrate appear trivial. However, the key point is that exactly the same operations also work for much more complex protocols (such as Windows SMB/CIFS). Figure 4.5 shows two SPD dialogs, each consisting of a request (left) and response (right). For RolePlayer to process these, we must inform it of the

embedded hostnames ("host01", "hosttwo") and arguments ("`printer`", "`map`"), though we do not need to specify their locations in the protocol. If we did not specify the arguments, they would instead be identified and treated as *don't-care* fields. We do not need to inform RolePlayer of the hostname of a live peer because RolePlayer can automatically find it as the byte subsequence aligned with the hostname in the primary dialog if it appears in an ADU from the live peer. In addition, we do *not* inform RolePlayer of the embedded transaction identifiers (`1314`, `0300`), length fields, IP addresses (10.2.3.4, 10.4.5.6), or port numbers ($567 = \texttt{2} \cdot 256 + \texttt{55}$, $654 = \texttt{2} \cdot 256 + \texttt{142}$).

The naive way to search for dynamic fields would be to align the byte sequences of the corresponding ADUs and look for subsequences that differ. However, we need to treat endpoint-address and argument fields as a whole; for example, we do *not* want to decide that one difference between the primary and secondary dialogs is changing "01" in the first to "two" in the second (the tail end of the hostnames). Similarly, we want to detect that **13** and **14** in the replies are length fields and not elements of the embedded IP addresses that changed. To do so, we proceed as follows:

1. Search for endpoint-address and argument fields in both dialogs by finding matches of *presentations* of their known values. For example, we will find "host01" as an endpoint-address field and "printer" as an argument field in the primary's request.

   We consider seven possible presentations and their Unicode [94] equivalents for endpoint addresses, and one presentation and its Unicode equivalent for arguments. For example, for the primary's reply we know from the packet headers in the primary trace that the server's IP address is 10.2.3.4, in which case we search for: the binary equivalent (0x0A020304); ASCII dotted-quad notation ("10.2.3.4"); and comma-separated octets ("10,2,3,4"). The latter locates the occurrence of the address in the reply. (If the server's address was something different, then we would *not* replace "10,2,3,4" in the replayed dialog.)

2. If we have a secondary dialog, then RolePlayer splits each ADU into segments based on the endpoint-address and argument fields found in the previous step.

3. Finally, RolePlayer searches for length, cookie, and *don't-care* fields by aligning and comparing each pair of data segments. By "alignment" here we mean application of the Needleman-Wunsch algorithm [60], which efficiently finds the minimal set of difference between two

byte sequences subject to constraints; see Section 4.3.3 below for discussion. An important point is that at this stage we do not distinguish between cookie fields and *don't-care* fields. Only during the actual subsequent live session will we see whether these fields are used in a manner consistent with cookies (which need to be altered during replay) or *don't-care*'s (which shouldn't). See Section 4.3.2 for the process by which we make this decision.

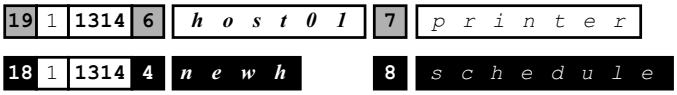
In the SPD example (Figure 4.5), aligning and comparing the five-byte initial segments in the primary and secondary requests results in the discovery of two pairs of length fields (`19` vs. `16`, and `6` vs. `7`) and one cookie field (`1314` vs. `0300`). To find these, RolePlayer first checks if a pair of differing bytes (or differing pairs of bytes) are consistent with being length fields, i.e., their numeric values differ by one of: (1) the length difference of the whole ADU, (2) the length difference of an endpoint-address or argument field that comes right after the length fields, or (3) the double-byte length difference of these, if the subsequent field is in Unicode format. For example, RolePlayer finds `19` and `16` as length fields because their difference matches the length difference of the request messages, while the difference between `6` and `7` matches the length difference of the client hostnames. (Note that, to accommodate Unicode, we merge any two consecutive differing byte sequences if there is only one single zero byte between them.)

4.3.2 Replay Stage

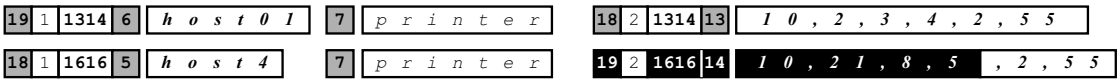
With the preparation complete, RolePlayer can communicate with a live peer (the remote application program that interacts with RolePlayer). To do so, it uses the primary application dialog plus the discovered dynamic fields as its "script," allowing it to replay either the initiator or the responder. Figures 4.6(a) and 4.6(b) give examples of creating an initial request and responding to a request from a live peer in SPD, respectively. To construct these requires several steps, as shown in Figure 4.7.

Deciding Packet Direction

We read the next data packet from the script to see whether we now expect to receive a packet from the live peer or send one out.



(a) The scripted (primary) dialog (top) and RolePlayer's generated dialog (bottom) for an SPD request for which we have instructed it to use a different hostname and service. The fields in black background reflect those updated to account for the modified request and the automatically updated length fields.



(b) The same for constructing an SPD reply to a live peer that sends a different transaction ID in their request, and for which the replayer is running on a different IP address. Note that the port in the reply stays constant.

Figure 4.6. Initiator-based and responder-based SPD replay dialogs.

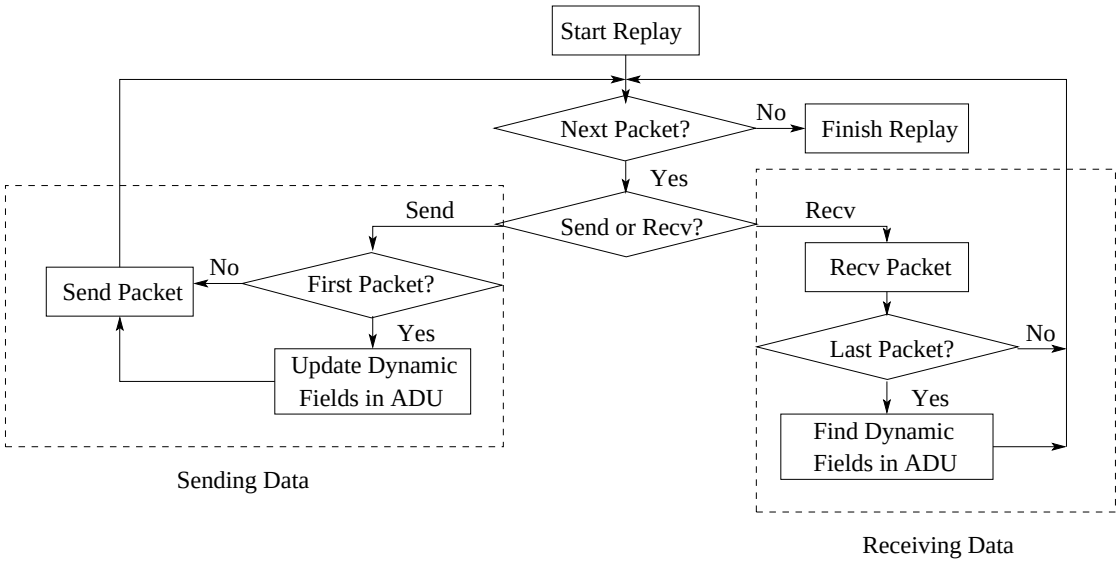


Figure 4.7. Steps in the replay stage.

Receiving Data

If we expect to receive a packet, we read data from the specific connection. Doing so requires deciding whether the received data is complete (i.e., equivalent to the corresponding data in the script). To do so, we use the alignment algorithm (Section 4.3.3) with the constraint that the match should be weighted to begin at the same point (i.e., at the beginning of the received data). If it yields a match with no trailing "gap" (i.e., it did not need to pad the received data to construct a good match), then we consider that we have received the expected data. Otherwise, we wait for more data to arrive.

After receiving a complete ADU, we compare it with the corresponding one in the script to locate dynamic fields. This additional search is necessary for two reasons. First, RolePlayer may need to find new values of endpoint addresses or arguments. Second, cookie fields found in the replay stage may differ from those found in the preparation stage due to accidental agreement between the primary and secondary dialogs.

We can apply the techniques used in the preparation stage to find dynamic fields, but with the major additional challenge that now for the live dialog (the ongoing dialog with the live peer) we do *not* have access to the endpoint addresses and application arguments. While the script provides us with guidance as to the existence of these fields, they are often not in easily located positions but instead surrounded by *don't-care* fields. The difficult task is to pinpoint the fields that need to change in the midst of those that do not matter. If by mistake this process overlaps an endpoint-address or argument field with a *don't-care* field, then this will likely substitute incorrect text for the replay. However, we can overcome these difficulties by applying the alignment algorithm with pairwise constraints (Section 4.3.3). Doing so, we find that RolePlayer correctly identifies the fields with high reliability.

After finding the endpoint-address and argument fields, we then use the same approach as for the preparation stage to find length, cookie and *don't-care* fields. We save the data corresponding to newly found endpoint-address, argument, and cookie fields for future use in updating dynamic fields.

Sending Data

When the script calls for us to send a data packet, we check whether it is the first one in an ADU. If so, we update any dynamic fields in it and packetize it. The updated values come from three sources: (1) analysis of previous ADUs; (2) the IP address and transport port numbers on the replayer; (3) user-specified (for argument fields). After updating all other fields, we then adjust length fields.

This still leaves the cookie fields for updating. RolePlayer only updates cookie fields *reactively*, i.e., to reflect changes first introduced by the live peer. So, for example, in Figure 4.6(a) we do not change the transaction ID when sending out the request, but we do change it in the reply shown in Figure 4.6(b).

To update cookie values altered by the live peer, we search the ADU we are currently constructing for matches to cookie fields we previously found by comparing received ADUs with the script. However, some of these identified cookie fields may in fact not be true cookies (and thus should not be reflected in our new ADU), for three reasons: some Windows applications use non-zero-byte padding; sometimes a single, long field becomes split into multiple, short cookie fields due to partial matches within it; and messages such as FTP server greetings can contain inconsistent (varying) data (e.g., the FTP software name and version).

Thus, another major challenge is to determine robustly which cookie matches we should actually update. We studied several popular protocols and found that cookie fields usually appear in the same context (i.e., with the same fields preceding and trailing them). Also, the probability of a false match to an N-byte cookie field is very small when N is large (e.g., when $N \geq 4$). Hence, to determine if we should update a cookie match, we check four conditions, requiring at least two to hold.

1. *Does the byte sequence have at least four bytes?* This condition reflects the fact that padding fields are usually less than four bytes in length because they are used to align an ADU to a 32-bit boundary.

2. *Does the byte sequence overlap a potential cookie field found in the preparation stage?* (If there is no secondary dialog, this condition will always be false, because we only find cookie fields during preparation by comparing the primary and secondary dialog.) The intuition here

is that the matched byte sequence is more likely to be a correct one since it is part of a potential cookie field.

3. *Is the prefix of the byte sequence consistent with that of the matched cookie field?* The prefix is the byte sequence between the matched cookie field and the preceding non-cookie dynamic field (or the beginning of the ADU). For prefixes exceeding 4 bytes, we consider only the last four bytes (next to the targeted byte sequence). For empty prefixes, if the non-cookie dynamic fields are the same type (or it is the beginning of the ADU) then the condition holds. This condition matches cookie fields with the same leading context.

4. *Is the suffix consistent?* The same as the prefix condition but for the trailing context.

In the SPD example (Figure 4.5), byte sequences of `1314` in the request and response messages in the primary dialog are cookie fields because they meet the second and fourth condition above. By the second condition, they both overlap `0300` in the secondary dialog. By the fourth condition, they are both trailed by a length field. On the other hand, the first and third conditions are not true because they have only two bytes and their prefix fields are `1` and `2`, respectively.

4.3.3 Design Issues

Sequence Alignment

The cornerstone of our approach is to compare two byte streams (either a primary dialog and a secondary dialog, or a script and the ADUs we receive from a live peer) to find the best description of their differences. The whole trick here is what constitutes "best". Because we strive for an application-independent approach, we cannot use the semantics of the underlying protocol to guide the matching process. Instead we turn to generic algorithms that compare two byte streams using customizable, byte-level weightings for determining the significance of differences between the two.

The term used for the application of these algorithms is "alignment", since the goal is to find which subsequences within two byte streams should be considered as counterparts. The Needleman-Wunsch algorithm [60] we use is parameterized in terms of weights reflecting the value associated with identical characters, differing characters, and missing characters ("gaps"). The algorithm then

uses dynamic programming to find an alignment between two byte streams (i.e., where to introduce, remove, or transform characters) with maximal weight.

We use two different forms of sequence alignment, global and semi-global. Global refers to matching two byte streams against one another in their entirety, and is done using the standard Needleman-Wunsch algorithm. Semi-global reflects matching one byte stream as a prefix or suffix of the other, for which we use a *modified* Needleman-Wunsch algorithm (see below).

When considering possible alignments, the algorithm assigns different weightings for each aligned pair of characters depending on whether the characters agree, disagree, or one is a gap. Let the weight be m if they agree, n if they disagree, and g if one is a gap. The total score for a possible alignment is then the sum of the corresponding weights. For example, given *abcdf* and *acef*, with weights $m = 2, n = -1, g = -2$, the optimal alignment (which the algorithm is guaranteed to find) is *abcdf* with *a-cef*, with score $m + g + m + n + m = 3$, where "-" indicates a gap.

To compute semi-global alignments of matching one byte stream as prefix of the other, we modify the algorithm to ignore trailing gap penalties. For example, given two strings *ab* and *abcb*, we obtain the same global alignment score of $2m + 2g$ for the alignments *ab--* with *abcb*, versus *a--b* with *abcb*. But for semi-global alignment, the similarity score is $2m$ for the first and $2m + 2g$ for the second, so we prefer the first since g takes a negative value.

The quality of sequence alignment depends critically on the particular parameters (weightings) we use. For our use, the major concern is deciding how many gaps to allow in order to gain a match. For example, when globally aligning *ab* with *bc*, two possible alignment results are *ab* with *bc* (score $n + n$) or *ab-* with *-bc* (score $g + m + g$). The three parameters will have a combined linear relationship (since we add combinations of them linearly to obtain a total score), so we proceed by fixing n and g (to 0 and -1, respectively), and adjusting m for different situations.

For global alignment—which we use to align two sequences before comparing them and locating length, cookie, and *don't-care* fields—we set $m = 1$ to avoid alignments like *ab-* with *-bc*. For semi-global alignment—used during replay to decide whether received data is complete—we set m to the length difference of the two sequences. The notion here is that if the last character of the received data matches the last one in the ADU from the script, then m is large enough to

offset the gap penalty caused by aligning the characters together. However, if the received data is indeed incomplete, its better match to only the first part of the ADU will still win out. Using the semi-global alignment example above, we will set $m = 2$ (due to the length of *ab* vs. *abcb*), and hence still find the best alignment with *abcb* to be *ab--* rather than *a--b*.

Finally, we make a powerful refinement to the Needleman-Wunsch algorithm for pinpointing endpoint-address and/or argument fields in a received ADU: we modify the algorithm to work with a *pairwise constraint matrix*. The matrix specifies whether the ith element of the first sequence can or cannot be aligned with the jth element of the second sequence. We dynamically generate this matrix based on the structure of the data from the primary dialog. For example, if the data includes an endpoint-address field represented as a dotted-quad IP address, then we add entries in the matrix prohibiting those digits from being matched with non-digits in the second data stream, and prohibiting the embedded "."s from being matched to anything other than "."s. This modification significantly improves the efficacy of the alignment algorithm in the presence of *don't-care* fields.

Removing Overlap

When we don't have a secondary dialog, and search for matches of cookie fields in an ADU in the primary dialog (so that we can update them with new values), sometimes we find fields that overlap with one another. We remove these using a greedy algorithm. For each overlapping dynamic field, we set the penalty for removing it as the number of bytes we would lose from the union set of all dynamic fields. (So, for example, a dynamic field that is fully embedded within another dynamic field has a penalty of 0.) We then select the overlapping field with the least penalty, remove it, and repeat the process until there is no overlap.

Handling Large ADUs

ADUs can be very large, such as when transferring a large data item using a single Windows CIFS, NFS or FTP message. RolePlayer cannot ignore these ADUs when searching for dynamic fields because there may exist dynamic fields embedded within them—generally at the beginning. For example, an NFS "READ Reply" response comes with both the read status and the corresponding file data, and includes a cookie field containing a transaction identifier. However, the complexity of sequence alignment is $O(MN)$ for sequences of lengths M and N, making its application in-

tractable for large sequences. RolePlayer avoids this problem by considering only fixed-size byte sequences at the beginning of large ADUs (we take the first 1024 bytes in our current implementation).

4.4 Evaluation

Protocol	Initiator Program	Responder Program	# Connections	# ADUs	# Initiator Fields		# Responder Fields	
					received	sent	received	sent
SMTP	manual	Sendmail	1	13	22	3	3	3
DNS	nslookup	BIND	1	2	8	0	0	1
HTTP	wget	Apache	1	2	10	0	0	0
TFTP	*W32.Blaster*	Windows XP	3	34	5	1	1	1
FTP	wget	ProFTPD	2	18	12	0	0	2
NFS	*mount*	Linux Fedora NFS	9	36	34	12	46	23
CIFS	*W32.Randex.D*	Windows XP	6	86	101	65	80	63

Table 4.2. Summary of evaluated applications and the number of dynamic fields in data received or sent by RolePlayer during either initiator or responder replay.

In the previous section, we described the design of RolePlayer. We implemented RolePlayer in 7,700 lines of C code under Linux, using libpcap [50] to store traffic and the standard socket API to generate traffic. Thus, RolePlayer only needs root access if it needs to send traffic from a privileged port. RolePlayer tries to packetize data of TCP connections in the same fashion as seen in the primary/secondary dialogs. However, the socket APIs do not support such functionality. In our implementation, RolePlayer calls send and sendto to dispatch data segments at the same boundary as seen in those dialogs, and usually achieves expected packetization. We tested the system on a variety of protocols widely used in malicious attacks and network applications. Table 4.2 summarizes our test suite. RolePlayer can successfully replay both the initiator and responder sides of all of these dialogs. In the remainder of this section, we first describe our test environment, then present the evaluation results, and discuss the limitations of RolePlayer in the end.

4.4.1 Test Environment

We conducted our evaluation in an isolated testbed consisting of a set of nodes running on VMware Workstation [95] interconnected using software based on Click [39]. We used VMware Workstation's support for multiple guest operating systems and private networks between VM in-

stances to construct different, contained test configurations, with the Click system redirecting malware scans to our chosen target systems. We gave each running VM instance a distinct IP address and hostname, and used non-persistent virtual disks to allow recovery from infection. For the tests we used both Windows XP Professional and Fedora Core 3 images, to verify that replay works for both Windows and Linux services. RolePlayer itself ran on the Linux host system rather than within a virtual machine, enabling it to communicate with any virtual machine on the system.

4.4.2 Simple Protocols

Our simplest tests were for SMTP, DNS, and HTTP replay. For testing SMTP, we replayed an email dialog with RolePlayer changing the recipient's email address (an "argument" dynamic field). In one instance, RolePlayer itself made this change as the session initiator; in the other, it played the role of the responder (SMTP server).

For DNS, RolePlayer correctly located the transaction ID embedded within requests, updating it when replaying the responder (DNS server). The HTTP dialog was similarly simple, being limited to a single request and response. Since the request did not contain a cookie field, replaying trivially consisted of purely resending the same data as seen originally (though RolePlayer found some *don't-care* fields in the HTTP header of the response message).

4.4.3 Blaster (TFTP)

As discussed in Section 4.2, Blaster [97] (see Figure 4.1) attacks its victim using a DCOM RPC vulnerability. After attacking, it causes the victim to initiate a TFTP transfer back to the attacker to download an executable, which it then instructs the victim to run. A Blaster attack session does not contain any length fields or hostnames, so we can replay it without needing a secondary application dialog or hostname information.

RolePlayer can replay both sides of the dialog. As an initiator, we successfully infected a remote host with Blaster. As a fake victim, we tricked a live copy of Blaster into going through the full set of infection steps when it probed a new system.

When replaying the initiator, RolePlayer found five dynamic fields in received ADUs. Of these,

it correctly deemed four as *don't-care* fields. These were each part of a confirmation message, specifying information such as data transfer size, time, and speed. The single dynamic field found and updated was the IP address of the initiator, necessary for correct operation of the injected TFTP command.

When replaying the responder, RolePlayer found only a single dynamic field, the worm's IP address, again necessary to correctly create the TFTP channel.

4.4.4 FTP

To test FTP replay (see Figure 4.3 for a sample dialog of PASV FTP), we used the wget utility as the client program to connect to two live FTP servers, *fedora.bu.edu* and *mirrors.xmission.com*. We collected two sample application dialogs using the command wget ftp://ftp-server-name/fedora/core/3/i386/os/Fedora/RPMS/crontabs-1.10-7.noarch.rpm. In both cases, wget used passive FTP, which uses dynamically created ports on the server side. When acting as the initiator, we replayed the *fedora.bu.edu* dialog over a live session to *mirrors.xmission.com*, and vice versa. There were no length fields, so we did not need a secondary dialog (nor hostnames). In both cases, RolePlayer successfully downloaded the file.

The system found twelve dynamic fields in the ADUs it received. Among them, the only two meaningful ones were endpoint-address fields: the server's IP address and the port of the FTP data-transfer channel. The rest arose due to differences in the greeting messages and authentication responses. RolePlayer recognized these as *don't-care*'s and did not update them.

When replaying the responder, wget successfully downloaded the file from a fake RolePlayer server pretending to be either *fedora.bu.edu* or *mirrors.xmission.com*, with the same two endpoint-address fields updated in ADUs sent by the replayer.

We tested support for argument fields by specifying the filename crontabs-1.10-7.noarch.rpm as an argument. When replaying the initiator, we replaced this with pyorbit-devel-2.0.1-1.i386.rpm, another file in the same directory. RolePlayer successfully downloaded the new file from *fedora.bu.edu* using the script from *mirrors.xmission.com*. Since the two files are completely different, the system found many *don't-care* fields. None of these affected the replay because they

did not meet the conditions for updating cookie fields. We also confirmed that RolePlayer can replay non-passive FTP dialogs successfully.

4.4.5 NFS

We tested the NFS protocol running over SunRPC using two different NFS servers. We used the series of commands mount, ls, cp, umount to mount an NFS directory, copy a file from it to a local directory, and unmount it. We used one NFS server for collecting the primary application dialog and a second as the target for replaying the initiator. The NFS session consisted of nine TCP connections, including interactions with three daemons running on the NSF server: portmap, nfs, and mountd. The first two ran on 111/tcp and 2049/tcp, while mountd used a dynamic service port. As is usually the case with NFS access, in the session both of the latter two ports were found via RPCs to portmap.

When replaying the initiator, RolePlayer found 34 dynamic fields in received ADUs, and changed 12 fields in the ADUs it sent. When replaying the responder, RolePlayer found 46 dynamic fields in received ADUs, and changed 23 fields in the ADUs it sent. The cookie fields concerned RPC call IDs.

RolePlayer successfully replayed both the initiator side (receiving the directory listing and then the requested file) and the responder side (sending these to a live client, which correctly displayed the listing and copied the file).

4.4.6 Randex (CIFS/SMB)

To test RolePlayer's ability to handle a complex protocol while interacting with live malware, we used the *W32.Randex.D* worm [98]. This worm scans the network for SMB/CIFS shares with weak administrator passwords. To do so, it makes numerous SMB RPC calls (see Figure 4.8, reproduced with permission from [63]). When it finds an open share, it uploads a malicious executable msmgri32.exe. In our experiments, we configured the targeted Windows VM to accept blank passwords, and turned off its firewall so it would accept traffic on ports 135/tcp, 139/tcp, 445/tcp, 137/udp, and 138/udp.

```
-> SMB Negotiate Protocol Request
<- SMB Negotiate Protocol Response
-> SMB Session Setup AndX Request
<- SMB Session Setup AndX Response
-> SMB Tree Connect AndX Request,
        Path:  \\XX.128.18.16\IPC$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request, Path: \samr
<- SMB NT Create AndX Response
-> DCERPC Bind: call_id: 1 UUID: SAMR
<- DCERPC Bind_ack:
-> SAMR Connect4 request
<- SAMR Connect4 reply
-> SAMR EnumDomains request
<- SAMR EnumDomains reply
-> SAMR LookupDomain request
<- SAMR LookupDomain reply
-> SAMR OpenDomain request
<- SAMR OpenDomain reply
-> SAMR EnumDomainUsers request

Now start another session, connect to the
SRVSVC pipe and issue NetRemoteTOD
(get remote Time of Day) request

-> SMB Negotiate Protocol Request
<- SMB Negotiate Protocol Response
-> SMB Session Setup AndX Request
<- SMB Session Setup AndX Response
-> SMB Tree Connect AndX Request,
        Path: \ \XX.128.18.16\IPC$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request, Path: \srvsvc
<- SMB NT Create AndX Response
-> DCERPC Bind: call_id: 1 UUID: SRVSVC
<- DCERPC Bind_ack: call_id: 1
-> SRVSVC NetrRemoteTOD request
<- SRVSVC NetrRemoteTOD reply
-> SMB Close request
<- SMB Close Response

Now connect to the ADMIN share and write the file

-> SMB Tree Connect AndX Request, Path: \\XX.128.18.16\ADMIN$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request,
    Path:\system32\msmsgri32.exe  <<<===

<- SMB NT Create AndX Response, FID: 0x74ca
-> SMB Transaction2 Request SET_FILE_INFORMATION
<- SMB Transaction2 Response SET_FILE_INFORMATION
-> SMB Transaction2 Request QUERY_FS_INFORMATION
<- SMB Transaction2 Response QUERY_FS_INFORMATION
-> SMB Write Request
....
```

Figure 4.8. The application-level conversation of W32.Randex.D.

To collect sample application dialogs, we manually launched a malware executable from another Windows VM, redirecting its scans to the targeted Windows VM, recording the traffic using tcpdump. We stored two attacks because replaying CIFS requires a secondary application dialog to locate the numerous length fields.

There are six connections in *W32.Randex.D*'s attack, all started by the initiator. Of these, two are connections to 139/tcp, which are reset by the initiator immediately after it receives a SYN-ACK from the responder. One connects to 80/tcp (HTTP), reset by the responder because the victim did not run an HTTP server. The remaining three connections are all to 445/tcp. The worm uses the first of these to detect a possible victim; it does not transmit any application data on this connection. The worm uses the second to enumerate the user account list on the responder via the SAMR named pipe. The final connection uploads the malicious executable to \Admin$\system32\msmsgri32.exe via the Admin named pipe.

When replaying the initiator, RolePlayer found 101 dynamic fields in received ADUs, and changed 65 fields in the ADUs it sent. When replaying the responder, it found 80 dynamic fields in received ADUs, and changed 63 fields in the ADUs it sent. (The difference in the number of fields is because some dynamic fields remain the same when they come from the replayer rather than the worm. For example, the responder chooses the context handle of the SAMR named pipe; when replaying the responder, the replayer just uses the same context handle as in the primary application dialog.) Considering ADUs in both directions, there were 21 endpoint-address fields, 76 length fields, and 32 cookie fields. The cookie fields reflect such information as the context handles in SAMR named pipes and the client process IDs.

As with our Blaster experiment, RolePlayer successfully infected a live Windows system with *W32.Randex.D* when replaying the initiator side, and, when replaying the responder, successfully drove a live, attacking instance of the worm through its full set of infection steps.

To demonstrate the function of RolePlayer, we select six consecutive messages from the conversation of W32.Randex.D (shown in bold-italic in Figure 4.8), consisting of SAMR Connect4 request/response, SAMR EnumDomains request/response, and SAMR LookupDomain request/response. We show the content of these messages from the primary, secondary, initiator-based

A->V	N1|**180**|S26|**0388**|S7|**96**|S20|**96**|S8|**113**|S16|R8|**96**|R6|**72**|R4|0014CCB8|**18**|M4|**18**|M4 |***144.165.114.119***|M20
A<-V	N4|S26|**0388**|S28|R24|M16|00000000**92F3E82470**FDD91195F8000C29**5763F7**|M4
A->V	N4|S26|**0388**|S56|R24|00000000**92F3E82470**FDD91195F8000C29**5763F7**|M8
A<-V	N1|**180**|S26|**0388**|S7|**124**|S8|**124**|S6|**125**|S1|R8|**124**|DECRPC-6|**100**|R4|M4|000B0BB0 |M4|000B27A0|M8|**8**|**10**|000B8D18|M8|000BC610|**5**|M4|**4**|***hone***|M36
A->V	N1|**156**|S26|**0388**|S7|**72**|S20|**72**|S8|89|S16|R8|**72**|R6|**48**|R4|M4 |00000000**92F3E82470**FDD91195F8000C29**5763F7**|**8**|**10**|001503F8|**5**|M4|**4**|***hone***
A<-V	N4|S26|**0388**|S28|R24|000B27A0|M32

(a) Primary Dialog

A->V	N1|**172**|S26|**0474**|S7|**88**|S20|**88**|S8|**105**|S16|R8|**88**|R6|**64**|R4|0014CCB8|**14**|M4|**14**|M4 |***48.196.8.48***|M20
A<-V	N4|S26|**0474**|S28|R24|M16|00000000**6093917586**FDD91195F8000C29**4A478F**|M4
A->V	N4|S26|**0474**|S56|R24|00000000**6093917586**FDD91195F8000C29**4A478F**|M8
A<-V	N1|**184**|S26|**0474**|S7|**128**|S8|**128**|S6|**129**|S1|R8|**128**|DECRPC-6|**104**|R4|M4|000B0BB0 |M4|000B6380|M8|**12**|**14**|000B76C0|M8|000C9FA8|**7**|M4|**6**|***host02***|M36
A->V	N1|**160**|S26|**0474**|S7|**76**|S20|**76**|S8|89|S16|R8|**76**|R6|**52**|R4|M4 |00000000**6093917586**FDD91195F8000C29**4A478F**|**12**|**14**|001503F8|**7**|M4|**6**|***host02***
A<-V	N4|S26|**0474**|S28|R24|000B27A0|M32

(b) Secondary Dialog

A->V	N1|**176**|S26|**0388**|S7|**92**|S20|**92**|S8|**109**|S16|R8|**92**|R6|**68**|R4|0014CCB8|**16**|M4|**16**|M4 |***192.168.170.3***|M20
A<-V	N4|S26|**0388**|S28|R24|M16|00000000**18B30AD10B**FDD91195F8000C29**3573E4**|M4
A->V	N4|S26|**0388**|S56|R24|00000000**18B30AD10B**FDD91195F8000C29**3573E4**|M8
A<-V	N1|**188**|S26|**0388**|S7|**132**|S8|**132**|S6|**133**|S1|R8|**132**|DECRPC-6|**108**|R4|M4|000B0BB0 |M4|000B9358|M8|**16**|**18**|000BEF40|M8|000B6BA0|**9**|M4|**8**|***hostpeer***|M36
A->V	N1|**164**|S26|**0388**|S7|**80**|S20|**80**|S8|89|S16|R8|**80**|R6|**56**|R4|M4 |00000000**18B30AD10B**FDD91195F8000C29**3573E4**|**16**|**18**|001503F8|**9**|M4|**8**|***hostpeer***
A<-V	N4|S26|**0388**|S28|R24|000BEF40|M32

(c) Initiator-based Replay Dialog

A->V	N1|**176**|S26|**0608**|S7|**92**|S20|**92**|S8|**109**|S16|R8|**92**|R6|**68**|R4|0014CCB8|**16**|M4|**16**|M4 |***169.91.250.93***|M20
A<-V	N4|S26|**0608**|S28|R24|M16|00000000**92F3E82470**FDD91195F8000C29**5763F7**|M4
A->V	N4|S26|**0608**|S56|R24|00000000**92F3E82470**FDD91195F8000C29**5763F7**|M8
A<-V	N1|**180**|S26|**0608**|S7|**124**|S8|**124**|S6|**125**|S1|R8|**124**|DECRPC-6|**100**|R4|M4|000B0BB0 |M4|000B27A0|M8|**8**|**10**|000B8D18|M8|000BC610|**5**|M4|**4**|***hone***|M36
A->V	N1|**156**|S26|**0608**|S7|**72**|S20|**72**|S8|89|S16|R8|**72**|R6|**48**|R4|M4 |00000000**92F3E82470**FDD91195F8000C29**5763F7**|**8**|**10**|00150A88|**5**|M4|**4**|***hone***
A<-V	N4|S26|**0608**|S28|R24|000B27A0|M32

(d) Responder-based Replay Dialog

Figure 4.9. A portion of the application-level conversation of W32.Randex.D. Endpoint-address fields are in bold-italic, cookie fields are in bold, and length fields are in gray background. In the initiator-based and responder-based replay dialogs, updated fields are in black background.

replay, and responder-based replay dialogs in Figure 4.9. To fit each message more compactly, we present them in the following format:

1. We split each message based on dynamic fields.

2. XY means we skipped Y bytes from protocol X (N for NetBIOS, S for SMB, R for DCE-RPC, M for Security Account Manager), since they do not change between different dialogs. These represent the fixed fields as part of the dialog.

3. We present endpoint-address fields such as IP addresses and hostnames in ASCII format in bold-italic. Three hostnames appear in the messages: "***hone***", "***host02***", and "***hostpeer***".

4. We show length fields in decimal, with a gray background. For example, "180" and "96" in the first message of the primary dialog are length fields.

5. We show cookie fields and *don't-care* fields, including the client process IDs and the SAMR context handles, in octets. For example, "0388" in the first message of the primary dialog is a client process ID. For convenience, we highlight in bold the cookies in the primary and secondary dialog which require dynamic updates during the replay process.

Note that RolePlayer located two cookie fields from the SAMR context handles because part of the handles didn't change between dialogs (e.g., the middle portion was constant in all the dialogs).

4.4.7 Discussion

From the experiments we can see that it is necessary to locate and update all dynamic fields—endpoint-address, cookie, and length fields—for replaying protocols successfully, while argument fields are important for extending RolePlayer's functionality. TFTP, FTP, and NFS require correct manipulation of endpoint-address fields. DNS, NFS, and CIFS also rely on correct identification of cookie files. CIFS has numerous length fields within a single application dialog. Leveraging argument fields, RolePlayer can work as a low-cost client for SMTP, FTP, or DNS.

While RolePlayer can replay a wide class of application protocols, its coverage is not universal. In particular, it cannot accommodate protocols with time-dependent state, nor those that use cryp-

tographic authentication or encrypted traffic, although we can envision dealing with the latter by introducing application-specific extensions to provide RolePlayer with access to a session's clear-text dialog. Another restriction is that the live peer with which RolePlayer engages must behave in a fashion consistent with the "script" used to configure RolePlayer. This requirement is more restrictive than simply that the live peer follows the application protocol: it must also follow the particular path present in the script.

Since RolePlayer keeps some dynamic fields unchanged as in the primary dialog, it is possible for an adversary to detect the existence of a running RolePlayer by checking if certain dynamic fields are changed among different sessions. For example, RolePlayer will always open the same port for the data channel when replaying the responder of an FTP dialog, and it will use the same context handles in SAMR named pipes when replaying the responder of a CIFS dialog. Another possible way to detect RolePlayer is to discover inconsistencies between the operating system the application should be running on versus the operating system RolePlayer is running on, by fingerprinting packet headers [118]. In the future, we plan to address these problems by randomizing certain dynamic fields and by manipulating packet headers to match the expected operating system.

Many applications of replay require automatic replay without human intervention. However, this causes a problem for our assumption that the domain names of the participating hosts in the example dialogs can be provided by the user. In fact, this assumption may be eliminated by clustering consecutive printable ASCII bytes in a message to find domain names. Note that we don't need to know the semantic meaning of the printable ASCII strings because in RolePlayer we use domain names synthetically for splitting a message into segments and identifying length fields.

To locate length and (sometimes) cookie fields, RolePlayer needs a secondary dialog. In some cases, it is not trivial to *automatically* identify two example dialogs that realize the same application session or malicious attack. In addition, RolePlayer as presented in this chapter can only handle one script at a time. In the next chapter, we will present the *replay proxy*, which extends the functionality of RolePlayer to process multiple scripts concurrently. We will also describe how we find two example dialogs for the same malicious attack.

4.5 Summary

We have presented RolePlayer, a system that, given examples of an application session, can mimic both the initiator and responder sides of the session for a wide variety of application protocols. We can potentially use such replay for recognizing malware variants, determining the range of system versions vulnerable to a given attack, testing defense mechanisms, and filtering multi-step attacks. However, while for some application protocols replay can be essentially trivial—just resend the same bytes as recorded for previously seen examples of the session—for other protocols replay can require correctly altering numerous fields embedded within the examples, such as IP addresses, hostnames, port numbers, transaction identifiers and other opaque cookies, as well as length fields that change as these values change.

We might therefore conclude that replay inevitably requires building into the replayer specific knowledge of the applications it can mimic. However, one of the key properties of RolePlayer is that it operates in an *application-independent* fashion: the system does not require any specifics about the particular application it mimics. It instead uses extensions of byte-stream alignment algorithms from bioinformatics to compare different instances of a session to determine which fields it must change to successfully replay one side of the session. To do so, it needs only two examples of the particular session; in some cases, a single example suffices.

RolePlayer's understanding of network protocols is very limited—just knowledge of a few low-level syntactic conventions, such as common representations of IP addresses and the use of length fields to specify the size of subsequent variable-length fields. (The only other information RolePlayer requires—depending on the application protocol—is context for the example sessions, such as the domain names of the participating hosts and specific arguments in requests or responses if we wish to change these when replaying.) This information suffices for RolePlayer to heuristically detect and adjust network addresses, ports, cookies, and length fields embedded within the session, including for sessions that span multiple, concurrent connections.

We have successfully used RolePlayer to replay both the initiator and responder sides for a variety of network applications, including: SMTP, DNS, HTTP; NFS, FTP and CIFS/SMB file transfers; and the multi-stage infection processes of the Blaster and W32.Randex.D worms. The

latter require correctly engaging in connections that, within a single session, encompass multiple protocols and both client and server roles.

Based on RolePlayer, we implemented a replay proxy for our honeyfarm system. This proxy can filter known multi-step attacks by replaying the server-side responses and allow new attacks through without dropping the ongoing ones by replaying the incomplete dialog to a high-fidelity honeypot. In the next chapter, we describe our honeyfarm system with the replay proxy in detail.

Chapter 5

GQ: Realizing a Large-Scale Honeyfarm System

In the previous chapter, we described RolePlayer, a system which, given one or two examples of an application session, can mimic both the client side and the server side of the session for a wide variety of application protocols. Based on RolePlayer, we develop a replay proxy to filter previously seen attacks in our honeyfarm system. In this chapter, we describe the design, implementation, and evaluation of our honeyfarm system.

5.1 Motivation

In Section 2.2, we reviewed the related work in the area of Internet epidemiology and defenses in detail. In this section, we briefly recall our discussion there to motivate our work.

Since 2001, Internet worm outbreaks have caused severe damage that affected tens of millions of individuals and hundreds of thousands of organizations. The Code Red worm [56], the first outbreak in today's global Internet after the Morris worm [20, 79] in 1988, compromised 360,000 vulnerable hosts with the Web server vulnerability and launched a distributed denial of service attack against a government web server from those hosts. The Blaster worm [97] was the first outbreak that exploited a vulnerability of a service running on millions of personal computers. The Slammer

worm [54] used only a single UDP packet for infection and took only 10 minutes to infect its vulnerable population. Also using a single UDP packet for infection, the Witty worm [55] exploited a vulnerability, which was publicized only one day before, in a commercial intrusion detection software, and was the first to carry a destructive payload of overwriting random disk blocks. Prior study [57, 83, 82] suggests that *automated early* detection of new worm outbreaks is essential for any effective defense.

Honeypots, as vulnerable network decoys, can be used to detect and analyze the presence, techniques, and motivations of an attacker [81]. By themselves, however, honeypots serve as poor "early warning" detection of new worm outbreaks because of the low probability that any particular honeypot will be infected early in a worm's growth. On the other hand, *network telescopes*, which work by monitoring traffic sent to unallocated portions of the IP address space, have been a powerful tool for understanding the behavior of Internet worm outbreaks at scale [58]. Combining them, a central tool that has emerged recently for detecting Internet worm outbreaks is the *honeyfarm* [80], a large collection of honeypots fed Internet traffic by a network telescope [6, 34, 58, 96]. The honeypots interact with remote sources whose probes strike the telescope, using either "low fidelity" mimicry to characterize the *intent* of the probe activity [6, 63, 66, 117], or with "high fidelity" full execution of incoming service requests in order to purposefully become infected as a means of assessing the malicious activity in detail [17, 29, 34, 96].

Considerable strides have been made in deploying low-fidelity honeyfarms at large scale [6, 117], but high-fidelity systems face extensive challenges, due to the much greater processing required by the honeypot systems, and the need to safely contain hostile, captured contagion while still allowing it enough freedom of execution to fully reveal its operation. Recent work has seen significant advances in some facets of building large-scale, high-interaction honeyfarms, such as making much more efficient use of Virtual Machine (VM) technology [96], but these systems have yet to see live operation *at scale*.

To achieve automated early detection of Internet worm outbreaks, we develop a large-scale, high-fidelity honeyfarm system named *GQ*. Our work combines clusters of high-fidelity honeypots with the large address spaces managed by network telescopes. By leveraging extensive filtering and engaging honeypots dynamically, we can use a small number of honeypots to inspect untrusted

traffic sent to a large number of IP addresses, with goals of automatically detecting Internet worm outbreaks within seconds, and facilitating a wide variety of analyses on detected worms, such as extracting (vulnerability, attack, and behavioral) signatures, and testing whether these signatures can correctly detect repeated attacks by deploying them within the controlled environment of the honeyfarm. Architecturally, this is similar in spirit to our UCSD colleagues' work on the *Potemkin* system, though that effort particularly emphasizes developing powerful VM technology for running large numbers of servers optimized for worm detection [96].

In the remainder of this chapter, we first describe GQ's design (see Section 5.2) and implementation (see Section 5.3). Then we report on our preliminary experiences with operating the system to monitor more than a quarter million addresses, during which we captured 66 distinct worms over the course of four months (see Section 5.4). Finally, we finish with a summary of our work (see Section 5.5).

5.2 Design

In this section we present GQ's design, beginning with an overview of the design goals and system's architecture. We then discuss management of the incoming traffic stream: obtaining input from the network telescope, prefiltering this stream with lightweight mechanisms, and further filtering it using a replay proxy. We follow this with how we address the thorny problem of containment, detailing the containment and redirection policy we have implemented that endeavors to strike a tenable balance between allowing contagion executing within the honeyfarm to exhibit its complete infection (and possibly self-propagation) process, while preventing it from damaging external hosts. We finish with a discussion of issues regarding the management of the honeypots.

5.2.1 Overview and Architecture

In developing our honeyfarm system, we aim for the system to achieve several important goals:

- **High Fidelity**: The system should run fully functional servers to detect new attacks against even unknown vulnerabilities.

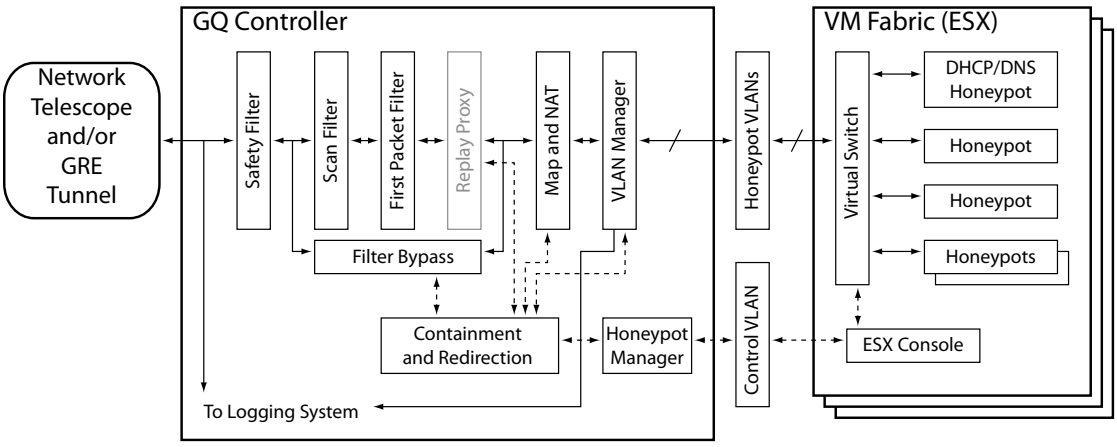


Figure 5.1. GQ's general architecture.

- **Scalability**: The system should be able to analyze in real-time the scanning probes seen on a large number of Internet addresses (e.g., a quarter million addresses).
- **Isolation**: The system should isolate the network communications of every single honeypot.
- **Stringent Control**: The system should prevent honeypots from attacking external hosts.
- **Wide Coverage**: The system should run honeypots with a wide range of configurations with respect to operating systems and services.

As Figure 5.1 portrays, GQ's design emphasizes high-level modularity, consisting of a front-end controller and back-end honeypots, mediated by a honeypot manager. This structure allows the bulk of our system to remain honeypot-independent: although we currently use VMware ESX server for our honeypot substrate, we intend to extend the system to include both Xen-based honeypots using the technology developed for Potemkin [96] and a set of "bare metal" honeypots that execute directly on hardware rather than in a VM environment, so we can detect malware that suppresses its behavior when operating inside a VM.

The front-end controller supports data collection from multiple sources, including both direct network connections and GRE tunnels. The latter allow us to topologically separate the honeyfarm from the network telescope, and potentially to scatter numerous "wormholes" around the Internet to obtain diverse sensor deployment. These features are desirable both for masking the honeyfarm's scope from attacker discovery and to ensure visibility into a broad spectrum of the Internet address

space, as previous work has shown large variations in the probing seen at different locations in the network [14]. The controller also performs extensive filtering, using both simple mechanisms and the more sophisticated replay proxy.

The front-end also performs Network Address Translation (NAT). Rather than require that each honeypot be dynamically reconfigured to have an internal address equivalent to the external telescope address for which it is responding, we use NAT to perform this rewriting. However, if the honeypot itself supports dynamic address reconfiguration (e.g., [96]), the front end simply skips the NAT.

The final function of the front-end is containment and redirection. When a honeypot makes an outgoing request, GQ must decide whether to allow it out to the public Internet. Based on the policy in place, the front-end either allows the connection to proceed, redirects it to another honeypot, or blocks it. Redirection to another honeypot is the means by which we can directly detect a worm's self-propagating behavior (namely if, once contacted, the new target honeypot itself then initiates network activity). However, many types of malware require multi-stage infection processes for which the initial infection will fail to fully establish (and thus fail to exhibit subsequent self-propagation) if we do not allow some preliminary connections out to the Internet. Finding the right balance between these two takes careful consideration, as we discuss below.

5.2.2 Network Input

The initial input to GQ is a stream of network connection attempts. GQ can process two different forms of input: a directly routed address range, and GRE-encapsulated tunnels. A direct-routed range simply is a local Ethernet or other network device connected to one or more subnets, while a GRE tunnel requires a remote peer.

Supporting direct connections gives a simple and efficient deployment option. Supporting GRE gives the opportunity to decouple the honeyfarm system from the telescope's address space, and can ultimately serve as a bridge between multiple honeyfarms.

5.2.3 Filtering

We term the initial stage of responding as *prefiltering*, which aims to use a few simple rules to eliminate probe sources likely of low interest. Prefiltering has several components. First, GQ limits the total number of distinct telescope addresses that engage a given remote source to a small number N. The assumption here is that a modest N suffices to identify a source engaged in a new or otherwise interesting form of activity. This filter can greatly reduce the volume of traffic processed by the honeyfarm because of the very common phenomenon of a single remote source engaging in heavy scanning of the telescope's address space.

We note that there is no optimal value for N. Too low, and we will miss processing sources that engage in diverse scanning patterns that send different types of probes to different subsequent addresses. Too high, and we burden the honeypots with a great deal of redundant traffic.

In principle, knowledge of N also allows an adversary to design their probes to escape detection by the honeyfarm. However, to do so the adversary also needs to know the address space over which we apply the threshold of N, and if they know that, they can more directly escape detection.

As a way to offset both these considerations, GQ's architecture supports *sampling* of the probe stream: taking a small, randomly selected subset of the stream and accepting it for processing without applying any of our filtering criteria (see "Filter Bypass" in Figure 5.1). We also use this mechanism for sources deemed "interesting" during the filtering stages, such as those that deviate from the replay proxy's scripts (see Section 5.2.4).

Our current implementation of this prefilter maintains state for each IP address that contacts our address ranges. Since we require $8 * N$ bytes for each IP address, we need only 400M bytes of memory for 10 million hosts when N is taken to be five. To date, we have not seen significant memory issues with keeping complete state.

GQ uses a second prefilter, too, taken from the work of Bailey *et al.* [6]. After acknowledging an initial SYN packet, we examine the first data packet sent by the putative attacking host. If this packet unambiguously identifies the attack as a known and classified threat, the filter drops the connection. While configuration of these filters requires manual assessment (since many attacks

cannot be unambiguously identified by their first data packet), we can use this mechanism to weed out background radiation from endemic sources such as Code Red and Slammer.

Note that our use of TCP header rewriting when instantiating our honeypots greatly simplifies use of this second prefilter. Without it, we would have to allocate a honeypot upon the initial SYN (if it passes the scan-N prefilter) so it could complete the SYN handshake, agree upon sequence numbers, and elicit the first data packet, just in case that packet proved of interest and we wished to continue the connection. We could not proxy for the beginning of the connection because any honeypot we might wind up instantiating later to handle the remainder of the connection would not have knowledge of the correct sequence numbers.

Additionally, we include a final "safety" filter, used only for filtering outbound traffic. Its role is to limit external traffic from the honeyfarm if any other component fails. This is a more relaxed but simpler policy than we implement in the rest of the honeyfarm, as the point is to assure—via independent operation—that any failure of our more complicated containment mechanism will not have a deleterious external effect.

5.2.4 Replay Proxy

Something quite important to realize (which we quantify in Section 5.4) is that the simple prefilters discussed in the previous section lack the power to identify many instances of known attacks. As an extreme example, the *W32.Femot* worm goes through *90 pairs* of message exchanges before we can distinguish one variant from another, because it is only at that point that the attack reveals the precise code it injects into the victim. Even for well-known worms such as Blaster, the first-packet filter will fail to recognize new and potentially interesting variants, because these will not manifest until later. Yet creating a custom filter for each significant piece of known worms is quite impractical due to their prevalence.

To filter these frequent multi-stage attacks, we employ technology developed for RolePlayer, our application-independent replay system discussed in Chapter 4. RolePlayer uses a *script* automatically derived from one or two examples of a previous application session as the basis for mimicking either the client or the server side of the session in a subsequent instance of the same

transaction. The idea is that we can represent each previously seen attack using a script that describes the network-level dialog (both client and server messages) corresponding to the attack. A key property of such scripts is that RolePlayer can automatically extract them from samples of two dialogs corresponding to a given attack, *without* having to code into RolePlayer any knowledge of the semantics of the application protocol used in the attack. We leverage this feature of RolePlayer to build up a set of scripts with which we configure our *replay proxy*.

The replay proxy extends RolePlayer's functionality in three substantial ways:

- Processing multiple scripts concurrently;
- Recognizing when a source deviates from this set of scripts;
- Serving as an intermediary between a source and a high-fidelity honeypot once the script no longer applies.

The proxy first plays the role of the server side of the dialog, replying to incoming traffic from a remote source. By matching the source's messages against multiple scripts, the proxy can perform a fine-grained classification of known attacks without relying on application-specific responders. If the source never deviates from the set of scripts, then we have previously seen its activity, and we can drop the connection.

If the dialog deviates from the scripts, however, the proxy has found something that is in some fashion new. At this point it needs to facilitate cutting the remote source over from the faux dialog with which it has engaged so far to a continuation of that dialog with a high-fidelity honeypot. Doing so requires bringing the honeypot "up to speed." That is, as far as the source is concerned, it is already deeply engaged with a server. We need to synch the server up to this same point in the dialog so that the rest of the interaction can continue with full fidelity.

To do so, the proxy replays the dialog as it has received it so far back to the honeypot server; this time, the proxy plays the role of the client rather than the server's role. Once it has brought the honeypot up to speed, the source can then continue interacting with the honeypot. However, the replay proxy needs to remain engaged in the dialog as an intermediary, translating the IP addresses,

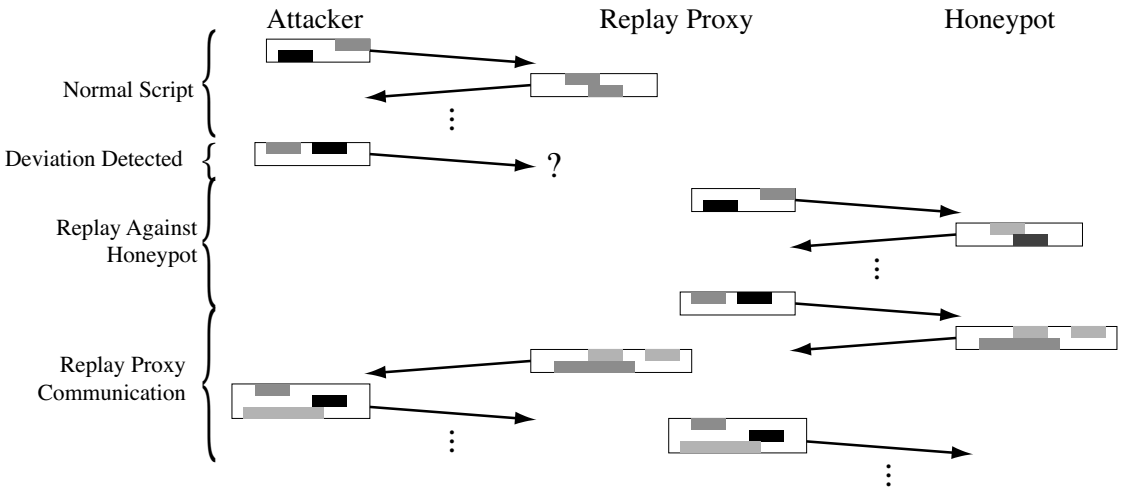


Figure 5.2. The replay proxy's operation. It begins by matching the attacker's messages against a script. When the attacker deviates from the script, the proxy turns around and uses the client side of the dialog so far to bring a honeypot server up to speed, before proxying (and possibly modifying) the communication between the honeypot and attacker. The shaded rectangles show in abstract terms portions of messages that the proxy identifies as requiring either echoing in subsequent traffic (e.g., a transaction identifier) or per-connection customization (e.g., an embedded IP address).

sequence numbers, and, most importantly, dynamic application fields (e.g., IP addresses embedded in application messages, or opaque "cookie" fields). Figure 5.2 shows the replay proxy's operation.

For new attacks, there exists a risk that the replay proxy might interfere with their execution because its knowledge is limited to scripts generated from previous attacks. Since the proxy has no semantic understanding of the communication relayed between the attacker and the honeypot, it cannot recognize if such interference occurs. Thus, the proxy marks the source of each attack it attempts to pass along to a backend honeypot as "interesting", such that any further connections from that source will be routed directly to a honeypot server, bypassing the replay proxy.

The replay proxy is one of GQ's most significant components. We will discuss some implementation considerations for it in Section 5.3.1.

5.2.5 Containment and Redirection

Another important design element concerns monitoring and enforcing the *containment policy*: i.e., as compromised honeypots initiate outbound network connections, do we allow these to traverse into the public Internet, redirect them to a fake respondent running inside the honeyfarm, or block

them entirely? As noted above, the fundamental tension here is the need to balance between allowing contagion to exhibit its complete infection process (necessary for identifying self-propagation), but preventing the contagion from damaging external hosts.

To do so, we work through the following policy, in the given order:

1. If an outbound request would exceed the globally enforced, per-instantiated-honeypot rate limit, drop it. This acts as a general safety feature for containment, complementary to, but independent of, the "safety filter" previously discussed.

2. If an outbound DNS request originates from the DHCP/DNS-server honeypot, allow it. We configure GQ's honeypots with the address of a DHCP/DNS server that itself runs on a GQ honeypot. We allow that one node to make outbound DNS requests.

3. If an outbound DNS request originates from a regular honeypot, redirect it to the DHCP/DNS honeypot. Additionally, we consider such requests as suspicious, since the honeypot should have originally sent them to the DHCP/DNS honeypot according to the configuration.

4. If an outbound request originates from a honeypot that has not been probed, drop it. If a honeypot creates spontaneous activity (such as automatic update requests or similar tasks), we simply block this activity. This serves both as a safety net and to ensure we know the exact behavior of the honeypot.

5. If an outbound request targets the address of the source that caused GQ to instantiate the honeypot, allow it. We allow the honeypot to always communicate with the source of the initial attack, to enable it to complete some forms of multi-stage infections and to participate in simple "phone home" confirmations.

6. If an outbound request is the first outbound connection to a host *other* than the infecting source, allow it. Many worms use a two-stage infection process, where an initial infection then fetches the bulk of the contagion from a second remote site, or "phones home" to register the successful infection.[1]

[1] For example, we received notification that three of our telescope addresses appeared in a list produced upon the arrest of the suspected operators of the *toxbot* botnet [99]. Unlike many erroneous notifications we receive due to our use of the address space, this one was correct: our honeypots had indeed become infected by *toxbot*, and our containment policy correctly allowed them to contact the botnet's control infrastructure.

7. If the destination port is 21/tcp, 80/tcp, 443/tcp, or 69/udp AND the port was not the one accessed when the original remote source probed the honeypot AND the outbound request is below a configurable threshold of how many such extra-port requests to allow, allow it. Again, we do this to enable multi-stage infections and their control traffic, while attempting to limit the probability of an escaped infection.

8. If the outbound request is a DHCP address-request broadcast, redirect it to the DHCP/DNS honeypot.

9. If a clean honeypot is available, redirect the outbound request to it. Note that this step is critical for detecting self-propagating behavior, and highlights the importance of not running the honeyfarm at full capacity.

10. Otherwise, drop the outbound request.

In addition, when this process yields a decision to redirect, GQ consults a configurable *redirection limit* (computed on a per-instantiated-honeypot basis). If this connection would cause the honeypot to exceed the limit, we drop the connection instead, to prevent runaway honeypots from consuming excessive honeyfarm resources.

It is while assessing containment/redirection that GQ also makes a determination, from a network perspective, that we have spotted a new worm. We consider a system as infected with a worm if, when redirected to another honeypot, it causes *that* honeypot to begin initiating connections. Thus, from a network perspective, a worm is a program which, when running on one honeypot, can cause other honeypots to change state sufficiently that they also begin generating connection requests. This definition gives us a behavioral way to separate an infection that causes non-self-propagating network activity from one that self-propagates at least one additional step (thus distinguishing worms from botnet "autorooters" that compromise a machine and perhaps download configurable code to it, but do not cause it to automatically continue the process).

Redirection within the honeyfarm also allows us to perform malware classification. As the worm continues to generate requests, we redirect these to honeypots with differing configurations, including patch levels and OS variants. Doing so automatically creates a system *vulnerability profile*, determining which systems and configurations the worm can exploit.

The policy engine also manages the mapping of external (telescope) addresses to honeypots. Currently, when doing so it allocates different honeypots to distinct attackers, even if those attackers happen to target the same telescope address. Doing so maintains GQ's analysis of exactly what remote traffic infected a given honeypot, but also makes the honeyfarm vulnerable to fingerprinting by attackers.

A final facet of containment concerns internal containment. We need to ensure that infected honeypots cannot directly contact any other honeypot in the honeyfarm without our control system mediating the interaction. To do so, we require that all honeypot-to-honeypot communication pass through the control system, which also allows us to maintain each honeypot behind a NAT device. Note that the load it adds to the control system is trivial compared with the scan probes from the network telescopes.

We use VLANs as the primary technology for internal containment. This imposes a requirement that the back-end system running the honeypot VM can terminate a VLAN. We assign each honeypot to its own VLAN, with the GQ controller on a VLAN trunk. The controller can then rewrite packets on a per-VLAN as well as per-IP-address basis, enabling fine-grained isolation and control of all honeypot traffic, even broadcast and related traffic.

5.2.6 Honeypot Management

To maintain modularity, the GQ honeypot manager (see Figure 5.1) isolates management of honeypot configurations. The controller keeps an inventory and state of the different available honeypots, but defers direct operation of the VM honeypots to the honeypot manager, which is responsible for starting and reseting each honeypot. Only the honeypot manager needs to understand the VM or bare-metal technology used to implement a given honeypot. By doing this, we not only avoid the risk of inconsistency between the controller and the (stateless) honeypot manager, but also make the controller independent of the back-end implementation.

In the large, the GQ controller simply monitors the honeypots and responds to network requests. When the manager restarts a honeypot, the honeypot exists in a newly born state until requesting its IP address through DHCP. After the honeypot received an address assignment, the controller

waits until it sees the system generate some network traffic (which it inevitably does as part of its startup). At this point the controller waits ten seconds (to enable remaining services to start up and to avoid startup transients) before considering the honeypot available. Once available, the controller can now allocate the honeypot to serve incoming traffic. When later the controller decides that the instantiated honeypot no longer has value, it instructs the manager to destroy the instance and restart the honeypot in a reinitialized state.

5.3 Implementation

The GQ system is implemented in C (the replay proxy) and C++ (the Click modules) for about 14,000 lines in total. In this section we describe the most significant issues that arose when realizing the GQ system as an implementation of the architecture presented in the previous section. These issues concerned the replay proxy system, the allocation and isolation mechanisms, and the VMware virtual machine management, which we detail in this section.

5.3.1 Replay Proxy

The replay proxy plays a central role in our architecture. Our current system takes a series of scripts for different attacks, with two sample dialogs (a "primary" dialog and a "secondary" dialog, in the terminology of Section 4.2.3, necessary for locating some types of dynamic fields) for each script. The main challenges for implementing the replay proxy are generating scripts for newly seen attacks and determining when a host has deviated from a script.

To generate a script for a newly seen worm attack, we take two different approaches depending on whether the attack succeeded or failed. For successful attacks, we can use the instances of infections redirected inside the honeyfarm for the primary and secondary dialogs. In this case, we also have control over replay-sensitive fields such as host names and IP addresses, which RolePlayer needs to know about to construct an accurate script.

For unsuccessful attacks, however, we have only one example of the application dialog corresponding to the attack. In this case, we face the difficult challenge of trying—without knowledge

of the application's semantics—to find another instance of the exact same attack (as opposed to merely a variant or an attack that has some elements in common). We leverage a heuristic to tackle this problem: we assume that two very similar attacks from the same attacking host are in fact very likely the same attack, and we can therefore use the pair as the primary and secondary dialogs required by RolePlayer. We test whether two candidate attacks from the same host are indeed "very similar" by checking the number of Application Data Units (ADUs) in the two dialogs, their sizes, and the dynamic fields we locate within them. We require that the number of ADUs must be the same; the difference in ADU size quite small; and dynamic fields consistent across the two dialogs.

We use this technique to construct a corpus of known types of activity automatically. After a single infection by a worm, we want to inform automatically the replay proxy of the worm's attack to allow it to filter out subsequent probes from the same worm (which could become huge very rapidly, if the worm spreads quickly), to avoid tying up the honeyfarm with uninteresting additional worm traffic. We also benefit significantly from having scripts to filter out uninteresting instances of "background radiation".

To efficiently filter out known activity, rather than comparing an instance with all of the scripts at each step, we merge the scripts into a tree structure. For N scripts, doing so reduces the computational complexity from $O(N)$ to $O(\log(N))$. Constructing the tree requires comparing ADUs from different scripts. We decide that two ADUs from different scripts are the same if the match of dynamic fields between them is at least as good as that between the primary and secondary dialogs used to locate the fields in each script in the first place.

In online filtering, we use the same approach to check if a received ADU matches the ADU of a node in the tree. If not, it represents a *deviation* from our collection of scripts.

When we compare an ADU with a set of sibling nodes, it is possible that more than one of the nodes matches. When this happens, we choose the node that minimizes the number of *don't-care* fields (opaque data segments that appear in only one side of the attacks; see Chapter 4 for more details) identified in the compared node.

To avoid comparing a received ADU with every node in a large set of sibling nodes (which can arise for some common attack vectors like CIFS/SMB exploits), we use a Most Recently Used

(MRU) technique to control the number of nodes compared. The intuition behind this technique is to leverage the locality of incoming probes. Given a set of sibling nodes, we associate with each node a time reflecting when it last matched a a received ADU. We compare newly received ADUs with those nodes having the most recent times first, to see if we can quickly find a close match.

The replay proxy also supports detection of attack variants for attacks that involve multiple connections. For example, if a new Blaster worm uses the same buffer overflow code as the old one, but changes the contents of the executable file then transferred over TFTP, the replay proxy can correctly follow the dialog up through the TFTP connection, retrieve the executable file, notice it differs from the script at that point, and proceed to replay this full set of activity to a honeypot server in order to bring it up to speed to become infected by the attack variant.

A final detail is that when receiving data the proxy must determine when it has received a complete ADU. Since it may interact with unknown attacks, it cannot always directly tell when this occurs, so we use a timeout (currently set to five seconds) to ensure it reaches a decision. If the source has not sent any further packets when the timeout expires, we assume the data received until that point comprises an ADU. Since the timeout adds extra latency for detecting a new worm, we plan to improve the design of the replay proxy to eliminate it by comparing incomplete ADUs with the scripts.

Our replay proxy implementation comprises 8,600 lines of C code. It currently uses the socket APIs to communicate with the attackers and the honeypots. This limits its performance and makes it unwieldy to implement the address and sequence number translation it needs to perform. We plan to replace the socket APIs with a user-level transport layer stack (using raw sockets) and port the replay proxy to Click. Doing so will let us closely integrate the replay proxy with the other components of the honeyfarm controller.

Currently, a single-threaded instance of the replay proxy can process about 100 concurrent probes per second. To use the proxy operationally, we must improve its performance so that it can process significantly more concurrent connections. In our near-term future work, we aim to solve this problem from three perspectives: (1) improve the replay algorithm to reduce the computational cost; (2) leverage multiple threads/processes/machines; (3) convert its network input/output to use

raw sockets. Our general goal is for the overhead of the replay proxy processing for an incoming probe to be much less than that for launching a virtual machine honeypot.

5.3.2 Isolation

Another critical component for our system implementation is maintaining rigorous isolation of all honeypots. The honeypots need to contact other honeyfarm systems—in particular, DHCP and DNS servers—but we need absolute control over these and any other interactions. Additionally, we need to keep our control traffic isolated from honeypot traffic.

To do so, we use VLANs and a VLAN-aware HP ProCurve 2800 series managed Ethernet switch [27]. We use a "base" VLAN with no encapsulation for a port on the controller and a port on each VMware ESX server. This LAN carries the management traffic for all the ESX servers, including access to our file server. By using an entirely separate (virtual) network with a switch capable of enforcing the separation, we can prevent even misbehaving honeypots from saturating our control plane.[2]

Equally important is maintaining isolation among honeypot traffic. Each individual honeypot should only communicate with authorized systems, which we accomplish using VLAN tagging and the Virtual Switch Tagging (VST) mode provided by the VMware ESX server. In this mode, each virtual switch in the ESX server will tag (untag) all outbound (inbound) Ethernet frames, leaving VLAN tagging transparent to the honeypots. This is important because we want to repeatedly use a single virtual machine image file for multiple running instances, and thus we want to avoid embedding any VLAN-related state in the image file. With this approach, the only untagged link is between the honeypot and the virtual switch; we achieve isolation for this segment by dedicating a port group to each honeypot (the port group number is the same as the VLAN identifier assigned to the honeypot). Thus, the GQ controller's interface to the honeypot network receives VLAN-tagged packets, with each honeypot on a different VLAN.

We also need to consider VLAN tagging/untagging for the controller. We implemented a VLAN manager in the controller, which maintains the mapping between IP addresses and VLAN IDs, and

[2]We are susceptible to bugs in the switch, however, if the VLAN encapsulation fails. We could instead use two separate switches if this proves problematic.

adds/removes VLAN tags before the Ethernet frames leave/enter the GQ controller. This approach allows us to completely abstract the isolation mechanism from the rest of the system.

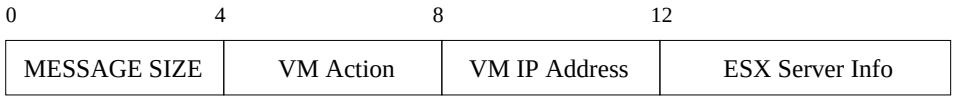
Additionally, the VLAN manager has a "default route" for forwarding DHCP packets to the dedicated DHCP/DNS VM system. Rather than running these servers on the controller, where they might be susceptible to attack by a worm, we use a separate Linux honeypot to provide DHCP and DNS service for the honeypots.

5.3.3 Honeypot Management

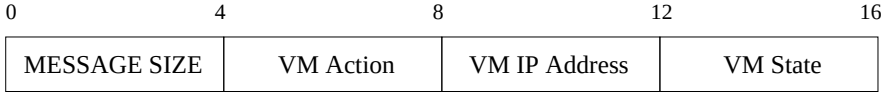
Table 5.1. Virtual machine image types installed in GQ.

Windows 2000 Professional with No Patches	Windows 2000 Professional with Service Pack 4
Windows 2000 Professional with All Patches	Windows 2000 Server with No Patches
Windows 2000 Server with Service Pack 4	Windows 2000 Server with All Patches
Windows XP Professional with No Patches	Windows XP Professional with Service Pack 2
Windows XP Professional with All Patches	Fedora Core 4 with All Patches

We have manually installed ten different virtual machine images for both Windows and Linux (see Table 5.1). For ease of initial deployment, we operate only a subset of these which together cover a wide range of vulnerabilities in Windows systems.



Request from GQ Controller to Honeypot Manager



Response from Honeypot Manager to GQ Controller

Figure 5.3. Message formats for the communication between the GQ controller and the honeypot manager. "ESX Server Info" is a string with the ESX server hostname, the ESX server control port number, the user name, the password, and the VM config file name, separated by tab keys.

We implemented the honeypot manager in C using the (undocumented other than by header files) vmcontrol interface provided by VMware for remotely controlling virtual machines running on VMware ESX servers. The communication between the honeyfarm controller and the honeypot manager follows a simple application-level protocol (see Figure 5.3) by which the controller instructs the manager what actions to take on what virtual machines, and the manager informs the controller of the results of those actions.

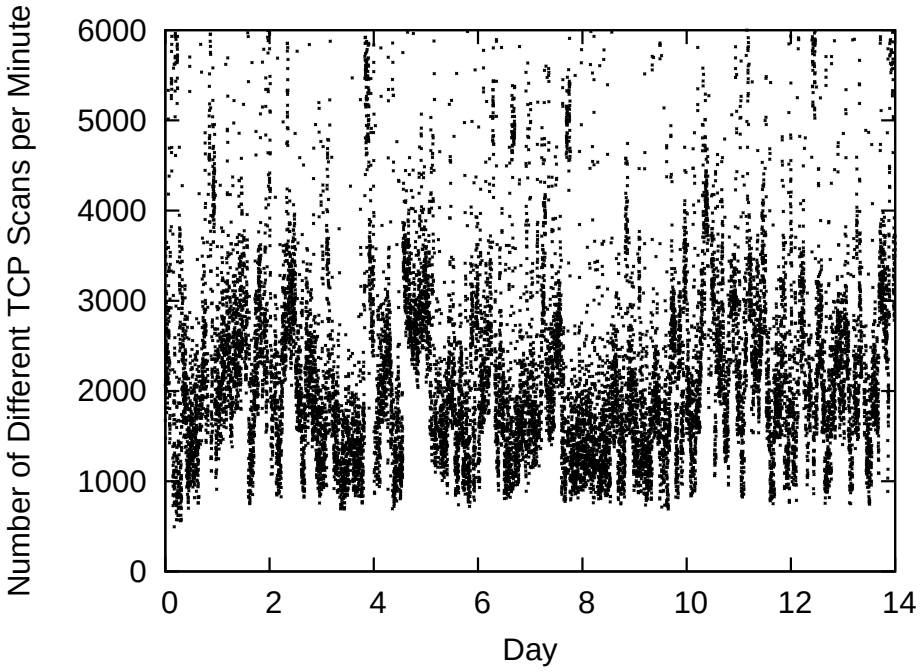
5.4 Evaluation

We began running GQ operationally in late 2005. The initial deployment we operate consists of four VMware ESX servers running 24 honeypots in three different virtual machine configurations, including unpatched Windows XP Professional, unpatched Windows 2000 Server, and a fully patched Windows XP Professional with an insecure configuration and weak passwords, since these three configurations have a good coverage of vulnerabilities (vulnerabilities in unpatched Windows 2000 Professional are a subset of those in unpatched Windows 2000 Server). Thus, GQ can currently capture worms that exploit Windows vulnerabilities on 80/tcp, 135/tcp, 139/tcp, and 445/tcp, but not other types of worms.

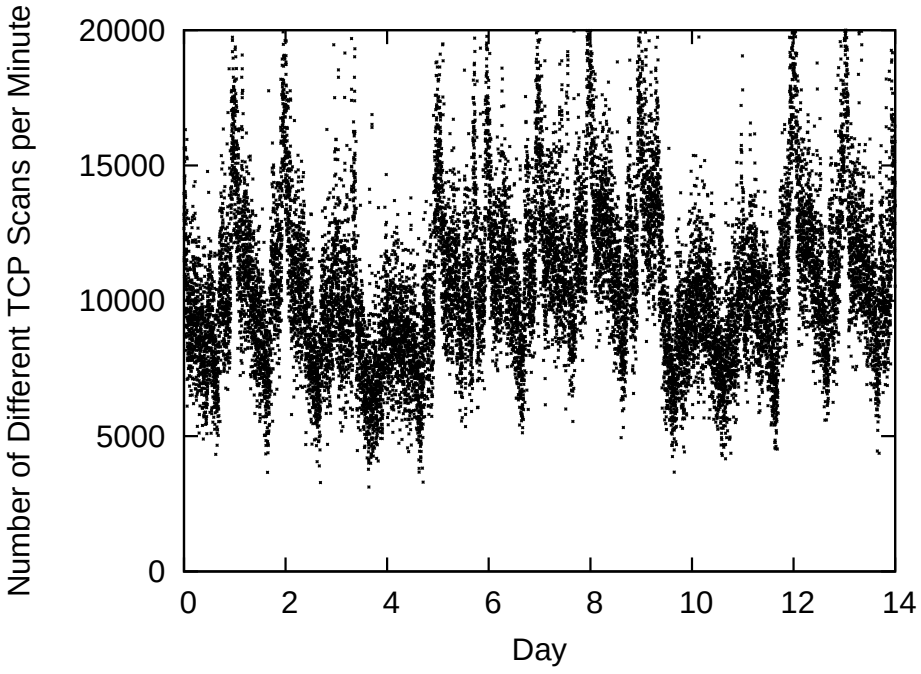
For our evaluation, we first examine the level of background radiation our network telescope captures and analyze the effectiveness of scan filtering for reducing this volume. We then evaluate the correctness and performance of the replay proxy. Finally, we present the worms GQ captured over the course of four months of operation.

5.4.1 Background Radiation and Scalability Analysis

Our network telescope currently captures the equivalent of a /14 network block (262,144 addresses), plus an additional "hot" /23 network (512 addresses). This latter block lies just above one of the private-address allocations. Any malware that performs sequential scanning using its local address as a starting point (e.g., Blaster [97]) and runs on a host assigned an address within the private-address block will very rapidly probe the adjacent public-address block, sending traffic to

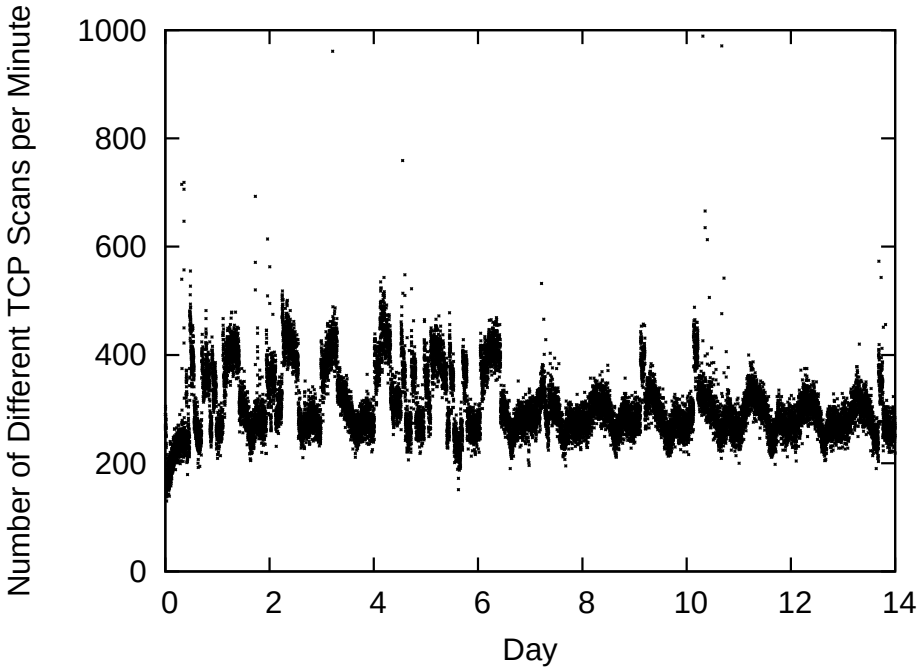


(a) TCP scans to the /14 network without any filtering.

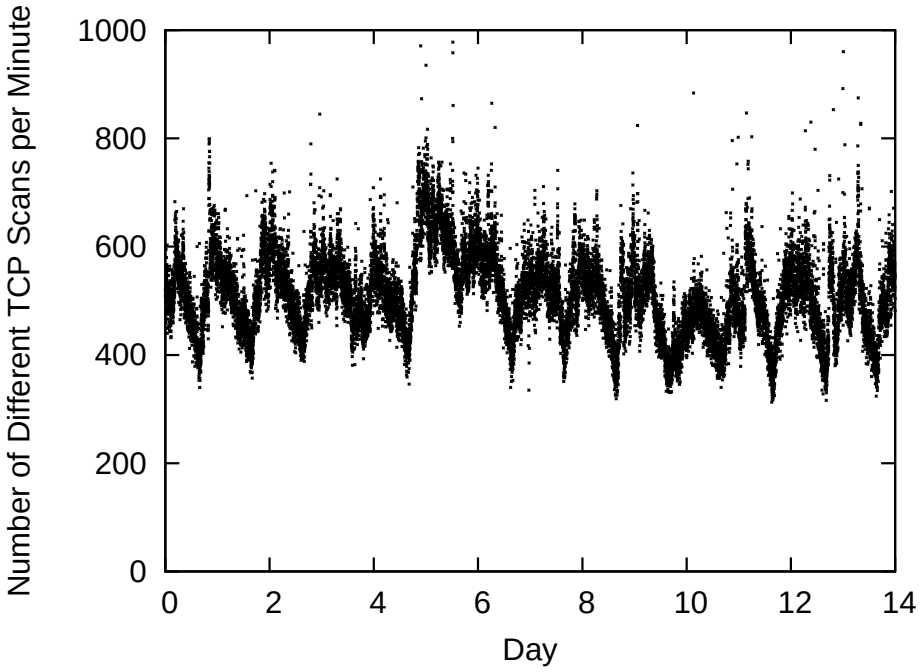


(b) TCP scans to the /23 network without any filtering.

Figure 5.4. TCP scans before scan filtering.



(a) TCP scans to the /14 network after scan filteringi.



(b) TCP scans to the /23 network after scan filtering.

Figure 5.5. TCP scans after scan filtering.

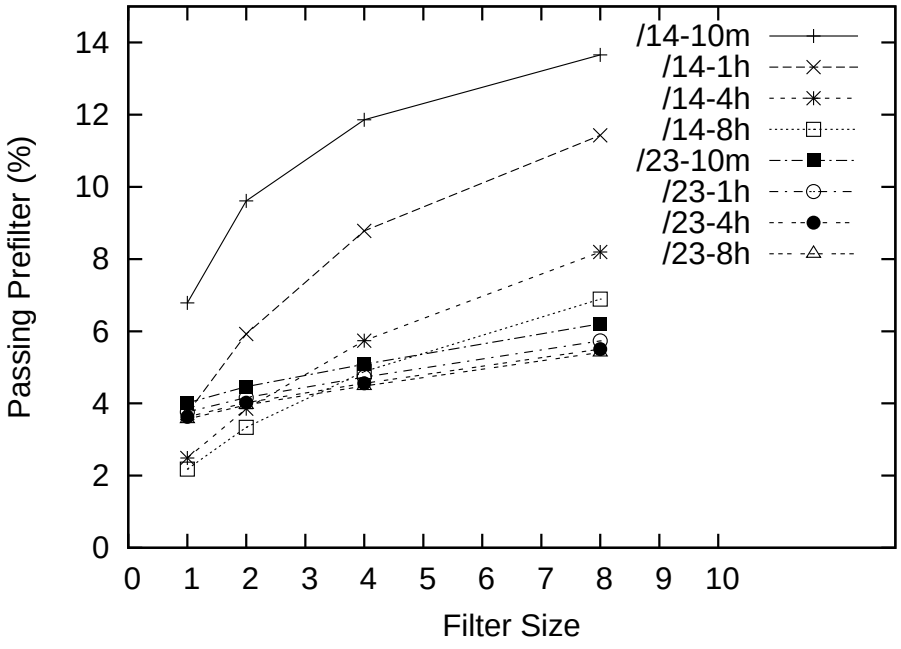


Figure 5.6. Effect of scan filter cutoff on proportion of probes that pass the prefilter.

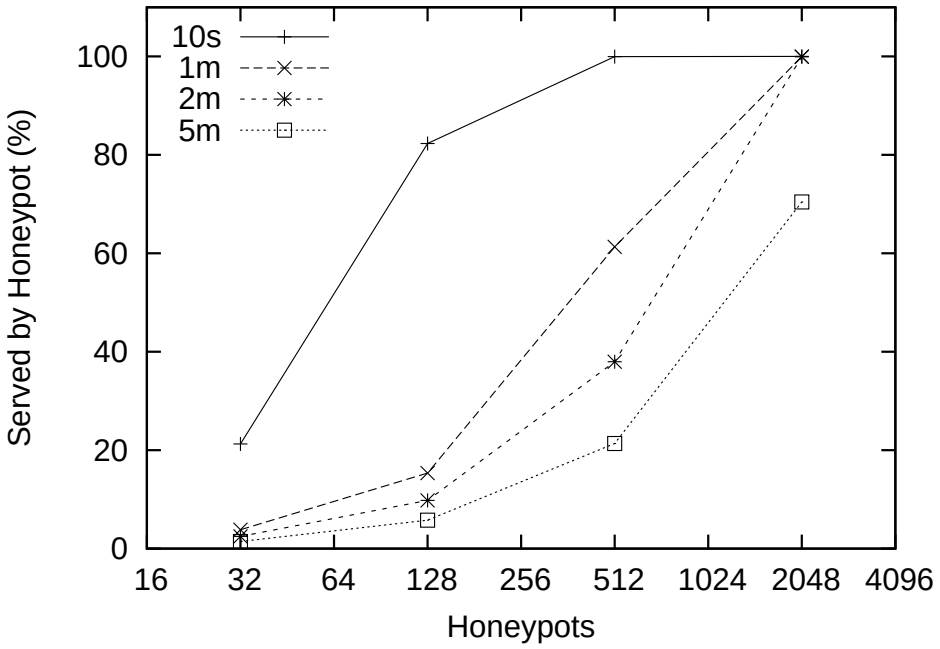


Figure 5.7. Effect of the number of honeypots on proportion of prefiltered probes that can engage a honeypot.

our telescope. Thus, this particular telescope feed provides *magnified* visibility into the activity of such malware.

We analyzed two weeks of network traffic recorded during honeyfarm operation. We find a huge bias in background radiation. Figures 5.4(a) and 5.4(b) show the number of TCP scans per minute seen by each telescope feed. Here, a "scan" means a TCP SYN from a distinct source-destination pair (any duplicate TCP SYNs that arrive within 60 seconds are considered as one). We see that despite being 1/512th the size, the /23 network sees *several times more* background radiation than does the /14 network, dramatically illustrating the opportunity it provides to capture probes from sequentially scanning worms early during their propagation.

Figures 5.5(a) and 5.5(b) show the reduction in the raw probing achieved by prefiltering the traffic to remove scans. Our prefilter's current configuration allows each source to scan at most $N = 4$ different telescope addresses within an hour. We picked a cutoff of 4 because we currently run three different honeypot configurations, and the GQ controller allocates honeypots with different configurations to the same source when it probes additional telescope addresses; we then chose $N = 4$ rather than $N = 3$ to add some redundancy.

This filtering has the greatest effect on traffic from the /23 network, which by its nature sees numerous intensive scans, for which the prefilter discards all but the first few probes. As a result, after prefiltering the honeyfarm as a whole receives over a dozen attack probes/sec.

We used trace-based analysis to study the impact of different parameters for the scan filtering. Figure 5.6 shows for the /14 and /23 network the proportion of the incoming traffic that passes the prefiltering stage for a given setting of N, the filter cutoff. The cutoff specifies number of different telescope addresses allowed for a remote source to probe for a given period of time (as shown by the different curves). For the /23 network, the pass-through proportion remains around 5%. For the /14 network, it is considerably higher, but prefiltering still reaps significant benefits in reducing the initial load.

However, even with aggressive prefiltering, more than eight attack probes pass the filtering every second. This level rapidly consumes the available honeypots. In our current operation, we use 20 of the 24 honeypots to engage remote attackers, and the other four for internal redirection.

Since engaging an attacker takes a considerable amount of time (including restarting the honeypot afresh after finishing), the honeyfarm is always saturated. To investigate the number of honeypots required in the absence of further filtering, we conducted a trace-based analysis using different mean honeypot "life cycles", where the life cycle consists of the attack engagement time plus the restart time. We varied the life cycle from ten seconds up to five minutes (note that Xen-like virtual machines may take only seconds [96] to restart, while a full restart for a VMware virtual machine may take a full minute).

Figure 5.7 shows that once the life cycle exceeds one minute, we would need over 1,000 honeypots to engage 90% of the attacks that pass the prefiltering stage. Thus we need some mechanism to filter down the attacks. To do so, we introduce the second stage of filtering provided by our replay proxy, which we assess next.

5.4.2 Evaluating the Replay Proxy

In this section we assess the correctness and efficacy of the replay proxy as a means for further reducing the volume of probes that the honeypots must process.

To assess correctness, we extracted 66 distinct worm attacks that successfully infected the honeypots and exhibited self-propagation inside the honeyfarm (detailed in Section 5.4.3). For each distinct attack, we validated the proxy's proper operation as follows:

1. We verified that the proxy indeed successfully replays the attack to compromise another vulnerable honeypot.

2. We ran the proxy with an incomplete script for the attack that only specifies part of the attack activity. We then launched the attack against the honeyfarm to confirm that the proxy would correctly follow the script through its end, and upon reaching the end would then successfully bring a vulnerable honeypot "up to speed" and mediate the continuation of the attack so that infection occurred.

3. We ran the proxy with a complete script of the attack to confirm that it successfully filtered further instances of the attack.

Of these, the first two are particularly critical, because incorrect operation can lead to either underestimating which other system configurations a given worm can infect (since we use replay to generate the test instances for constructing these vulnerability profiles), or failing to detect a novel attack (because the proxy does not manage to successfully transfer the new activity to a honeypot). Incorrect operation in the third case results in greater honeyfarm load, but does not undermine correctness.

Port	Nodes	Max Depth	Max Breadth	Pairs of Examples
80/tcp	28	5	7	39
135/tcp	548	176	13	914
139/tcp	312	55	13	63
445/tcp	3,977	104	42	1,551

Table 5.2. Summary of protocol trees.

Because we have not yet integrated the replay proxy into the operational honeyfarm, to assess its filtering efficacy we constructed a set of scripts for it to follow and then ran those against attacks drawn from traces of GQ's operation. To do so, we first extracted 42,004 attack events for which GQ allocated a honeypot. (Note that each attack might involve multiple connections.) We then followed the approach described in Section 5.3.1 to automatically find 2,572 pairs of same attacks. Using these pairs as the primary and secondary dialogs, we generated scripts for each and constructed four protocol trees for filtering, one for each of the four services accessed in the attack traces: 80/tcp, 135/tcp, 139/tcp, and 445/tcp. Table 5.2 shows the properties of the resulting trees.

Using this set of scripts to process the remaining 36,860 attack events, the proxy filters out 76% of them, indicating that operating it would offload the honeypots by a factor of four, a major gain.

Finally, it is difficult to directly assess the false positive rate (attacks that should not have been filtered out but were), because doing so entails determining the equivalence of the actual operational semantics of different sets of server requests. However, we argue that the rate should be low because we take a conservative approach by requiring that filtered activity must match all dynamic fields in a script, including large ADUs that correspond to buffer overruns or bulk data transfers.

Executable Name	Size (B)	MD5Sum	Worm Name	# Events	# Conns	Time (s)
a####.exe	10366	7a67f7c8...	W32.Zotob.E	4	3	29.0
a####.exe	10878	bf47cfe2...	W32.Zotob.H	9	3	25.2
a####.exe	25726	62697686...	Quarantined but no name	1	3	223.2
cpufanctrl.exe	191150	1737ec9a...	Backdoor.Sdbot	1	4	111.2
chkdisk32.exe	73728	27764a5d...	Quarantined but no name	1	4	134.7
dllhost.exe	10240	53bfe15e...	W32.Welchia.Worm	297	4 or 6	24.5
enbiei.exe	11808	d1ee9d2e...	W32.Blaster.F.Worm	1	3	28.9
msblast.exe	6176	5ae700c1...	W32.Balster.Worm	1	3	43.8
lsd	18432	17028f1e...	W32.Poxdar	11	8	32.4
NeroFil.EXE	78480	5ca9a953...	W32.Spybot.Worm	1	5	237.5
sysmsn.exe	93184	5f6c8c40...	W32.Spybot.Worm	3	3	79.6
MsUpdaters.exe	107008	aa0ee4b0...	W32.Spybot.Worm	1	5	57.0
RealPlayer.exe	120320	4995eb34...	W32.Spybot.Worm	2	5	95.4
WinTemp.exe	209920	9e74a7b4...	W32.Spybot.Worm	1	5	178.4
wins.exe	214528	7a9aee7b...	W32.Spybot.Worm	1	5	118.2
msnet.exe	238592	6355d4d5...	W32.Spybot.Worm	1	7	189.4
MSGUPDATES.EXE	241152	65b401eb...	W32.Spybot.Worm	2	5	125.3
ntsf.exe	211968	5ac5998e...	Quarantined but no name	1	5	459.4
scardsvr32.exe	33169	1a570b48...	W32.Femot.Worm	4	3	46.2
scardsvr32.exe	34304	b10069a8...	W32.Femot.Worm	1	3	66.5
scardsvr32.exe	34816	ba599948...	W32.Femot.Worm	55	3	96.6
scardsvr32.exe	35328	617b4056...	W32.Femot.Worm	2	3	179.6
scardsvr32.exe	36864	0372809c...	W32.Femot.Worm	1	5	49.3
scardsvr32.exe	39689	470de280...	W32.Femot.Worm	4	3	41.4
scardsvr32.exe	40504	23055595...	W32.Femot.Worm	1	3	41.1
scardsvr32.exe	43008	ff20f56b...	W32.Valla.2048	1	5	32.2
scardsvr32.exe	66374	f7a00ef5...	Quarantined but no name	1	7	54.8
scardsvr32.exe	205562	87f9e3d9...	W32.Pinfi	1	3	180.8

Table 5.3. Part 1 of summary of captured worms (worm names are reported by Symantec Antivirus).

5.4.3 Capturing Worms

In four months of operation, GQ inspected about 260,000 attacks, among which 2,792 attacks were worms and 32,884 attacks were not worms but successfully compromised honeypots (we observed suspicious outbound network requests from the honeypots). Overall, GQ automatically captured 66 distinct worms belonging to 14 different families. Table 5.3 and 5.4 (we split the table into two parts due to the large table size) summarize the different types, where a worm type corresponds to a unique MD5 checksum for the worm's executable code (note that all the worms we captured involved multiple connections and had buffer overflow or password guessing separated from their executables). Most of these executables have a name directly associated with them because they are uploaded as files by the worm's initial exploit (see below). The table also gives the size in bytes of the executable and how many times GQ captured an instance of the worm.

To identify the worms, we uploaded the malicious executables to a Windows virtual machine

Executable Name	Size (B)	MD5Sum	Worm Name	# Events	# Conns	Time (s)
x.exe	9343	986b5970...	W32.Korgo.Q	17	2	6.6
x.exe	9344	d6df3972...	W32.Korgo.T	7	2	9.5
x.exe	9353	7d99b0e9...	W32.Korgo.V	102	2	6.0
x.exe	9359	a0139d7a...	W32.Korgo.W	31	2	5.9
x.exe	9728	c05385e6...	W32.Korgo.Z	20	2	6.6
x.exe	11391	7f60162c...	W32.Korgo.S	169	2	6.6
x.exe	11776	c0610a0d...	W32.Korgo.S	15	2	8.6
x.exe	13825	0b80b637...	W32.Korgo.V	2	2	24.4
x.exe	20992	31385818...	W32.Licum	2	2	7.9
x.exe	23040	e0989c83...	W32.Korgo.S	3	2	10.4
x.exe	187348	384c6289...	W32.Pinfi	1	2	329.7
x.exe	187350	a4410431...	W32.Korgo.V	6	2	11.3
x.exe	187352	b3673398...	W32.Pinfi	5	2	20.1
x.exe	187354	c132582a...	W32.Pinfi	5	2	24.9
x.exe	187356	d586e6c2...	W32.Pinfi	2	2	27.5
x.exe	187358	2430c64c...	W32.Korgo.V	1	2	27.5
x.exe	187360	eb1d07c1...	W32.Pinfi	1	2	63.1
x.exe	187392	2d9951ca...	W32.Korgo.W	1	2	76.1
x.exe	189400	7d195c0a...	W32.Korgo.S	1	2	18.0
x.exe	189402	c03b5262...	W32.Pinfi	1	2	58.2
x.exe	189406	4957f2e3...	W32.Korgo.S	1	2	210.9
xxxx...x	46592	a12cab51...	Backdoor.Berbew.N	844	2	9.4
xxxx...x	56832	b783511e...	W32.Info.A	34	2	7.2
xxxx...x	57856	ab5e47bf...	Trojan.Dropper	685	3	10.0
xxxx...x	224218	d009d6e5...	W32.Pinfi	1	3	32.5
xxxx...x	224220	af79e0c6...	W32.Pinfi	3	2	34.2
n/a	10240	7623c942...	W32.Korgo.C	3	2	4.8
n/a	10752	1b90cc9f...	W32.Korgo.L	1	2	7.0
n/a	10752	32a0d7d0...	W32.Korgo.G	8	2	4.1
n/a	10752	ab7ecc7a...	W32.Korgo.N	2	2	5.3
n/a	10752	d175bad0...	W32.Korgo.G	3	2	5.4
n/a	10752	d85bf0c5...	W32.Korgo.E	1	2	5.6
n/a	10752	b1e7d9ba...	W32.Korgo.gen	1	2	5.0
n/a	10879	042774a2...	W32.Korgo.I	15	2	4.3
n/a	11264	a36ba4a2...	W32.Korgo.I	1	2	5.4
22 distinct files	n/a	n/a	W32.Muma.A	2	7	186.7
3 distinct files	n/a	n/a	W32.Muma.B	2	7	208.9
26/27/28 distinct files	n/a	n/a	BAT.Boohoo.Worm	4	72	384.9

Table 5.4. Part 2 of summary of captured worms (worm names are reported by Symantec Antivirus).

on which we installed Symantec Antivirus. While its identification is incomplete (and some of the names appear less convincing), since we lack access to a large worm corpus we include the names for completeness.

We captured not only buffer-overflow worms but also those that exploit weak passwords. All of those worm attacks required multiple connections to complete (as many as 72 for BAT.Boohoo.Worm), as listed in the penultimate column, and involved a data transfer channel separate from the exploit connection to upload the malicious executable to the victim. This nature highlights the weakness of using "first payload" techniques for filtering out known attacks from

background radiation: a large number of attacks only distinguish themselves as novel or known after first engaging in significant initial activity.

The table also shows the (minimum) detection time for each worm. We measure detection time as the interval between when the first scan packet arrived at the honeyfarm, to when a *second* honeypot (infected by redirection from the first honeypot that served the request itself) attempts to make an outbound connection attempt, which provides proof that we have captured code that self-propagates. Detection time depends on many factors: end host response delay, network latency, execution time of the malware within the honeypot, and redirection honeypot availability. We inspect network traces by comparing the times when a packet is read by the GQ controller and when it is forwarded to a virtual machine by it. We find that the delay due to processing by the GQ controller is essentially instantaneous, i.e., the clock (which has a granularity of 10^{-6} second) did not advance during processing by the controller.

While the times given in Table 5.3 and 5.4 seem high, often the culprit is a slow first stage during which the remote source initially infects a honeypot. When we tested GQ's detection process by releasing Code Red and Blaster within it, the detection time was around one second.

However, this detection time measures only the latency between a probe of a worm is admitted by the honeyfarm and the honeyfarm detects it as a worm. What is more critical for the possibility of automatic response and containment is the overall detection time of a new worm outbreak that measures the latency between the first spread attempt of the worm outbreak and the detection by the honeyfarm. Though we could apply the same techniques used in [69] to estimate the latency between the first spread attempt of a worm outbreak and its first probe hits our network telescope of a quarter million addresses, we still cannot estimate the overall detection time GQ may experience because only a small part of the probes that hit our network telescope are inspected by GQ in the current operation due to the facts that the replay proxy is not integrated and the number of virtual machine honeypots is limited. The biggest challenge in the future work is to solve these problems so that *all* the probes sent to the quarter million addresses can be inspected in real time. After it, we can study if the overall detection time experienced by GQ is low enough for automatic response and containment.

5.5 Summary

Recently, great interest has arisen in the construction of *honeyfarms*: large pools of honeypots that interact with probes received over the Internet to automatically determine the nature of the probing activity, especially whether it signals the onset of a global worm outbreak.

Building an effective honeyfarm raises many challenging technical issues. These include ensuring high-fidelity honeypot operation; efficiently discarding the incessant Internet "background radiation" that has only nuisance value when looking for new forms of activity; and devising and policing an effective "containment" policy to ensure that captured malware does not inflict external damage or skew internal analyses.

In this chapter we presented GQ, a high-fidelity honeyfarm system designed to meet these challenges. GQ runs fully functional servers across a range of operating systems known to be prime targets for worms (especially various flavors of Microsoft Windows), confining each server to a virtual machine to maintain full control over its operation once compromised. To cope with load, GQ leverages aggressive filtering, including a technique based on application-independent "replay" that holds promise to quadruple GQ's effective capacity.

Among honeyfarm efforts, our experiences with GQ are singular in that the scale at which we have operated it—monitoring more than a quarter million Internet addresses, and capturing 66 distinct worms in four months of operation—far exceeds that previously achieved for a high-fidelity honeyfarm. While much remains for pushing the system further in terms of capacity and automated, efficient operation, its future as a first-rate tool for analyzing Internet-scale epidemics appears definite.

Chapter 6

Conclusions and Future Work

We start this chapter with a summary of our contributions in Section 6.1 and finish it and this thesis with a discussion of directions for future work in Section 6.2.

6.1 Thesis Summary

In this thesis, we tackle the problem of automating detection of new unknown malware with our focus on an important environment, personal computers, and an important kind of malware, computer worms. We face two fundamental challenges: *false alarms* and *scalability*. To minimize false alarms, our approach is to *infer the intent* of user or adversary (the malware author). We infer user intent on personal computers by monitoring user-driven activity such as key strokes and mouse clicks. To infer the intent of worm authors, we leverage honeypots to detect *self-propagation* by first letting a honeypot become infected, and then letting the first honeypot infect another one, and so on. For the scalability problem—how to handle a huge number of repeated probes—our approach is to leverage a new technology, *protocol-independent replay* of application dialog, to filter frequent multi-stage attacks by replaying the server-side responses. Our main contributions are concluded as follows.

1. **BINDER: an Extrusion-based Break-In Detector for Personal Computers**.

   We designed a novel extrusion detection algorithm to detect break-ins of new unknown mal-

ware on personal computers. To detect extrusions, we first assume that user intent is implied by user-driven input such as key strokes and mouse clicks. We infer user intent by correlating outbound network connections with user-driven input at the process level, and use whitelisting to detect user-unintended but benign connections generated by system daemons. By doing so, we can detect a large class of malware such as worms, spyware and adware that (1) run as background processes, (2) do not receive any user-driven input, (3) and make outbound network connections. We implemented BINDER, a host-based detection system for Windows, to realize this algorithm. We evaluated BINDER on six computers used by different volunteers for their daily work over five weeks. Our limited user study indicates that BINDER controls the number of false alarms to at most five over four weeks on each computer and the false positive rate is less than 0.03%. To evaluate BINDER's capability of detecting break-ins, we built a controlled testbed using the Click modular router and VMware Workstation. We tested BINDER with the Blaster worm and 22 different email worms collected on a departmental email server over one week and showed that BINDER successfully detect the break-ins caused by all these worms. From this work, we demonstrated that user intent, a unique characteristic of personal computers, is a simple and effective detector for a large class of malware with few false alarms.

2. **RolePlayer: Protocol-Independent Replay of Application Dialog**.

   We designed and implemented RolePlayer, a system which, given examples of an application session, can mimic both the client side and the server side of the session for a wide variety of application protocols. A key property of RolePlayer is that it operates in an application-independent fashion: the system does not require any specifics about the particular application it mimics. It instead uses byte-stream alignment algorithms from bioinformatics to compare different instances of a session to determine which fields it must change to successfully replay one side of the session. Drawing only on knowledge of a few low-level syntactic conventions such as representing IP addresses using "dotted quads", and contextual information such as the domain names of the participating hosts, RolePlayer can heuristically detect and adjust network addresses, ports, cookies, and length fields embedded within the session, including sessions that span multiple, concurrent connections on dynamically assigned ports. We suc-

cessfully used RolePlayer to replay both the client and server sides for a variety of network applications, including NFS, FTP, and CIFS/SMB file transfers, as well as the multi-stage infection processes of the Blaster and W32.Randex.D worms. We can potentially use such replay for recognizing malware variants, determining the range of system versions vulnerable to a given attack, testing defense mechanisms, and filtering multi-step attacks. From this work, we demonstrated that it is possible to achieve automatic network protocol analysis for certain problems.

3. **GQ: a Large-Scale, High-Fidelity Honeyfarm System**.

   We designed, implemented, and deployed GQ, a large-scale, high-fidelity honeyfarm system that can detect new worm outbreaks by analyzing in real-time the scanning probes seen in a quarter million Internet addresses. In GQ, we overcame many technical challenges, including ensuring high-fidelity honeypot operation; efficiently discarding the incessant Internet background radiation that has only nuisance value when looking for new forms of activity; and devising and policing an effective containment policy to ensure that captured malware does not inflict external damage or skew internal analyses. GQ leverages aggressive filtering, including a technique based on application-independent replay that holds promise to quadruple GQ's effective capacity. Among honeyfarm efforts, our experiences with GQ are singular in that the scale at which we have operated it—monitoring more than a quarter million Internet addresses, and capturing 66 distinct worms in four months of operation—far exceeds that previously achieved for a high-fidelity honeyfarm.

6.2 Future Work

To win the war against malicious attackers, we must keep creating new defense mechanisms in the future. The work in this thesis sets the stage for follow-on work in a number of areas.

1. **Improving the performance of the replay technology**. From our discussion in Chapter 4 and 5 we can see that the replay technology has many potential applications. However, the replay proxy hasn't been integrated into the GQ honeyfarm operationally due to

its performance limitation (a single-threaded instance of the replay proxy can process about 100 concurrent probes per second). We plan to study this problem from three perspectives: (1) improving the replay algorithm to reduce the computational cost; (2) leveraging multiple threads/processes/machines; (3) converting its network input/output to use raw sockets. To improve the performance of the replay technology fundamentally, we need to have a better understanding of network application protocols (so that we can compare an ADU with replay "scripts" more efficiently). We need to develop new technologies for automatic inference of network application protocols. RolePlayer compares two examples of an application session to infer dynamic fields. We need to investigate what level of understanding we can achieve by analyzing hundreds of thousands of examples of a network application all together.

2. **Enriching the functionality of the honeyfarm technology**. In this thesis, we have studied detecting scanning worms (which probe a set of addresses to identify vulnerable hosts) by inspecting traffic from network telescopes on honeypots. We need to extend the design and operation of the GQ honeyfarm in two directions. First, once a worm is captured, we can perform a wide variety of analyses on it using the well-engineered, fully-controlled honeyfarm. For example, we can leverage existing technologies on vulnerability analysis and signature generation [62, 15, 11] to generate attack and vulnerability signatures. Moreover, we can verify if the analyses produce correct signatures by deploying them in the honeyfarm to check if repeated attacks can be blocked. How we can integrate these technologies in the honeyfarm efficiently remains to be a challenge. Second, we can use a honeyfarm system to detect and analyze other malware than scanning worms because the operation of the honeypot cluster is independent from the source of traffic. For example, we could feed the honeyfarm with spam emails sent to nonexistent recipients on well-known email servers such as those owned by universities; we could also subscribe honeyfarm nodes to application-layer networks such as instant messaging and peer-to-peer networks. We need to investigate new approaches to feed such data into the honeyfarm effectively.

In the long run, the following directions are interesting and important.

1. **High-speed inline network intrusion detection with application-level replay**. The replay proxy presented in Chapter 5 is promising on filtering previously seen multi-stage malicious attacks from a network telescope. Currently the inline network intrusion detection systems are limited to signature inspection. If we could apply the replay proxy in network intrusion detection, we could use it to reply to a suspicious request and replay the finished conversion against the original target host so that the conversation can be continued after the request is recognized as a normal one. It remains a challenge to achieve this with few false alarms at high speed.

2. **Emulating user (mis)behavior on honeypots**. Our work in this thesis shows that inferring intent is effective for malware detection and honeypots are a powerful tool for observing malware's behavior. However, existing work in honeypot research doesn't have users in the loop. This impedes honeypots from detect social-engineering attacks that exploit user's mis-behavior. On the other hand, attackers can detect the existence of honeypots by noticing zero user activity. So it is desirable to emulate user (mis)behavior on honeypots, but it remains a challenge due to the daunting complexity of user behavior.

3. **Virtual machines for better security**. In the GQ honeyfarm, virtual machine technology enables us to deploy and manage high-fidelity virtual honeypots. In addition to efficiency and isolation, an interesting property of virtual machines is disposability. If we could run multiple virtual machines instead of a single persistent operating system, we could run untrusted jobs on a separate virtual machine from those that we use for other activities. These virtual machines must be light weight with respect to resource consumption and boot-up speed. It also remains a challenge to design a usage model for simultaneous virtual machines such that it is easy for average users to migrate from today's single persistent operating system model.

Bibliography

[1] Adware.CNSMIN. http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=ADW_CNSMIN.A.

[2] Adware.Gator. http://securityresponse.symantec.com/avcenter/venc/data/adware.gator.html.

[3] Adware.Spydeleter. http://netrn.net/spywareblog/archives/2004/03/12/how-to-get-rid-of-spy-deleter/.

[4] James P. Anderson. Computer security threat monitoring and surveilance. Technical report, James P. Anderson Company, Fort Washington, PA, April 1980.

[5] Stefan Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communication Security*, 1999.

[6] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario, and David Watson. The Internet Motion Sensor - a distributed blackhole monitoring system. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2005.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[8] Marshall Beddoe. The protocol informatics project. http://www.baselineresearch.net/PI/.

[9] Richard Bejtlich. *Extrusion Detection: Security Monitoring for Internal Intrusions*. Addison-Wesley Professional, 2005.

[10] Kevin Borders and Atul Prakash. Web tap: Detecting covert web traffic. In *Proceedings of the 11th ACM Conference on Computer and Communication Security*, October 2004.

[11] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[12] Yu-Chung Cheng, Urs Holzle, Neal Cardwell, Stefan Savage, and Geoffrey Voelker. Monkey see, monkey do: a tool for TCP tracing and replaying. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[13] CipherTrust. Zombie statistics. http://www.ciphertrust.com/resources/statistics/zombie.php, June 2006.

[14] Evan Cooke, Michael Bailey, Z. Morley Mao, David Watson, Farnam Jahanian, and Danny McPherson. Toward understanding distributed blackhole placement. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, October 2004.

[15] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.

[16] Cybertrace. http://www.cybertrace.com/ctids.html.

[17] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian Grizzard, John Levine, and Henry Owen. HoneyStat: Local worm detection using honeypots. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[18] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2), February 1987.

[19] Jeff Dike. The user-mode linux. http://user-mode-linux.sourceforge.net/.

[20] Mark Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989.

[21] Daniel R. Ellis, John G. Aiken, Kira S. Attwood, and Scott D. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, October 2004.

[22] eXtremail. http://www.extremail.com/.

[23] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedings of IEEE Symposium on Security and Privacy*, 1996.

[24] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, MA, June 2006.

[25] Ajay Gupta and R. Sekar. An approach for detecting self-propagating email using anomaly detection. In *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection*, September 2003.

[26] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[27] Hewlett-Packard. http://www.hp.com/rnd/products/switches/2800_series/overview.htm.

[28] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[29] Honeynet. The honeynet project. http://www.honeynet.org/.

[30] Honeynet. Sebek. http://www.honeynet.org/tools/sebek/.

[31] Ruiqi Hu and Aloysius K. Mok. Detecting unknown massive mailing viruses using proactive methods. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[32] Computer Security Institute. 2005 CSI/FBI computer crime and security survey, 2005.

[33] Intel. Intel virtualization technology. http://www.intel.com/technology/computing/vptech/, 2005.

[34] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[35] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[36] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[37] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, April 2005.

[38] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.

[39] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[40] Eddie Kohler, Robert Morris, and Massimiliano Poletto. Modular components for network address translation. In *Proceedings of the Third IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.

[41] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of ACM SIGCOMM HotNets-II Workshop*, November 2003.

[42] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an Internet-scale event. In *Proceedings of the 2005 ACM Internet Measurement Conference*, October 2005.

[43] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, pages 471–482, April 1987.

[44] Wenke Lee and Salvatore J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the Seventh USENIX Security Symposium*, January 1998.

[45] Wenke Lee and Salvatore J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3(4), November 2000.

[46] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference*, December 2005.

[47] Zhenmin Li, Jed Taylor, Elizabeth Partridge, Yuanyuan Zhou, William Yurcik, Cristina Abad, James J. Barlow, and Jeff Rosendale. UCLog: A unified, correlated logging architecture for intrusion detection. In *the 12th International Conference on Telecommunication Systems - Modeling and Analysis (ICTSM)*, 2004.

[48] Yihua Liao and V. Rao Vemuri. Using text categorization techniques for intrusion detection. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[49] Don Libes. expect: Curing those uncontrollable fits of interaction. In *Proceedings of the Summer 1990 USENIX Conference*, pages 183–192, June 1990.

[50] libpcap. http://www.tcpdump.org/.

[51] Insecure.Com LLC. Nmap security scanner. http://www.insecure.org/nmap.

[52] McAfee Inc. McAfee Security Forensics. http://networkassociates.com/us/products/mcafee/forensics/security_forensics.htm.

[53] William Metcalf. Snort inline. http://snort-inline.sourceforge.net/.

[54] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer worm. *IEEE Magazine of Security and Privacy*, August 2003.

[55] David Moore and Colleen Shannon. The spread of the Witty worm. *IEEE Security and Privacy*, 2(4), July/August 2004.

[56] David Moore, Colleen Shannon, and Jeffery Brown. Code-Red: a case study on the spread and victims of an Internet worm. In *Internet Measurement Workshop*, November 2002.

[57] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the 2003 IEEE Infocom Conference*, San Francisco, CA, April 2003.

[58] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Network telescopes. Technical report, CAIDA, April 2004.

[59] David Moore, Geoffrey Voelker, and Stefan Savage. Inferring Internet denial-of-service activity. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[60] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[61] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[62] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[63] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet background radiation. In *Proceedings of the 2004 ACM Internet Measurement Conference*, October 2004.

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

[65] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A.N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[66] Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[67] PsTools. http://www.sysinternals.com/ntw2k/freeware/pstools.shtml.

[68] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[69] Moheeb Abu Rajab, Fabian Monrose, and Andreas Terzis. On the effectiveness of distributed worm monitoring. In *Proceedings of the 14th USENIX Secuirty Symposium*, Baltimore, MD, August 2005.

[70] rattle. Using process infection to bypass windows software firewalls. http://www.phrack.org/show.php?p=62&a=13, 2004.

[71] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Systems Administration Conference*, 1999.

[72] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation*, pages 258–266, May 1996.

[73] Stefan Saroiu, Steven D. Gribble, and Henry M. Levy. Measurement and analysis of spyware in a university environment. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, March 2004.

[74] Stuart E. Schechter, Jaeyeon Jung, and Arthur W. Berger. Fast detection of scanning worm

infections. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[75] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behavior. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[76] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proceedings of the 2002 ACM Conference on Computer and Communications Security*, November 2002.

[77] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[78] Steven R. Snapp, James Brentano, Gihan V. Dias, Terrance L. Goan, L. Todd Heberlein, Che Lin Ho, Karl N. Levitt, Biswanath Mukherjee, Stephen E. Smaha, Tim Grance, Daniel M. Teal, and Doug Mansur. DIDS (Distributed Intrusion Detection System)—motivation, architecture, and an early prototype. In *Proceedings of the 14th National Computer Security Conference*, Washington, DC, 1991.

[79] Eugene Spafford. An analysis of the Internet worm. In *Proceedings of European Software Engineering Conference*, September 1989.

[80] Lance Spitzner. Honeypot farms. http://www.securityfocus.com/infocus/1720, August 2003.

[81] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003.

[82] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The top speed of flash worms. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, October 2004.

[83] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[84] Marc Stiegler, Alan H. Karp, Ka-Ping Yee, and Mark Miller. Polaris: Virus safe computing for windows xp. Technical Report HPL-2004-221, HP Labs, December 2004.

[85] Marc Stiegler and Mark Miller. E and CapDesk. http://www.combex.com/tech/edesk.html.

[86] Symantec. Symantec Internet Security Threat Report, March 2006.

[87] Symantec Norton Antivirus. http://www.symantec.com/.

[88] Symantec Security Response - Alphabetical Threat Index. http://securityresponse.symantec.com/avcenter/venc/auto/index/indexA.html.

[89] Synmantec. Decoy server product sheet. http://www.symantec.com/.

[90] Tcpreplay: Pcap editing and replay tools for *NIX. http://tcpreplay.sourceforge.net.

[91] TDIMon. http://www.sysinternals.com/ntw2k/freeware/tdimon.shtml.

[92] Marina Thottan and Chuanyi Ji. Anomaly detection in IP networks. *IEEE Transactions on Signal Processing*, 51(8), August 2003.

[93] Aaron Turner. Flowreplay design notes. http://www.synfin.net/papers/flowreplay.pdf.

[94] Unicode. http://www.unicode.org/.

[95] VMware Inc. http://www.vmware.com/.

[96] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[97] W32.Blaster.Worm. http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html.

[98] W32.Randex.D. http://securityresponse.symantec.com/avcenter/venc/data/w32.randex.d.html.

[99] W32.Toxbot. http://securityresponse.symantec.com/avcenter/venc/data/w32.toxbot.html.

[100] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[101] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, November 2002.

[102] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, August 2004.

[103] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. Automatic misconfiguration troubleshooting with *PeerPressure*. In *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, December 2004.

[104] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *Proceedings of the First ACM Workshop on Rapid Malcode (WORM)*, October 2003.

[105] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[106] Wikipedia. Logit analysis. http://en.wikipedia.org/wiki/Logit.

[107] Matthew M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172, HP Labs Bristol, 2002.

[108] Windows Hooks API. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/hooks.asp.

[109] Windows Security Auditing. http://www.microsoft.com/technet/security/prodtech/win2000/secwin2k/09detect.mspx.

[110] WinDump. http://windump.polito.it/.

[111] WinPcap. http://winpcap.polito.it/.

[112] Cynthia Wong, Stan Bielski, Jonathan M. McCune, and Chenxi Wang. A study of mass-mailing worms. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, October 2004.

[113] Yinglian Xie, Hyang-Ah Kim, David R. O'Hallaron, Michael K. Reiter, and Hui Zhang. Seurat: A pointillist approach to anomaly detection. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[114] Jintao Xiong. ACT: Attachment chain tracing scheme for email virus detection and control. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, October 2004.

[115] Ka-Ping Yee. User interaction design for secure systems. In *Proceedings of the Fourth International Conference on Information and Communications Security*, Singapore, 2002.

[116] Vinod Yegneswaran, Paul Barford, and Vern Paxson. Using honeynets for Internet situational awareness. In *Proceedings of ACM SIGCOMM HotNets-IV Workshop*, November 2005.

[117] Vinod Yegneswaran, Paul Barford, and Dave Plonka. On the design and use of Internet sinks for network abuse monitoring. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[118] Michal Zalewski. P0f: A passive OS fingerprinting tool. http://lcamtuf.coredump.cx/p0f.shtml.

[119] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[120] ZoneAlarm. http://www.zonelabs.com/.

[121] Cliff Changchun Zou, Lixin Gao, Weibo Gong, and Don Towsley. Monitoring and early warning for Internet worms. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, October 2003.

[122] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Worm propagation modeling and analysis under dynamic quarantine defense. In *Proceedings of the First ACM Workshop on Rapid Malcode (WORM)*, October 2003.