REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
15-01-2007	Final report	06/15/2001 - 11/14/2006

4. TITLE AND SUBTITLE
ENABLING DYNAMIC SECURITY MANAGEMENT OF NETWORKED SYSTEMS VIA DEVICE-EMBEDDED SECURITY (SELF-SECURING DEVICES)

5a. CONTRACT NUMBER
F49620-01-1-0433

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Ganger, Gregory R.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA
15213

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
USAF, AFRL
AF Office of Scientific Research
875 North Randolf Street
Arlington, VA 22203
Dr Todd Combs/NM

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR / AFRL

11. SPONSOR/MONITOR'S REPORT

AFRL-SR-AR-TR-07-0118

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approve for Publice Release: Distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report summarizes the results of the work on the AFOSR's Critical Infrastructure Protection Pro-gram project entitled Enabling Dynamic Security Management of Networked Systems via Device-Embedded Security (Self-Securing Devices) funded by the Air Force Research Laboratory contract num-ber F49620-01-1-043.
The scientific goal of this CIP/URI effort was to fundamentally advance the state-of-the-art in network security and digital intrusion tolerance by exploring a new paradigm in which individual devices erect their own security perimeters and defend their own critical resources (e.g., network links or storage me-dia). Together with conventional border defenses (e.g., firewalls), such self-securing devices provide a flexible infrastructure for dynamic prevention, detection, diagnosis, isolation, and repair of successful breaches in borders and device security perimeters. More specifically, the research sought to understand the costs, benefits and appropriate realization of (1) multiple, increasingly-specialized security perimeters placed between attackers and specific re-sources; (2) independent security perimeters placed around distinct resources, isolating each from compro-mises of the others; (3) rapid and effective intrusion detection, tracking, diagnosis, and recovery, using the still-standing security perimeters as a solid foundation from which to proceed; (4) the ability to dynamically shut away compromised systems, throttling their network traffic at its sources and using secure channels to reactively advise their various internal components to in-crease their protective measures; and (5) the ability to effectively manage and dynamically update security policies within and among the devices and systems in a networked environment.
The underlying motivation throughout this research was to go beyond the "single perimeter" mindset that typifies today's security solutions and results in highly brittle protections.

15. SUBJECT TERMS
network security, intrusion survival, adaptive defenses

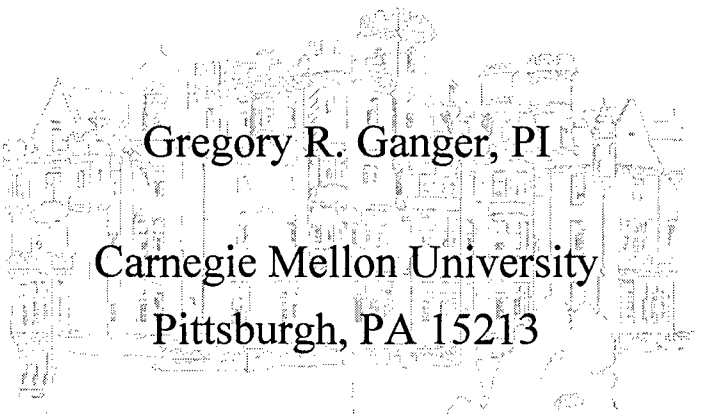
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE		186	Gregory R. Ganger
					19b. TELEPHONE NUMBER *(include area code)* 412-268-1297

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

ENABLING DYNAMIC SECURITY MANAGEMENT OF NETWORKED SYSTEMS VIA DEVICE-EMBEDDED SECURITY (SELF-SECURING DEVICES)

FINAL REPORT

Work funded by the AFOSR's Critical Infrastructure Protection Program
Air Force Research Laboratory contract number F49620-01-1-0433



Gregory R. Ganger, PI

Carnegie Mellon University

Pittsburgh, PA 15213

http://www.pdl.cmu.edu/SelfSecuring/

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This report summarizes the results of the work on the AFOSR's Critical Infrastructure Protection Program project entitled Enabling Dynamic Security Management of Networked Systems via Device-Embedded Security (Self-Securing Devices) funded by the Air Force Research Laboratory contract number F49620-01-1-043.

The scientific goal of this CIP/URI effort was to fundamentally advance the state-of-the-art in network security and digital intrusion tolerance by exploring a new paradigm in which individual devices erect their own security perimeters and defend their own critical resources (e.g., network links or storage media). Together with conventional border defenses (e.g., firewalls), such self-securing devices provide a flexible infrastructure for dynamic prevention, detection, diagnosis, isolation, and repair of successful breaches in borders and device security perimeters. More specifically, the research sought to understand the costs, benefits and appropriate realization of

1. multiple, increasingly-specialized security perimeters placed between attackers and specific resources;
2. independent security perimeters placed around distinct resources, isolating each from compromises of the others;
3. rapid and effective intrusion detection, tracking, diagnosis, and recovery, using the still-standing security perimeters as a solid foundation from which to proceed;
4. the ability to dynamically shut away compromised systems, throttling their network traffic at its sources and using secure channels to reactively advise their various internal components to increase their protective measures; and
5. the ability to effectively manage and dynamically update security policies within and among the devices and systems in a networked environment.

The underlying motivation throughout this research was to go beyond the "single perimeter" mindset that typifies today's security solutions and results in highly brittle protections.

1.1 Approach

To achieve these scientific objectives, the CMU team pursued a broad program of prototype building, experimental measurement, real-world activity tracing, and theory development. Augmenting individual border protection schemes, such as firewalls and COTS operating systems, with self-securing devices creates the potential for much greater robustness and flexibility for network security administrators. By having each device (e.g., network card, storage device, router/switch) erect independent security perimeters, the overall network environment gains many outposts from which to act when under attack. Devices can not only protect their own resources, they can observe, log, and react to the actions of other nearby devices. Compromises of one security perimeter will disrupt only a small fraction of the network environment, allowing other devices to dynamically identify the problem, warn still-secured devices about the compromised system, raise the security levels within the environment, and so forth. In exploring this new network security paradigm, we examined two broad research questions, each with a number of subquestions. The first was "what should each device do behind its security perimeter?" This question was answered by experimentally exploring the practical cost, performance, and flexibility trade-offs inherent in implementing a new type of security regime, as well as by discovering just what can be observed from the limited information available at any given device. The second question, "how does one dynamically manage a large collection of independent security perimeters?" was answered by exploring the practical and foundational aspects of techniques necessary for marshalling sets of self-securing devices, monitoring their current state, and dynamically updating policies in the face of changes to and attacks upon the network environment's state.

1.2 Contributions

Security compromises are a fact of life and a major concern for national defense as well as corporate America. No single defense is adequate, and this project developed and explored a model in which security functionality is distributed among physically distinct components. Inspired by siege warfare (Figure 1.1), the project explored the value of individual devices erecting their own security perimeters and defending their own critical resources (e.g., network link or storage media).

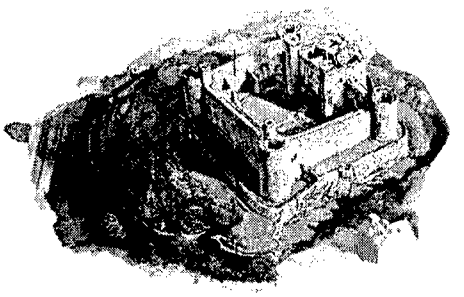


Figure 1.1: Harlech Castle, a 13th century castle in Gwynedd, West Wales, UK, capable of withstanding lengthy siege attacks due to the nature of its defensive position and construction.

Some significant accomplishments of this research:

- invention of storage-based intrusion detection, by which storage devices become a powerful new vantage point for spotting successful intrusions.
- development of NIC-embedded intrusion detection, isolation, and propagation mechanisms capable of, for example, slowing Internet worm propagation speeds from 15 minutes to 6 months.
- introduction of self-securing storage approaches for protecting data from intruders, enhancing posthoc diagnosis of and recovery from intrusions.
- creation of models for quantifying the effects of worm/attack mitigation techniques, such as NIC-embedded isolation and aggressive host inoculation, on worm/attack propagation.
- demonstration of functional prototypes of self-securing storage, self-securing network interface cards, and storage-based intrusion detection all managed by a common administrative console.
- discovery of additional uses for device-embedded data protection and versioning mechanisms — most notably for efficient survivable storage protocols — strengthening the case for their inclusion in future storage infrastructures.

The self-securing devices project has been quite large, and has resulted in many important contributions and the completion of all specified tasks. Although this report details the full effort, some notable results are highlighted in the following sections.

1.2.1 The Self-Securing Devices Concept

One of the first tasks of this project was think through approaches to security that could prevent the high-profile computer security failures that have become regular news items. In Chapter 2, and in "Better Security via Smarter Devices" [Ganger01] and "Enabling Dynamic Security Management of via Device-Embedded Security" [Ganger00], we explore why common approaches to computer security continue to fail and outline a more promising approach. Inspired by constructs from siege warfare, this approach extends the many devices in a computing infrastructure such that they each treat their interfaces as security perimeters and participate in defending the infrastructure. Together with conventional border defenses, like firewalls, such self-securing devices can provide dynamic prevention, detection, diagnosis, isolation, and repair when intruders get past an outer perimeter.

Thus, it was necessary to the project to build up proof-of-concept prototypes from which we could experimentally explore the research ideas. We have developed and experimented extensively with prototypes of a self-securing NIC and two self-securing NFS servers, and used them to explore both implementation trade-offs and fundamental costs associated with device-embedded security functionality. We have used these prototypes to explore new ways that individual self-securing devices can contribute to intrusion survival. We have also built up programming models for extending self-securing devices and administrative interfaces for sysadmins to work with them.

1.2.2 Self-Securing Network Interfaces

Self-securing network interfaces (NIs) examine the packets that they move between network links and host software, looking for and potentially blocking malicious network activity. By doing so, self-securing network interfaces can help administrators to identify and contain compromised machines within their intranet. By shadowing host state, self-securing NIs can better identify suspicious traffic originating from that host, including many explicitly designed to defeat network intrusion detection systems. With normalization and detection-triggered throttling, self-securing NIs can reduce the ability of compromised hosts to launch attacks on other systems inside (or outside) the intranet. In Chapter 3 we discuss a software architecture for self-securing NIs that separates scanning software into applications (called scanners) running on a NI kernel. The resulting scanner API simplifies the construction of scanning software and allows its powers to be contained even if it is subverted. We have developed a prototype self-securing NI and several example scanners for detecting such things as TTL abuse, fragmentation abuse, "SYN bomb" attacks, and random-propagation worms like Code-Red. This work has been published in two Carnegie Mellon University School of Computer Science Technical Reports "Self-Securing Network Interfaces: What, Why, and How" [Ganger02] and "Finding and Containing Enemies within the Walls with Self-securing Network Interfaces" [Ganger03a].

1.2.3 Self-Securing Storage

Self-securing storage turns storage devices into active parts of an intrusion survival strategy. From behind a thin storage interface (e.g., SCSI or CIFS), a self-securing storage server can watch storage requests, keep a record of all storage activity, and prevent compromised clients from destroying stored data. We explored three ways that self-securing storage enhances an administrator's ability to detect, diagnose, and recover from client system intrusions. First, storage intrusion detection offers a new observation point for noticing suspect activity. Second, post-hoc intrusion diagnosis starts with a plethora of normally-unavailable information. Third, post-intrusion recovery is reduced to restarting the system with a pre-intrusion storage image retained by the server. Combined, these features can improve an organization's ability to survive successful digital intrusions. The directions pursued in this work are overviewed Chapter 4, as well as in the Carnegie Mellon University School of Computer Science Technical Report "Intrusion Detection, Diagnosis, and Recovery with Self-Securing Storage" [Strunk02] and "Self-Securing Storage: Protecting Data in Compromised Systems" [Strunk00], presented at the 4th Symposium on Operating System Design and Implementation (OSDI00).

Chapter 4, and the paper "Metadata Efficiency in a Comprehensive Versioning File System" [Soules03], which was presented at the 2nd Usenix Conference on File and Storage Technologies (FAST03), also discuss the diagnosis and recovery features of self-securing storage, which arise because it prevents intruders from undetectably tampering with or permanently deleting stored data. To accomplish this, self-securing storage devices internally audit all requests and keep all versions of all data for a window of time, regardless of the commands received from potentially-compromised host operating systems. Within the window, system administrators have this valuable information for intrusion diagnosis and recovery. We have built a prototype self-securing NFS server that combines log-structuring with novel metadata journaling and data replication techniques to minimize the performance costs of comprehensive versioning. Experiments show that self-securing storage devices can deliver performance that is comparable with conventional storage. Further, analyses indicate that several weeks' worth of all versions can reasonably be kept on state-of-the-art disks, especially when differencing and compression technologies are employed.

1.2.4 Storage-Based Intrusion Detection

Another significant focus of the self-securing devices project has been on how self-securing storage devices enhance intrusion detection via a new mechanism: storage-based intrusion detection. Chapter 5 details how storage-based intrusion detection allows storage systems to watch for data modifications characteristic of system intrusions. Storage systems are able to spot several common intruder actions, such as adding backdoors, inserting Trojan horses, and tampering with audit logs. Further, an intrusion detection

system (IDS) embedded in a storage device continues to operate even after client systems are compromised. We have identified a number of specific warning signs visible at the storage interface. Examination of a number of real intrusion tools reveals that most intrusions can be detected based on changes to stored files. We have built and evaluated a prototype storage IDS, embedded in our self-securing NFS server, demonstrating both the feasibility and efficiency of storage-based intrusion detection. We also evaluate the case for implementing IDS functionality in the firmware of workstations' locally attached disks, on which the bulk of important system files typically reside. Experimental results from our prototype indicate that it would indeed be feasible, in terms of CPU and memory costs, to include IDS functionality in low-cost desktop disk drives. This work is described in "Storage-based Intrusion Detection: Watching Storage Activity For Suspicious Behavior" [Pennington03], a paper presented at the 2003 Usenix Security Conference, and in "On the Feasibility of Intrusion Detection inside Workstation Disks" [Griffin03], a CMU SCS technical report.

1.2.5 Blocking Rapidly Propagating Worms

Mass-mailing worms have made a significant impact on the Internet. These worms consume valuable network resources and can also be used as a vehicle for DDoS attacks. Chapter 6 discusses our work on using rate limiting mechanisms to block the invasive software intrusions of mass mailing worms. Specifically, we were interested in analyzing different deployment strategies of rate control mechanisms and the effect thereof on suppressing the spread of worm code. To provide context for our analysis, we examine real traffic traces obtained from a campus computing network and analyzed several different methods of rate limiting, investigating the impact of the critical parameters for each scheme in seeking to understand how these parameters might be employed in realistic network settings. We observed that rate throttling could be enforced with minimal impact on legitimate communications and that using DNS-based rate limiting has substantially lower error rates than schemes based on other traffic statistics. Two worms observed in the traces were significantly slowed down. Several papers describing our results in this area have been published. "Dynamic Quarantine of Internet Worms" [Wong04a] was presented at the International Conference on Dependable Systems and Networks (DSN-2004), "A Study of Mass-mailing Worms" [Wong04b] was presented at WORM'04, and "Empirical Analysis of Rate Limiting Mechanisms" [Wong05] was presented at the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005).

1.2.6 Survivable Storage in Self-Securing Storage Devices

Developing and experimenting with self-securing storage has uncovered another powerful feature of comprehensive versioning: it enables robust, high-performance consistency in survivable storage systems. Building on the self-securing storage prototype, we have designed, implemented, and evaluated a family of protocols for survivable, decentralized data storage. These protocols exploit storage-node versioning to efficiently achieve strong consistency semantics. These protocols allow erasure-codes to be used that achieve network and storage efficiency (and optionally data confidentiality in the face of server compromise). The protocol family is general in that its parameters accommodate a wide range of fault and timing assumptions, up to asynchrony and Byzantine faults of both storage-nodes and clients, with no changes to server implementation or client-server interface. Measurements of the prototype storage system using these protocols show that it scales well with the number of failures tolerated, and its performance compares favorably with an efficient implementation of Byzantine-tolerant state machine replication. Early progress on this work is described in the CMU SCS technical reports "Decentralized Storage Consistency via Versioning Servers" [Goodson02] and "Efficient Consistency for Erasure-coded Data via Versioning Servers" [Goodson03]. Advancements to this work were presented as follows: "Efficient Byzantine-tolerant Erasure-coded Storage" [Goodson04a], presented at the International Conference on Dependable Systems and Networks 2004 (DSN04) and "Lazy Verification in Fault-Tolerant Distributed Storage Systems" [Abd-El-Malek05], presented at the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005).

1.3 Technology Transfer

This research has had major impact, to date, with much more on the horizon. It has inspired new research efforts at unrelated Universities, notably the Michigan and Stanford projects exploring the use of virtual machines (offering self-securing features via virtualization) for intrusion detection and diagnosis. More importantly, from a practical standpoint, major infrastructure companies from industry are exploring and even beginning product development. 3Com, an early supporter of this research, now offers network card extensions for per-machine firewall functionality, taking a first step towards full self-securing network interfaces. Similarly, most high-end file servers, offered by supporters of this research, can now be extended with virus detectors, taking a first step towards full storage-based intrusion detection. This DoD-sponsored research is pioneering a new class of security product, which industry had no intention of considering on its own, potentially leading to major benefit for both parties by addressing DoD needs and enhancing U.S. industry.

Over the years, CMU's infrastructure researchers have developed a proven approach to technology transfer, based on a combination of open source software and industry consortia. Of course, sharing software prototypes publicly allows others to replicate our experiments and build on the work. To ensure that our research results transfer to military systems, however, we also work closely with the industry that provides military infrastructure. We have a close relationship and frequent interactions with industry leaders through our solid consortia in storage systems and security. Such interactions achieve the kind of technology transfer truly needed for new infrastructure approaches to become reality: integration into COTS components.

Along with publishing our results in several recognized, juried conferences, over the course of this research we have spent large amounts of time explaining and demonstrating our fault- and intrusion-tolerant services to industry leaders from the many companies who support our research over the lifetime of this project (including APC, Cicso, EMC Corporation, EqualLogic, Inc., Hewlett-Packard Labs, Hitachi, IBM, Intel Corporation, Microsoft Research, Network Appliance, Oracle Corporation, Panasas, Inc., Seagate Technology, Sun Microsystems, Symantec and Veritas). These interactions have refined the practicality of our work and its acceptance by the various companies. Important meetings with our sponsors include the annual Parallel Data Lab (http://www.pdl.cmu.edu) Retreat and Open House events, during which technical leaders (20 to 35 at each gathering) from member companies hear about and discuss our research activities. Our AFOSR Program Manager, Major Juan Vasquez, also attended two of these meetings. Based on these interactions, we expect to see several of the concepts explored in this research appearing in real products in the near term and we feel it is the most effective way for technology transfer of infrastructure research projects.

1.3.1 Personnel Development

During the course of this research, over its full 5 year term, eight Ph.D. students completed their degrees, 16 Master's degrees were obtained, and two postdocs completed their activities. As well several Masters and younger Ph.D. students are continuing to make progress.

1.4 Self-Securing Devices Report Overview

The rest of this report is structured as follows: Chapter 2 offers a high-level overview of the self-securing devices concept. Chapter 3 introduces self-securing network interfaces and explains their role in containing malicious network activity. Chapter 4 describes self-securing storage and our implementation of a self-securing storage server. It also discusses versioning file systems as a method of recovery from user mistakes or system corruption. Chapter 5 explains storage-based intrusion detection, diagnosis and recovery. Chapter 6 introduces a number of responses to rapid propagation attacks and worms (e.g., email mass mailing worms such as MyDoom) and discusses rate limiting mechanisms as a solution. Chapter 7 discusses survivable storage, where storage systems spread data redundantly across a set of distributed storage-nodes to ensure its availability despite node failure or compromise, based on self-securing devices.

2 BETTER SECURITY VIA SMARTER DEVICES

2.1 Overview

Computer security breaches are a huge and growing challenge for digital infrastructures. Since such infrastructures play critical roles in our economic, government, and military interests, we must find ways of better preventing intrusions and mitigating the damage that they can do. Doing so, however, is going to require fundamental change in the approaches taken to computer security.

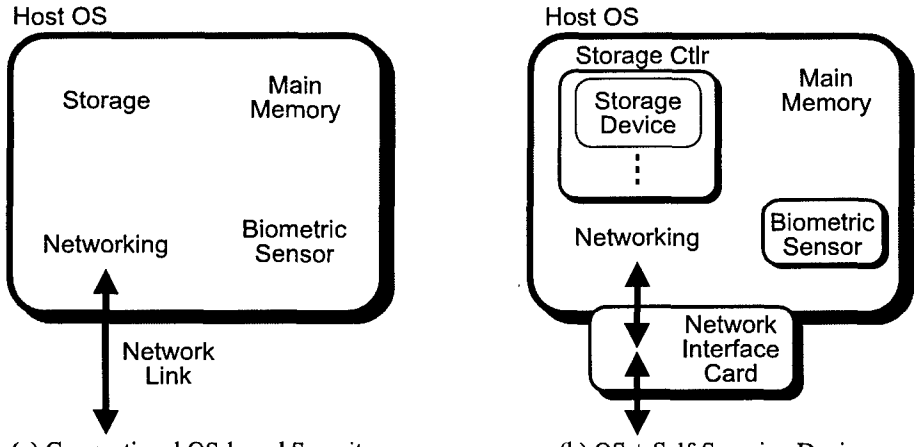
Today's main security tools provide border defenses and watch for attempts to breach them. The most common border defense is the network "firewall," which sits between a managed network and the rest of the Internet. It examines packets going in each direction, allowing only those that conform to rules defined by the network's administrator. A border exists within most computer systems, as well, in the form of the operating system's kernel and administrative roles. Behind each of these borders, intrusion detection systems can look for attempts to breach the border defense.

This conventional security architecture is fundamentally brittle, because it relies on a small number of border protections to protect a large number of resources and services with many users. This reliance comes with three major difficulties: (1) the many interfaces and functionalities needed to support many services make correct implementation and administration extremely difficult; the practical implications are daily security alerts for popular OSs and network applications; (2) once they bypass a border protection, attackers can freely manipulate everything it protects, which greatly complicates most phases of security management, including intrusion detection, isolation, diagnosis, and recovery; intruders commonly disable intrusion detection systems and remove traces of their presence after breaking into a system; (3) having a central point of security checks creates performance, fault-tolerance, and flexibility limitations for large-scale environments.

In practice, it is impossible to completely prevent intrusions into a network of any size and function. There are simply too many humans and too much software (created by humans) involved. Humans are fallible. They make mistakes when building software, when configuring systems, and when giving trust. All of these open doorways for intruders. And, the doorways get wider as systems become more flexible, since it becomes harder and harder to distinguish good uses of the flexibility from bad.

Users of computer systems are not going to allow a reverse in the trend towards greater functionality, so a different security approach is needed. This chapter promotes an alternative architecture in which individual system components erect their own security perimeters and protect their resources (e.g., network, storage, or video feed) from intruder tampering. This "self-securing devices" architecture distributes security functionality amongst physically isolated components, avoiding much of the fragility and unmanageability inherent in today's border-based security. Specifically, this architecture addresses the three fundamental difficulties by: (1) simplifying each security perimeter (e.g., consider network card or disk interfaces), (2) reducing the power that an intruder gains from compromising just one of the perimeters, and (3) distributing security enforcement checks among the many components of the system.

Conventional CPUs will still run application programs, but they won't dictate which packets are transferred onto network wires and they won't dictate which disk blocks are overwritten. Instead, self-securing network interface cards (NICs) will provide firewall and proxy server functionality for a given host, as well as throttling or labelling its outbound traffic when necessary. Likewise, self-securing storage devices will detect and protect their data from compromised client systems, and self-securing user interface components can assist with end-to-end trust in the sources of user input and system output. In a system of self-securing devices, compromising the OS of the main CPU won't give a malicious party complete control over all system resources — to gain complete power, an intruder must also compromise the disk's OS, the network card's OS, etc. Similarly, getting past the outer firewall will not give unchecked access to the internal network; self-securing switches, NICs, and routers will still be in the way.



(a) Conventional OS-based Security (b) OS + Self-Securing Devices

Figure 2.1. Two security approaches for a computer system. On the left, (a) shows the conventional approach, which is based on a single perimeter around the set of system resources. On the right, (b) shows our new approach, which augments the conventional security perimeter with perimeters around each self-securing device. These additional perimeters offer additional protection and flexibility for defense against attackers. Firewall-enforced network security fits a similar picture, with the new architecture providing numerous new security perimeters within each system on the internal network.

Augmenting current border protections with self-securing devices promises much greater flexibility for security administrators. By having each device erect an independent security perimeter, the network environment gains many outposts from which to act when under attack. Devices not only protect their own resources, but they can observe, log, and react to the actions of other nearby devices. Infiltration of one security perimeter will compromise only a small fraction of the environment, allowing other devices to dynamically identify the problem, alert still-secured devices about the compromised components, raise the security levels of the environment, and so forth.

Self-securing devices will require more computational resources in each device. However, with rapidly shrinking hardware costs, growing software development costs, and astronomical security costs, it makes no sense to not be throwing hardware at security problems. A main challenge has been identifying the best ways to partition (and replicate) functionality across self-securing components in order to enhance security and robustness. A corollary challenge is to re-marshal the distributed functionality to achieve acceptable levels of performance and manageability. After describing our inspiration for this architecture (medieval siege warfare), this chapter overviews some of our experiences and success stories from years of research.

2.2 Siege Warfare in the Internet Age

Despite enormous effort and investment, it has proven nearly impossible to prevent computer security breaches. To protect our critical information infrastructures, we need defensive strategies that can survive determined and successful attacks, allowing security managers to dynamically detect, diagnose, and recover from breaches in security perimeters. Borrowing from lessons learned in pre-gun warfare, we propose a new network security architecture analogous to medieval defense constructs.

Current security mechanisms are based largely on singular border protections. This roughly corresponds to defense practices during Roman times, when defenders erected walls around their camps and homes to provide protective cover during attacks. Once inside the walls, however, attackers faced few obstacles to gaining access to all parts of the enclosed area. Likewise, a cracker who successfully compromises a firewall or OS has complete access to the resources protected by these border defenses—no additional obsta-

cles are faced.[1] Of course, border defenses were a large improvement over open camps, but they proved difficult to maintain against determined attackers — border protections can be worn down over time and defenders of large encampments are often spread thin at the outer wall.

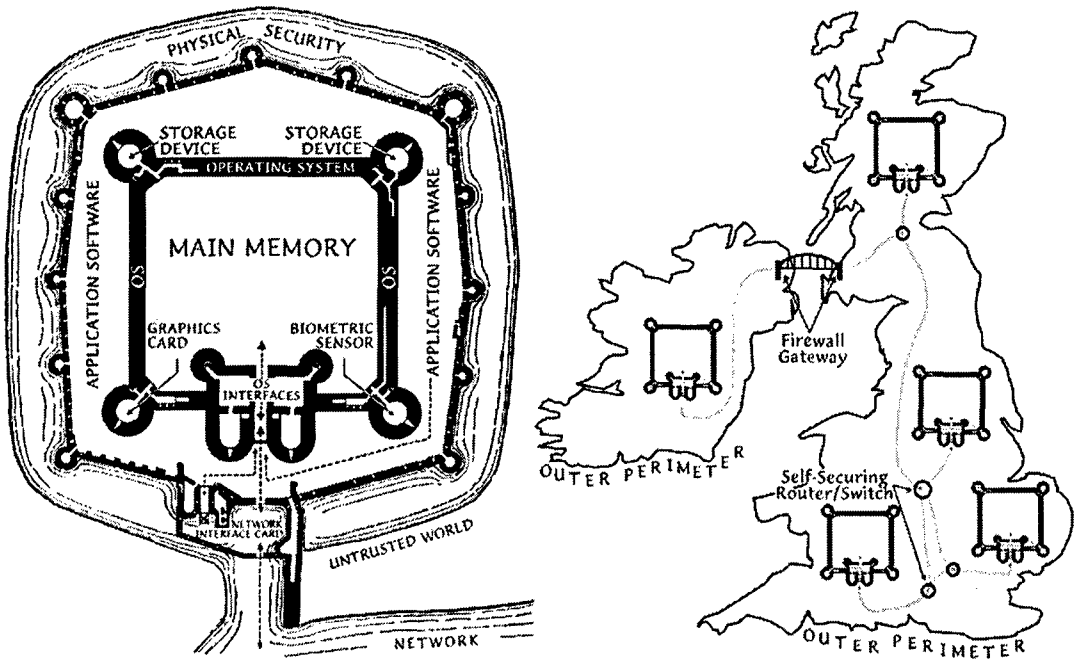


Figure 2.2. The self-securing device architecture illustrated via the siege warfare constructs that inspired it. On the left, (a) shows a siege-ready system with layered and independent tiers of defense enabled by device-embedded security perimeters. On the right, (b) shows two small intranets of such systems, separated by firewall-guarded entry points. Also note the self-securing routers/switches connecting the machines within each intranet.

As the size and sophistication of attacking forces grew, so did the sophistication of defensive structures. The most impressive such structures, constructed to withstand determined sieges in medieval times, used multiple tiers of defenses. Further, tiers were not strictly hierarchical in nature — rather, some structures could be defended independently of others. This major advancement in defense capabilities provided defenders with significant flexibility in defense strategy, the ability to observe attacker activities, and the ability to force attackers to deal with multiple independent defensive forces.

Applying the same ideas to computer and network security, border protections (i.e., firewalls and host OSs) can be augmented with security perimeters erected at many points within the borders. Enabled by low-cost computation (e.g., embedded processors, ASICs), security functionality can be embedded in most device microcontrollers, yielding "better security via smarter devices." We refer to devices with embedded security functionality as *self-securing devices*.

Self-securing devices can significantly increase network security and manageability, enabling capabilities that are difficult or impossible to implement in current systems. For example, independent device-

[1] This is not quite correct in the case of a firewall protecting a set of hosts that each run a multi-program OS, such as Linux. Such an environment is more like a town of many houses surrounded by a guarded wall. Each house affords some protection beyond that provided by the guarded wall, but not as much in practice as might be hoped. In particular, most people in such an environment will simply open the door when they hear a knock, assuming that the wall keeps out attackers. Worse, in the computer environment, homogeneity among systems results in a single set of keys (attacks) that give access to any house in the town.

embedded security perimeters guarantee that a penetrated boundary does not compromise the entire system. Uncompromised components continue their security functions even when other system components are compromised. Further, when attackers penetrate one boundary and then attempt to penetrate another, uncompromised components can observe and react to the intruder's attack; from behind their intact security perimeters, they can send alerts to the security administrator, actively quarantine or immobilize the attacker, and wall-off or migrate critical data and resources. Pragmatically, each self-securing device's security perimeter is simpler because of specialization, which should make correct implementations more likely. Further, distributing security checks among many devices reduces their performance impact and allows more checks to be made.

By augmenting conventional border protections with self-securing devices, this new security architecture promises substantial increases in both network security and security manageability. As with medieval fortresses, well-defended systems conforming to this architecture could survive protracted sieges by organized attackers.

2.3 Device-embedded Security Examples

This section overviews some prime examples of self-securing devices and what can be done with them. By embedding intrusion survival functionality into devices, one enjoys their benefits even when host OSes, user accounts, and other devices are compromised.

2.3.1 Network Interface Cards (NICs)

The role of NICs in a computer system is to move packets between the system's components and the network. Thus, the natural security extension is to enforce security policies on packets forwarded in each direction, looking for and potentially blocking malicious network activity [Friedman00, Ganger02, Ganger03a].

Like a firewall, a self-securing NIC can examine packet headers and simply not forward unacceptable packets into or out of the computer system; in fact, 3Com has offered firmware extensions to NICs that allowed them to do exactly this. In addition to avoiding the bottleneck imposed by current centralized approaches, NIC-based firewalls can also protect systems from some insider attacks as well as Internet attacks, since only the one host system is inside the NIC's boundary.

Most interestingly, a self-securing NIC can watch its own host system for signs of compromise, helping administrators to identify and contain compromised machines within their intranet. Given its closeness to its host, a NIC can have a much clearer view of its host's network interactions, more easily identifying denial-of-service attacks and propagation attempts, including many explicitly designed to defeat network intrusion detection systems. By throttling suspicious behavior, self-securing NICs can reduce damage and give administrators time to react—notably, rapid-propagation worms like Code Red and Slammer within the intranet can be quickly identified by self-securing NICs and throttled at their source. If all machines on the Internet were so equipped, such worms could not spread rapidly in the way that they do—our analyses [Ganger02, Ganger03a, Wong04a] indicate such throttling could extend the spread time from 15 minutes to over six months.

2.3.2 Storage Devices

The role of storage devices in computer systems is to persistently store data. Thus, natural security extensions are to watch the access stream for signs of misbehavior, keep a record of storage activity, and prevent compromised clients from destroying stored data.

Storage-based intrusion detection is a new approach that can spot several common intruder actions, such as adding backdoors, inserting Trojan horses, and removing traces of their presence [Pennington03]; as a simple form of this, some high-end file servers now come with built-in support for virus scanning. (A virus scanner in the file server cannot be disabled by viruses that compromised clients machines, like scan-

ners within those clients can.) Experiments show that most real intrusion tools (15 of the 18 studied) are quickly detected by a storage IDS.

A self-securing storage device can also protect stored data from attackers, preventing undetectable tampering and permanent deletion [Strunk00]. It does so by managing storage space from behind its security perimeter, keeping an audit log of all requests, and keeping previous versions of data modified by attackers. Since a storage device cannot distinguish compromised user accounts from legitimate users, the latter requires keeping all versions of all data. Finite capacities will limit how long such comprehensive versioning can be maintained, but 100% per year storage capacity growth will allow modern disks to keep several weeks of all versions. If intrusion detection mechanisms reveal an intrusion within this multi-week detection window, security administrators will have this valuable audit and version information for diagnosis and recovery. This information will enable diagnosis by not allowing system audit logs to be doctored, exploit tools to be deleted, or back doors to be hidden—the common steps taken by intruders to disguise their presence. This information will also simplify recovery by allowing rapid restoration of pre-intrusion versions and incremental examination of intermediate versions for legitimate updates. Finally, self-securing storage devices can be a critical component of survivable storage systems [Goodson04a, Wylie00].

2.3.3 Routers and Switches

The role of routers and switches in a network environment is to forward packets from one link to an appropriate next link. Thus, natural security extensions for such devices are traffic monitoring (e.g., attack detection), control (e.g., firewalls), and isolation (e.g., VLANs). Many current routers now provide such functions.

More advanced mechanisms combine the information and capabilities of multiple network switches and routers to enable distributed monitoring. Traffic monitoring at different vantage points allow us to detect anomalous traffic patterns and hence detect attacks at an early stage. For example, traffic monitoring on routers/switches in internal networks allows us to detect a compromised internal machine when it tries to probe and login to other machines that the compromised machine does not usually connect to. Such anomolous traffic patterns would not be caught using the traditional firewall approach because the traffic is soly internal. On the other hand, large scale Internet attacks such as distributed Denial-of-Service (DDoS) attacks and fast worm propagation often cause drastic changes in Internet traffic patterns, and traffic monitoring on the routers on the Internet could detect such traffic pattern change and hence the attacks at an early stage. While traffic monitoring on low bandwidth links or internal networks could use detailed profiling information, traffic monitoring on high bandwidth links and on the Internet often has stringent speed and memory requirements that require the design of more efficient and less fine-grained mechanisms to detect traffic pattern changes. For example, monitoring with per-flow state is not possible for high-bandwidth routers.

We have designed efficient algorithms to detect DDoS attacks and worm propagations at an early stage on high-speed routers [Newsome05, Venkataraman05, Yaar05, Yaar03]. Such traffic monitoring can not only detect attacks, but also automatically develop appropriate filters to stop or throttle attacks. For example, we have developed efficient traffic monitoring mechanisms to enable detection of the victims and attackers in DDoS attacks and compromised machines that try spread worms in real time. By dropping or delaying the traffic from attackers and to the victims, we could defend against or mitigate large scale Internet attacks. Traffic monitoring can not only be applied to the header information in the packets, but also to the content of the packets. Viruses and worms often have distinct signatures in the packet payload. Traffic monitoring can also be used to identify such signatures and throttle packets according to the signatures.

2.3.4 Virtual Machines

Following on our work, several groups have noted that virtual machine technology offers boundaries that can be as strong as physical isolation. A virtual machine monitor is a small piece of software that works

within a host's main CPU to provide "virtual machines" in which OSs and applications can execute. To the code within them, each virtual machine can look just like a complete hardware system, isolated from the other virtual machines. At the University of Michigan, researchers are exploring intrusion diagnosis based on complete machine replay based on a log of all input events (network packets, keystrokes, etc.). With such replay, a forensic analyst can replay and watch the actions of an intrusion in a controlled environment, examining what they did and how. At Stanford, researchers are exploring intrusion detection within a virtual machine monitor, using techniques similar to those described above, but with the extra benefit of being able to examine the host's main memory contents. Self-securing virtual machine monitors have all of the features of other self-securing devices, so long as the isolation implementation has no exploitable bugs.

2.3.5 User Interface Devices

An important emerging form of self-securing device is used for trusted computing. Controversial efforts like the TCPA (Trusted Computing Platform Alliance) and Microsoft's Palladium project seek to increase "trust" in computing: users should be able to trust the apparent source of output, software should be able to trust its runtime environment and the sources of input, and remote systems should be able to trust the sources of information. Each of these is largely impossible in today's systems, where digital intruders (or misbehaving users) can completely control and manipulate all software and data on their machines. Let's consider an example of each type of trust:

Users trusting output: an intruder who compromises your desktop computer can make it display anything it wants. So, for example, it can manipulate the stock quote, trick you into typing in your passwords or credit card numbers, or show you fake news stories. With trusted computing, output devices could be taught to ignore (or visibly tag) output that does not come from trusted sources.

Software trusting input: an intruder who compromises your desktop computer can manipulate (or fabricate) input signals conveyed to the software running on the system. So, for example, it could replay your password to authentication services to gain access to "secure data." Often-touted biometric sensors, which promise to distinguish between users based on measurements of their physical features, would do nothing to change this unless there is evidence (e.g., a digital signature) of the source and timeliness. Such evidence of when and where readings were taken is needed because, unlike passwords, biometrics are not secrets [Klosterman00]. For example, anyone can lift fingerprints from a laptop with the right tools or download facial images from a web page. Thus, the evidence is needed to prevent straightforward forgery and replay attacks.

Remote system trusting source: an identity thief who gains access to your personal information and any relevant passwords can use Internet services as you. Likewise, an intruder who compromises your desktop computer can conduct transactions using your credentials from your actual computer. A trusted computing component could assure remote systems that transactions come from you—they could ensure that the right software (which they trust) ran on your actual machine and that you actually typed commands into the machine (using the input trust discussed above). This form of trusted computing is the one most frequently discussed, because it makes e-commerce more secure by allowing service providers to trust that their software runs as desired on customer systems—in fact, the best product name we've heard for this is EMBASSY (from Wave Systems), evoking the image of a trusted outpost in a foreign land. It is also the most controversial form, since some worry that it will allow powerful corporations or governments to dictate the software running on everyone's computers.

The "secure coprocessors" often used to achieve trusted computing functions can also be used for intrusion detection and diagnosis functions.

2.4 Summary

This chapter promotes a type of security architecture in which traditional boundary protections are coupled with security functionality embedded into self-securing devices. The resulting collection of inde-

pendent security perimeters enables a flexible infrastructure for dynamic prevention, detection, diagnosis, isolation, and repair of successful intrusions. Self-securing devices have been shown a promising and viable approach to making digital infrastructures much more intrusion-tolerant.

3 SELF-SECURING NETWORK INTERFACES

3.1 Introduction to Self-Securing NIs

Multi-purpose computer systems are, and likely will remain, vulnerable to network intrusions for the foreseeable future. Implementers and administrators are unable to make them bulletproof, because the software is too big and complex, supports too many features and requirements, and involves too many configuration options. As a result, most network environments rely on firewalls and service proxies, with moderate success, to keep malicious parties from exploiting OS and service weaknesses. These special-purpose components observe and filter network traffic before it reaches the vulnerable systems. Usually placed at the boundary between a LAN and the "rest of the world," these components generally limit themselves to trivial filter rules.

More complete protection could be provided by extending individual network interfaces (NIs) to observe and even contain malicious network activity. Embedding such functionality in each machine's NI has a number of advantages over firewalls placed at LAN boundaries. First, distributing firewall functionality among end-points [Ioannidis00] avoids a central bottleneck and protects systems from local machines as well as the WAN. Second, a misbehaving host can be throttled at its source; as with firewalls, embedding checks and filtering into NIs [Friedman01] isolates them from vulnerable host software, preventing a successful intruder or malicious insider from disabling the checks. Third, and most exciting, each NI can focus on a single host's traffic, digging deeper into the lower-bandwidth, less noisy signal comprised of fewer aggregated communication channels. For example, reconstruction of application-level streams and inter-connection relationships becomes feasible without excessive cost. We refer to NIs extended with intrusion detection and containment functionality as *self-securing network interfaces*.

Counters to several common network attacks highlight the potential of self-securing NIs. For example, e-mail viruses can be contained much more effectively than the current approach of sending email warnings. Specifically, once a new virus is discovered, the administrator can update all self-securing NIs to identify propagation attempts to and from their hosts, prevent them, and identify machines already infected. As welll, the recent Code Red worm [CERT01a] (and follow-ons [CERT01b, Weaver01]) can be readily identified by the traffic pattern at a self-securing NI. Specifically, these worms spread exponentially by the abnormal behavior of targeting large numbers of randomly-chosen IP addresses with no corresponding DNS translations.

Digging deeply into network traffic, as promoted here, will greatly increase the codebase executing in an NI. Further, it will inevitably lead to less-expert and less-hardened implementations of scanning code, particularly code that examines the application-level exchanges. As a result, well-designed system software will be needed for self-securing NIs, both to simplify scanner implementations and to contain rogue scanners (whether buggy or compromised).

This chapter describes a software architecture that addresses both issues. A trusted NI kernel controls the flow of packets between the host interface and the network wire. Scanners, running as application processes, look for and possibly block suspicious network activity. With the right permissions, a scanner can subscribe to see particular traffic via a socket-like interface, pass or clip parts of that traffic, and inject pre-registered packets. The simple API should simplify scanner implementation. More importantly, the separation of power offers a critical degree of safety: while it can disrupt traffic flow, a rogue scanner cannot act arbitrarily as a man-in-the-middle.

We have built a prototype of this self-securing NI software architecture and a number of interesting scanners. The internal protection boundary between scanners and the trusted base comes with a reasonable cost. More importantly, our experiences indicate that the scanner API meets its programmer support goal. In particular, the network interaction part of writing scanner code is straightforward; the complexity is where it belongs: in the scanning functionality.

3.1.1 Finding and Containing the Enemies within the Walls

Traditional network security focuses on the boundary between a protected intranet and the Internet, with the assumption being that attacks come from the outside. The common architecture uses network intrusion detection systems (NIDSs) and firewalls to watch for and block suspicious network traffic trying to enter the protected intranet. This is a very appropriate first line of defense, since many digital attacks do indeed come from the wild. But, it leaves the protected intranet unprotected from an intruder who gets past the outer defenses.[2]

It is increasingly important to detect and contain compromised hosts, in addition to attempting to prevent initial compromise. Such containment offers a degree of damage control within the intranet, preventing one compromised system from infecting the others unchecked (NIDSs and firewalls are rarely used on the bulk of internal traffic). In addition, such containment is needed to reduce liability—as attack traceback techniques improve, increasing numbers of attacks may be traced to their apparent sources. Dealing with the aftermath of being blamed for attacks can be expensive (in money and time), even if the source machine in question is subsequently found to have been compromised.

A mechanism for identifying compromised hosts must have two features: (1) it must be able to observe potentially-malicious activities, and (2) it must not be trivially susceptible to being disabled. Host-based IDSs have the first feature but not the second; they are highly vulnerable to exactly the software whose soundness is being questioned, a fact illustrated by the many rootkits that disable host IDS checks. A NIDS at the edge of the intranet has the second feature but lacks much of the first; it can observe attacks launched outward but does not see attacks on other systems in the intranet. To also contain problem machines, a mechanism must have a third feature: (3) control over their access to the network.

We have extended the various network interfaces (NIs) within the intranet to look for and perhaps contain compromised hosts [Ganger02]. An NI is some component that sits between a host system and the rest of the intranet, such as a network interface card (NIC) or a local switch port. The NIs are isolated from the host OS, running distinct software on separate hardware, and are in the path of a host's network traffic. In addition, because they are in their host's path to the LAN, such NIs will see every packet, can fail closed, can isolate their host if necessary, and can actively normalize [Handley01, Malan00] the traffic. We refer to NIs extended with intrusion detection and containment functionality as *self-securing network interfaces*.

Self-securing NIs enjoy the scalability and coverage benefits of recent distributed firewall systems [Friedman01, Ioannidis00, 3Com01a]. They also offer an excellent vantage point for looking inward at a host and watching for misbehavior. In particular, many of the difficulties faced by NIDSs [Ptacek98] are avoided: there are no topology or congestion vagaries on an NI's view of packets moved to/from its host and there are no packets missed because of NI overload. This allows the NI to more accurately shadow important host OS structures (e.g., IP route information, DNS caches, and TCP connection states) and thereby more definitively identify suspicious behavior.

We discuss several examples of how the NI's view highlights host misbehavior. First, many of the NIDS attacks described by Ptacek and Newsham [Ptacek98] involve abusing TTLs, fragmentation, or TCP sequencing to give the NIDS a different view of sent data than the target host. From its local vantage, a self-securing NI can often tell when use of these networking features is legitimate. For example, IP fragmentation is important, but is only used by a well-behaved packet source in certain ways; a self-securing NI can shadow state and identify these. The IP Time-To- Live (TTL) field may vary among packets seen by a NIDS at an intranet edge, because packets may traverse varied paths, but should not vary in the original

[2] Our focus in this work is on network intruders who successfully gain access to an internal machine, as opposed to insiders who can exploit physical access to the hardware.

packets sent by a given host for a single TCP stream. After taking away most deception options, a self-securing NI (or other NIDS) can use traditional NIDS functionality to spot known attacks [Handley01].

Second, state-holding DoS attacks will be more visible, since the NI sees exactly what the host receives and sends. For example, the NI can easily tell if a host is ignoring SYN/ACK packets, as would be the case if it is participating in a "SYN bomb" DoS attack. Third, the random propagation approach used by the recent Code-Red worm [CERT01a] (and follow-ons [CERT03, CERT01b, Staniford02]) can be readily identified by the abnormal behavior of contacting large numbers of randomly-chosen IP addresses with no corresponding DNS translations. To detect this, a self-securing NI can shadow its host's DNS cache and check the IP address of each new connection against it.

3.2 Towards Per-machine Perimeters

The common network security approach maintains an outer perimeter (perhaps with a firewall, some proxies, and a NIDS) around a protected intranet. This is a good first line of defense against network attacks. But, it leaves the entire intranet wide open to an attacker who gains control of any machine within the perimeter. This section expands on this threat model, self-securing NIs, how they help, and their weaknesses.

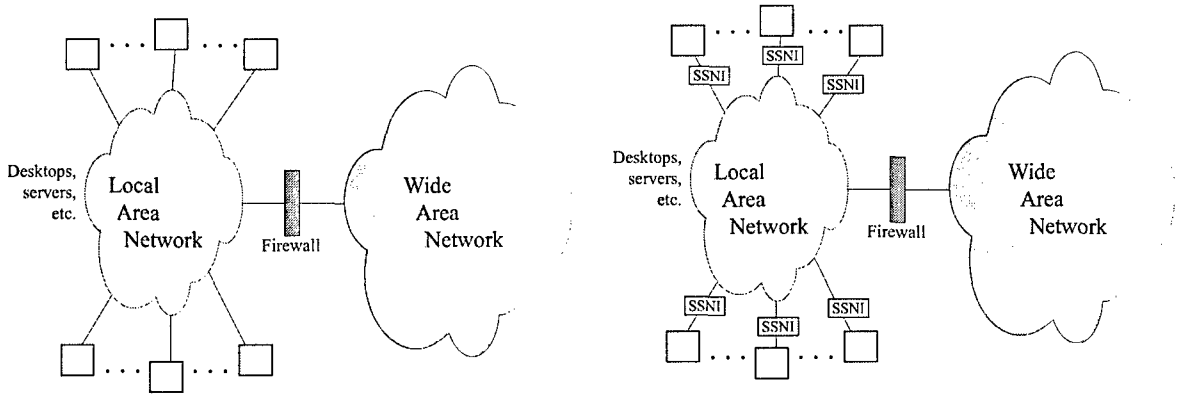
3.2.1 Threat Model

The threat with which we are most concerned here is a multi-stage attack. In the first stage, the attacker compromises any host on the inside of the intranet perimeter. By "compromises," we mean that the attacker subverts its software system, gaining the ability to run arbitrary software on it with OS-level privileges. In the second stage, the attacker uses the internal machine to attack other machines within the intranet.

This form of two-stage attack is of concern because only the first stage need worry about intranet-perimeter defenses; actions taken in the second stage do not cross that perimeter. Worse, the first stage need not be technical at all; an attacker can use social engineering, bribery, a discovered modem on a desktop, or theft (e.g., of a password or insecure laptop) for the first stage.

Finding a single hole is unlikely to be difficult in any sizable organization. Once internal access is gained, the second stage can use known, NIDS-detectable system vulnerabilities, since it does not enter the view of the perimeter defenses. In some environments, depending on their configuration, known attacks launched out of the organization may also proceed unhampered; this depends on whether the NIDS and firewall policies are equally restrictive in both directions.

Our main focus is on the second stage of such two-stage attacks. A key characteristic of the threat model described is that the attacker has software control over a machine inside the intranet, but does not have physical access to its hardware. We are not specifically trying to address insider attacks, in which the attacker would also have physical access to the hardware and its network connections. Clearly, for a self-securing NI to be effective, we must also assume that neither the administrative console nor the NI itself is compromised.



a) conventional security configuration b) addition of self-securing NIs

Figure 3.1.Self-securing network interfaces. (a) shows the common network security configuration, wherein a firewall protects LAN systems from some WAN attacks. (b) shows the addition of self-securing NIs, one for each LAN system.

3.3 Self-Securing NIs

A countermeasure to our two-stage attack scenario must have two properties: (1) it must be isolated from the system software of the first stage's target, since it would otherwise be highly vulnerable in exactly the situations we want it to function, and (2) it must be close to its host in the network path, or it will be unable to assist with intranet containment.[3] We focus on the network interface (NI) as a good place for countermeasures.

The role of the network interface (NI) in a computer system is to move packets between the system's components and the network. A *self-securing NI* additionally examines the packets being moved and enforces network security policies. Like a firewall, the security administrator can configure each self-securing NI to examine packet headers and simply not forward unacceptable packets into or out of its computer system. A self-securing NI can also dig deeper into traffic patterns, slow strange behavior, alert administrators of potential problems, etc. By embedding this traffic management functionality inside the NI, one enjoys its benefits even when the host OS or other machines inside the LAN border are compromised. This section makes a case for self-securing NIs by describing the system architecture in more detail and discussing its features.

3.3.1 Basic Architecture

A self-securing NI extends the NI's base functionality of transferring packets between the host software and the wire [Ganger01]. In most systems, this base functionality is implemented in a network interface card (NIC). For our purposes, the component labeled as "NI" will have three properties: (1) it will perform the base packet moving function, (2) it will do so from behind a simple interface with few additional services, and (3) it will be isolated (i.e., compromise independent) from the remainder of the host software. Examples of such NIs include NICs, DSL or cable modems, and NI emulators within a virtual machine monitor [Sugerman01]. As well, leaf switches have these properties for the hosts directly connected to them. The concepts and challenges of embedding the new functionality in an NI applies equally to each of these. We define such an NI to be a *self-securing NI* if it internally monitors and enforces policies on packets forwarded in each direction. No change to the host interface is necessary; these security functions

[3] Many of the schemes explored here could be also used at the edge to detect second-stage attacks from inside to out. Our goal of internal containment requires finer-grained perimeters. As discussed below, distributed firewalls can also protect intranet nodes from internal attacks, though they must be extended to what we call self-securing NIs in order to benefit from the containment and source-specific detection features described.

can occur transparently within the NI (except, of course, when suspicious activity is actively filtered). For traditional NIs, which exchange link-level messages (e.g., Ethernet frames), examination of higher-level network protocol exchanges will require reconstruction within the self-securing NI software. Although this work is redundant with respect to the host's network stack, it allows self-securing NIs to be deployed with no client software modification. For NIs that offload higher-level protocols (e.g., IP security or TCP) from the host [Cooper90, Dalton93], redundant work becomes unnecessary because the only instance of the work is already within the NI. Self-securing NIs enforce policies set by the network administrator, just as a centralized firewall would. In fact, administrators will configure and manage self-securing NIs over the network, since they must obviously be connected directly to it — doing so allows an administrator to use the NI to protect the network from its host system as well as the other way around. This approach also decouples the NI-enforced policies from the host software; even the host OS and its most privileged users should not be able to reconfigure or disable the NI's policies. Prior work provides solid mechanisms for remote policy configuration of this sort, and recent research [Arbaugh97, Bartal99, Friedman01, Ioannidis00] and practice [3Com01b, Miller96] clarifies their application to distributed firewall configuration. In addition to configuration over the network, alerts about suspicious activity will be sent to administrative systems via the network. The same secure channels used for configuration can be reused for this purpose. These administrative systems can log alerts, construct an aggregate view of individual NI observations, and notify administrators if so configured. As with any intrusion detection system, policy-setting administrators must balance the desire for containment with the damage caused by acting on false alarms. Self-securing NIs can watch for suspicious traffic and generate alerts transparently. But, if they are configured to block or delay suspicious traffic, they may disrupt legitimate user activity.

3.3.2 Why Self-securing NIs

The self-securing NI architecture described above has several features that combine to make it a compelling design point. This subsection highlights six. Two of them, scalability and full coverage, result from distributing the functionality among the endpoints [Ioannidis00]. Two, host independence and host containment, result from the NI being close to and yet separate from the vulnerable host software [Friedman01]. Two, less aggregation and more per-link resources, build on the scalability benefit but are worthy of independent mention.

Scalability. The work of checking network traffic is distributed among the endpoints. Each endpoint's NI is responsible for checking only the traffic to and from that one endpoint. The marginal cost of the required NI resources will be relatively low for common desktop network links. More importantly, the total available resources for examining traffic inherently scales with the number of nodes in the network, since each node should be connected to the network by its own self-securing NI. In comparison, the cost of equivalent aggregate resources in a centralized firewall configuration make them expensive (in throughput or dollars) or limit the checking that they can do. This argument is much like cost-effectiveness arguments for clusters over supercomputers [Anderson95].

Full coverage. Each host system is protected from all other machines by its self-securing NI, including those inside the same LAN. In contrast, a firewall placed at the LAN's edge protects local systems only from attackers outside the LAN. Thus, self-securing NIs can address some insider attacks in addition to Internet attacks, since only the one host system is inside the NI's boundary.

Host independence. Like network firewalls, self-securing NIs operate independently of vulnerable host software. Their policies are configured over the network, rather than by host software. So, even compromising the host OS will not allow an intruder to disable the self-securing NI functionality. (Successful intruders and viruses commonly disable any host-based mechanisms in an attempt to hide their presence.)

Host containment. Self-securing NIs offer a powerful control to network administrators: the ability to throttle network traffic at its sources. Thus, the LAN and its other nodes can be protected from a misbehaving host. For example, a host whose security status is questionable could have its network access slowed, filtered, or blocked entirely.

Less aggregation. Connected to only one host system, a self-securing NI investigates a relatively simple signal of network traffic. In comparison, a firewall at a network edge must deal with a noisier signal consisting of many aggregated communication channels. The clearer signal may allow a self-securing NI to more effectively notice strange network behavior.

More per-link resources. Because each self-securing NI focuses on only one host's traffic, more aggressive investigation of network traffic is feasible. Although this is really a consequence of the scalability feature, it is sufficiently important that we draw it out explicitly. Also, note that not all traffic must be examined in depth; for example, a self-securing NI could decide to examine e-mail and web traffic in depth while allowing NFS and Quake traffic to pass immediately. Each feature alone is valuable, which is why most other network security configurations include one or more of them. Self-securing NIs are a compelling addition, because they offer all of these features. Switches and routers within an organization's networking infrastructure share many of these features, though some (e.g., scalability and host containment) degrade the further one gets from the end systems. The following subsection provides some concrete examples of the potential benefits of these features.

3.3.3 Spotting and Containing Attacks

The most obvious use of the self-securing NI is to enforce standard firewall filtering rules [Cheswick94]. These rules typically examine a few fields of packet headers and allow only those for allowed network protocols to pass. Because the filtering occurs within an NI, it can also prevent basic spoofing (e.g., of IP addresses) and sniffing (e.g., by listening with the NI in "promiscuous mode") of network traffic. Previous researchers [Friedman01, Ioannidis00] have made a strong case for distributing such rules among the endpoints, and at least one product [3Com01a] has been put on the market.

Traditional firewall rules, however, barely scratch the surface of what can be done with self-securing NIs. The reduced aggregation and reduced link rate/usage make it possible to analyze more deeply the traffic seen. Examples include reconstructing and examining application-level exchanges, shadowing protocol state and examining state transitions, and shadowing host state and correlating inter-protocol relationships. Several concrete examples of network attacks that can be discovered and mitigated are described below.

E-mail viruses. One commonly observed security problem is the rapidly-disseminated email virus. Even after detecting the existence of a new virus, it often takes a significant amount of time to prevent its continued propagation. Ironically, the common approach to spreading the word about such a virus is via an e-mail message (e.g., "don't open unexpected e-mail that says 'here is the document you wanted'"). By the time a user reads the message, it is often too late. An alternative, enabled by self-securing NIs, is for the system administrator to immediately send a new rule to all NIs: check all in-bound and out-bound e-mail for the new virus's patterns.[4] E-mail exchanges generally conform to one of a small set of known protocols. Thus, a properly configured self-securing NI could scan attachments for known viruses before forwarding them. In many cases, this would immediately stop further spread of the virus within an intranet, as well as quickly identifying many of the infected systems. At the least, it would reduce the lifetimes of given e-mail viruses, which usually persist long after they are discovered and automated detection methods are available. Section 3.6.2 and Chapter 6 explores this in greater depth.

Buffer overflow attacks. The most common technique used to break into computer systems over the network is the buffer overflow attack. A buffer overflow attack exploits a particular type of programming error: failing to perform bounds checks and consequently writing past the end of a finite array in memory. The memory beyond the array often holds variables of importance to subsequent program execution. If an attacker knows of such a programming error, he may be able to send to the software messages that exploit

[4] Many sites route e-mail through a dedicated mail server, which makes it a natural site for this kind of checking (as long as scalability is not an issue). In general, dedicated proxies are a good place to check the associated application level exchanges. Self-securing NIs are a good place for such checks when dedicated proxies are not present, not checking, or not required and enforced (since an intruder does not have to conform to client configurations).

the error. Such attacks are particularly powerful when the overflowed buffer is allocated on the stack [One96]. A clever attacker with full program knowledge can carefully craft a stack overflow to rewrite the return address of the current procedure to point further up the stack and place code he wishes to execute at that location. With such an attack, a malicious party can run their own code with the permissions of the compromised application process (often "Administrator" or "root" for network services).

The infamous "Internet worm" of 1988 [Spafford89] exploited a known such weakness (in the `fingerd` application) and the recent Code Red worms did the same (in Microsoft's IIS server). In each case, the particular overflow attack was well-known ahead of time, but the software fixes were slow to appear and administrators remained vulnerable until the software owners provided patches. By having a self-securing NI look for network service requests that would trigger such overflows, one can prevent them from reaching the system until patches are provided.

SYN bombs. Another frequently cited network attack, called a "SYN bomb," exploits a characteristic of the state transitions within the TCP protocol [Postel80] to prevent new connections to the target. The attack consists of repeatedly initiating, but not completing, the three-packet handshake of initial TCP connection establishment, leaving the target with many partially completed sequences that take a long time to "time out". Specifically, an attacker sends only the first packet (a packet with the SYN flag set), ignoring the target's correct response (a second packet with the SYN and ACK flags set). This attack is difficult to deal with at the target machine, but a self-securing NI connected to the attacking machine (often a compromised host) can easily do so.

There are two variants of the SYN bomb attack. In one variant, the attacker uses its true address in the source fields, and the target's response goes to the attacker but is ignored. To detect this, a self-securing NI can simply notice that its host is not responding with the necessary final packet of the handshake (an ACK of the target's SYN flag). Upon detecting the problem, the NI could prevent continued "bombing" or even repair the damage itself (e.g., by sending an appropriate packet with the RST flag set to eliminate the partial connection). In the second variant, the attacker forges false entries in the initial packet's source fields, so that the target's reply goes to some other machine. A self-securing NI can prevent such spoofing easily, even if the host OS has been compromised.

Random, exponential spread (Code Red). A highly-visible network attack in 2001 was the Code Red worm (and follow-ons) that propagated rapidly once started, hitting most susceptible machines in the Internet in less than a day [Moore01]. Specifically, these worms spread exponentially by having each compromised machine target random 32-bit IP addresses. Extensions to this basic algorithm, such as hit list scanning and local subnet scanning, can reduce the propagation time to less than an hour [Weaver01]. Looking deeply at the network traffic, however, reveals an abnormal signature: contacting new IP addresses without first performing DNS translations. Although done occasionally, such behavior is uncommon, particularly when repeated rapidly. To detect this, a self-securing NI can shadow its one host's DNS table and check the IP address of each new connection against it. The DNS table can be shadowed easily, in most systems, since the translations pass through the NI. Section 3.6.3 explores this example in greater depth.

3.3.4 Costs and Limitations

Self-securing NIs are promising, but there is no silver bullet for network security. They can only detect attacks that use the network and, like most intrusion detection systems [Axelsson98], are susceptible to both false positives and false negatives. Containment of false positives yields denial of service, and failure to notice false negatives leaves intruders undetected. Also, until anomaly detection approaches solidify, only known attack patterns will be detected. Beyond these fundamental limitations, there are also several practical costs and limitations. First, the NI, which is usually a commodity component, will require additional CPU and memory resources for most of the attack detection and containment examples above. Although the marginal cost for extra resources in a low-end component is small, it is non-zero. Providers and administrators will have to consider the trade-off between cost and security in choosing which scanners to employ. Second, additional administrative overheads are involved in configuring and managing

self-securing NIs. The extra work should be small, given appropriate administrative tools, but again will be non-zero. Third, like any in-network mechanism, a self-securing NI cannot see inside encrypted traffic. While IP security functionality may be offloaded onto NI hardware in many systems, most application-level uses of encryption will make opaque some network traffic. If and when encryption becomes more widely utilized, it may reduce the set of attacks that can be identified from within the NI. Fourth, each self-securing NI inherently has only a local view of network activity, which prevents it from seeing patterns of access across systems. For example, probes and port scans that go from system to system are easier to see at aggregation points. Some such activities will show up at the administrative system when it receives similar alerts from multiple self-securing NIs. But, more global patterns are an example of why self-securing NIs should be viewed as complementary to edge-located protections. Fifth, for host-embedded NIs, a physical intruder can bypass self-securing NIs by simply replacing them (or plugging a new system into the network). The networking infrastructure itself does not share this problem, giving switches an advantage as homes for self-securing NI functionality.

3.4 Self-Securing NI Software Design

Self-securing NIs offer exciting possibilities for detecting and containing network security problems. But, the promise will only be realized if the required software can be embedded into network interfaces effectively. In particular, the proposed network traffic analyses will involve a substantial body of new software in the NI. Further, some of this software will need to be constructed and deployed rapidly in response to new network security threats. These characteristics will require an NI software system that simplifies the programming task and mitigates the dangers created by potentially buggy software running within the NI.

3.4.1 Goals

The overall goal of self-securing NIs is to improve system and network security. Clearly, therefore, they should not create more difficult security problems than they address. In addition, writing software to address new network security problems should not require excessive expertise. Other goals for NI-embedded software include minimizing the impact on end-to-end exchanges and minimizing the hardware resources required.

Containing compromised scanning code. As the codebase inside the NI increases, it will inevitably become more vulnerable to many of the same attacks as host systems, including resource exhaustion, buffer overflows, and so on. This fact is particularly true for code that scans application-level exchanges or responds to a new attack, since that code is less likely to be expertly implemented or extensively tested. Thus, a critical goal for self-securing NI software is to contain compromised scanning code. That is, the system software within the NI should be able to limit the damage that malicious scanning code can cause, working on the assumption that it may be possible for a network attacker to subvert it (e.g., by performing a buffer overflow attack).

Assuming that the scanning code decides whether the traffic it scans can be forwarded, malicious scanning code can certainly perform a denial-of-service attack on that traffic. Malicious scanning code also sees the traffic (by design) and will most likely be able to leak information about it via some covert channel. The largest concerns revolve around the potential for man-in-the- middle attacks and for effects on other traffic. Our main goal is to prevent malicious scanning code from executing these attacks: such code should not be able to replace the real stream with its own arbitrary messages and should not be able to read or filter traffic beyond that which it was originally permitted to control.

Reduced programming burden. We anticipate scanning code being written by non-experts (i.e., people who do not normally write NI software or other security-critical software). To assist programmers, the NI system software should provide services and interfaces that hide unnecessary details and reduce the burden. In the best case, programming new scanning code should be as easy as programming network applications with sockets.

Containing broken scanning code. Imperfect scanning code can fail in various ways. Beyond preventing security violations, it is also important to fault-isolate one such piece of code from the others. This goal devolves to the basic protection boundaries and bounded resource utilization commonly required in multi-programmed systems.

Transparency in common case. Although not a fundamental requirement, one of our goals is for self-securing NI functionality to not affect legitimate communicating parties. Detection can occur by passively observing network traffic as it flows from end to end. Active changes of traffic occur only when needed to enforce a containment policy.

Efficiency. Efficiency is always a concern when embedding new functionality into a system. In this case, the security benefits will be weighed against the cost of the additional CPU and memory resources needed in the NI. Thus, one of our goals is to avoid undue inefficiencies. In particular, non-scanned traffic should incur little to no overhead, and the system-induced overhead for scanned streams should be minimal. Comprehensive scanning code, on the other hand, can require as many resources as necessary to make their decisions — administrators can choose to employ such scanning code (or not) based on the associated trade-off between cost and security.

3.4.2 Basic Design Achieving These Goals

This section describes a system software architecture for self-securing NIs that addresses the above goals. As illustrated in Figure 3.2, the architecture is much like any OS, with a trusted kernel and a collection of untrusted applications. The trusted NI kernel manages the real network interface resources, including the host and network links. The application processes, called scanners, use the network API offered by the NI kernel to get access to selected network traffic and to convey detection and containment decisions. Administrators configure access rights for scanners via a secure channel.

Scanners. Non-trivial traffic scanning code is encapsulated into application processes called scanners. This allows the NI kernel to fault-isolate them, control their resource usage, and bound their access to network traffic. With a well-designed API, the NI kernel can also simplify the task of writing scanning code by hiding some unnecessary details and protocol reconstruction work. In this design, programming a scanner should be similar to programming a network application using sockets, both in terms of effort required and basic expertise required. (Of course, scanners that look at network protocols in detail, rather than application-level exchanges, will involve detailed knowledge of those protocols.)

Scanner interface. Table 3.1 lists the basic components of the network API exported by the NI kernel. With this interface, scanners can examine specific network traffic, alert administrators of potential problems, and prevent unacceptable traffic from reaching its target.

The interface has four main components. First, scanners can *subscribe* to particular network traffic, which asks the NI kernel for `read` and/or `contain` rights; the desired traffic is specified with a packet filter language [Mogul87]. The NI kernel grants access only if the administrator's configuration for the particularly scanner allows it. In addition to the basic packet capture mechanism, the interface should allow a scanner to subscribe to the data stream of TCP connections, hiding the stream reconstruction work in the NI kernel.

Second, scanners ask the NI kernel for more data via a *read* command. With each data item returned, the NI kernel also indicates whether it was sent by or to the host. Third, for subscriptions with `contain` rights, a decision for each data unit must be conveyed back to the kernel. The data unit can either be *pass*ed along (i.e., forwarded to its destination) or *cut* (i.e., dropped without forwarding). For a data stream subscription, *cut* and *pass* refer to data within the stream; in the base case, they refer to specific individual packets. For TCP connections, a scanner can also decide to *kill* the connection.

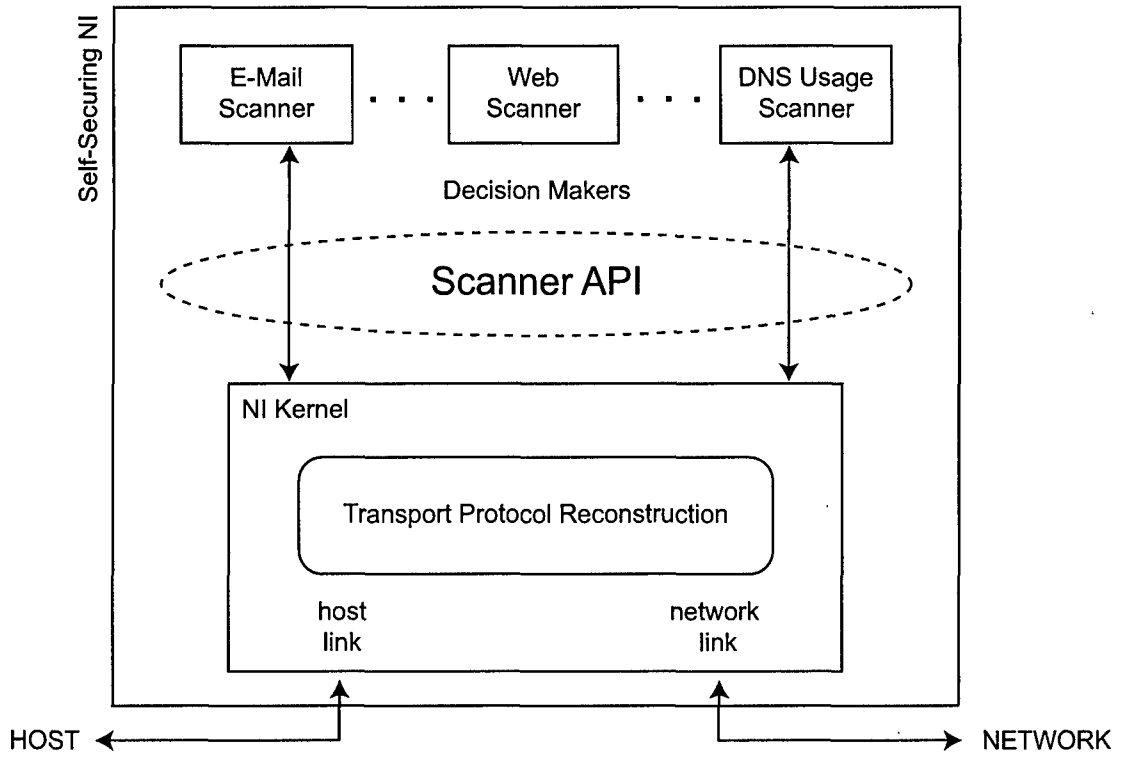


Figure 3.2: Self-securing NI software architecture. An "NI kernel" manages the host and network links. Scanners run as application processes. Scanner access to network traffic is limited to the API exported by the NI kernel.

Fourth, a scanner can *inject* pre-registered data into scanned communications, which may involve insertion into a TCP stream or generation of an individual packet. A scanner can also send an *alert*, coupled with arbitrary information or even copies of packets, to an administrative system.

The scanner interface simplifies programming, allows necessary powers, and yet restricts the damage a rogue scanner can do. A scanner can look at and gate the flow of traffic with a few simple commands, leaving the programmer's focus where it belongs: on the scanning algorithms. A scanner can ask for specific packets, but will only see what it is allowed to see. A scanner can decide what to pass or drop, but only for the traffic to which it has `contain` rights. A scanner can inject data into the stream, but only pre-configured data in its entirety. Combining *cut* and *inject* allows replacement of data in the stream, but the pre-configured *inject* data limits the power that this conveys. Alerts can contain arbitrary data, but they can only be sent to a pre-configured administrative system.

NI Kernel. The NI kernel performs the core function of the network interface: moving packets between the host system and the network link. In addition, it implements the functionality necessary to support basic scanner (i.e., application) execution and the scanner API. As in most systems, the NI kernel owns all hardware resources and gates access to them. In particular, it bounds scanners' hardware usage and access to network traffic.

Packets arrive in NI buffers from both sides. As each packet arrives, the NI kernel examines its headers and determines whether any subscriptions cover it. If not, the packet is immediately forwarded to its destination. If there is a subscription, the packet is buffered and held for the appropriate scanners. After each `contain`-subscribing scanner conveys its decision on the packet, it is either dropped (if any say drop) or forwarded.

Command	Description
Subscribe	Ask to scan particular network data
Read	Retrieve more from subscribe buffers
Pass	Allow scanned data to be forwarded
Cut	Drop scanned data
Kill	Terminate the scanned session (if TCP)
Inject	Insert pre-registered data and forward
Alert	Send an alert message to administrator

Table 3.1: Network API exported to scanner applications. This interface allows an authorized scanner to examine and block specific traffic, but bounds the power gained by a rogue scanner. Pass, cut, kill and inject can only be used by scanners with both read and contain rights.

NI kernels should also reconstruct transport-level streams for protocols like TCP, to both simplify and limit the power of scanners that focus on application-level exchanges. Such reconstruction requires an interesting network stack implementation that shadows the state of both endpoints based on the packets exchanged. Notice that such shadowing involves reconstructing two data streams: one in each direction. When a scanner needs more data than the TCP window allows, indicated by blocking *read*s from a scanner with pending decisions, the NI kernel must forge acknowledgement packets to trigger additional data sent from endpoints. In addition, when data is cut or injected into streams, all subsequent packets must have their sequence numbers adjusted appropriately.

Administrative interface. The NI's administrative interface serves two functions: receiving configuration information and sending alerts. (Although we group them here, the two functions could utilize different channels.)

The main configuration information is scanner code and associated access rights. For each scanner, provided access rights include allowed subscriptions (`read` and `contain`) and allowed injections. When the NI kernel starts a new scanner, it remembers both, preventing a scanner from subscribing to any other traffic or injecting arbitrary data (or even other scanners' allowed injections).

When requested by a scanner, the NI kernel will send an alert via the administrative interface. Overall, scanner transmissions are restricted to the allowed injections and alert information sent to pre-configured administrative systems.

3.4.3 Discussion

Implemented properly, we believe this design can meet the goals for self-securing NIs. Our experiences indicate that writing scanners is made relatively straightforward by the scanner API. Moreover, restricting scanners to this API bounds the damage they can do. Certainly a scanner with `contain` rights can prevent the flow of traffic that it scans, but its ability to prune other traffic is removed and its ability to manipulate the traffic it scans is reduced. A scanner with `contain` rights can play a limited form of man-in-the-middle by selectively utilizing the *inject* and *cut* interfaces. The administrator can minimize the danger associated with *inject* by only allowing distinctive messages. (Recall that *inject* can only add pre-registered messages and in their entirety. Also, a scanner cannot *cut* portions of *inject*ed data.) In theory, the ability to transparently *cut* bytes from a TCP stream could allow a rogue scanner to rewrite the stream arbitrarily. Specifically, the scanner could clip bytes from the existing stream and keep just those that form the desired message. In practice, we do not expect this to be a problem; unless the stream is already close to the desired output, it will be difficult to construct the desired output without either breaking something or being obvious (e.g., the NI kernel can be extended to watch for such detailed clipping patterns). Still, small *cut*s (e.g., removing the right "not" from an e-mail message) could produce substantial

changes that go undetected. Although scanners still have some undesirable capabilities, we believe that the NI software architecture described is a significant improvement over unbounded access.

3.5 Implementation

This section describes a prototype implementation of the self-securing NI software architecture described in Section 3.4.

3.5.1 Overview

The prototype self-securing NI is actually an old PC (referred to below as the "NI machine") with two Ethernet cards, one connected to the real network and one connected point-to-point to the host machine's Ethernet link. Figure 3.3 illustrates the hardware setup. Clearly, the prototype hardware characteristics differ from real NIC hardware, but it does allow us to explore the system software issues that are our focus in this work.

Our software runs on the FreeBSD 4.4 operating system. Both network cards are put into "promiscuous mode," such that they grab copies of all frames on their Ethernet link; this configuration allows the host machine's real Ethernet address to be used for communication with the rest of the network. Our NI kernel, which we call *Siphon*, runs as an application process and uses BPF [McCanne93] to acquire copies of all relevant frames arriving on both network cards. Frames destined for the NI machine flow into its normal in-kernel network stack. Frames to or from the host machine go to Siphon. All other frames are dropped. Scanners also run as application processes, and they communicate with Siphon via named UNIX sockets. Datagram sockets are used for getting copies of frames, and stream sockets are used for reconstructed data streams. (In OS architecture terms, FreeBSD is being used as a microkernel with Siphon as an "NI kernel server.")

3.5.2 Scanner interface implementation

Functionally, our implemented scanner interface matches the one described in Section 3.3 and illustrated in Table 3.1. The structure of our prototype, however, pushes for a particular style in the interface. Scanners communicate with Siphon via sockets, receiving subscribed-to traffic via READ and passing control information via WRITE. This section details the interactions for both frame-level scanning and reconstructed-stream scanning.

Frame-level scanning interface. Scanners can see and make decisions on raw Ethernet frames via the frame-level scanning interface, which is a datagram socket connected to Siphon.

Frames that match any of a scanner's frame-level subscriptions are written by Siphon to that scanner's socket. For each successful READ call on the socket, a scanner gets a small header and a received frame. The header indicates the frame's length and whether it came from the host or from the network. In addition, each frame is numbered according to how many previous frames the scanner has READ: the first frame read is #1, the second frame is #2, and so on.

A frame-level scanner conveys decisions on scanned frames via WRITEs to the socket, each consisting of a frame number and the decision (*cut* or *pass*). A scanner conveys its requests via this interface as well. *Inject* requests specify which pre-registered packet should be sent (via an index into a per-scanner table) and in which direction. *Alert* requests provide the message that should be sent to the administrative system. *Subscribe* requests ask for additional frames to be seen via the same socket; the desired frames are described via a sequence of <offset,value> pairs, much like most packet filter languages [Mogul87].

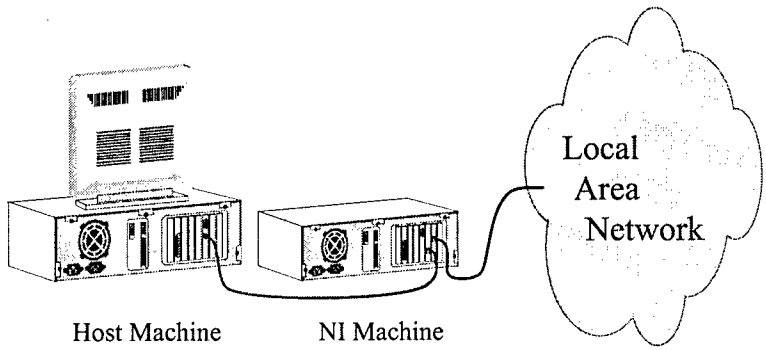


Figure 3.3: Self-securing NI prototype setup. The prototype self-securing NI is an old PC with two network cards, one connected directly to the host machine and one connected to the network.

Reconstructed-stream scanning interface. Scanners use stream sockets to get reconstructed TCP streams from Siphon, with a similar interface to that described above. The differences are changes to some parameters, two new requests, and a way of attaching to new connections as they are established. The main parameter changes relate to how scanned data is identified: *cut* and *pass* decisions apply to a byte offset and length within the stream in a particular direction. As well, *inject* requests must specify the byte offset at which the pre-registered data should be inserted into the stream (shifting everything after it forward by length bytes). Finally, *subscribe* requests simply specify inbound and outbound TCP port numbers (or wildcards) on which to listen.

The two new requests are *kill*, which tells the NI kernel to terminate the connection being scanned, and *more*, which tells the NI kernel that more data is necessary before a decision can be made. The *more* request is needed because our application-level NI kernel cannot actually see a scanner blocking on a READ request, as it would if implemented as a true kernel. The NI kernel must know about the need for more data, since it may need to offer extra space in the TCP window to trigger additional data transmission.

For reconstructed-stream scanning, several sockets are required. One is used to convey subscription requests. A second listens for new connections from Siphon. An ACCEPT on this second connection creates a new socket that corresponds to one newly established TCP connection between the host machine and some other system. READs and WRITEs to such new connections receive data to be scanned and convey decisions and requests.

3.5.3 The NI kernel: Siphon

Siphon performs the basic function of a network interface, moving packets between the host and the network. It also exports the scanner API described above.

During initialization, Siphon sets up a BPF interface from which it can READ all frames sent to or from the host machine. Each frame is buffered and passed through a packet filter engine. If the frame does not match any of the packet filter rules, it is immediately forwarded to its target (either the host machine or the network link). There are three types of packet filter rules: *prevent*, *scan*, and *reconstruct*. If the frame matches a *prevent* rule, it is dropped immediately; *prevent* rules provide traditional firewall filtering without the overhead of an application-level scanner. If the frame matches a *scan* rule, it is written to the corresponding scanner's datagram socket. If the frame matches a *reconstruct* rule, it is forwarded to the TCP reconstruction code. For frames that match *scan* and *reconstruct* rules for subscriptions with `contain` rights, Siphon keeps copies and remembers the decisions that it needs. A frame is forwarded if and only if all subscribed scanners decide *pass*; otherwise, it is dropped.

Siphon's TCP reconstruction code translates raw Ethernet frames into the reconstructed-stream interface described above. While doing so, it tries to minimize the perturbation on an end-to-end exchange.

In the common case, Siphon can reconstruct TCP streams by just watching the packets that go by. Upon seeing the first packet of a new TCP connection that matches a subscription, Siphon creates two protocol

control blocks, one to shadow the state of each end-point. Each new packet indicates a change to one end-point or the other. When the connection is fully established, Siphon opens and CONNECTs a stream socket to each subscribed scanner. When one side tries to send data to the other, that data is first given to subscribed scanners. If all such scanners with `contain` rights decide *pass*, the original packets are forwarded. When the TCP connection closes, Siphon CLOSEs the corresponding stream socket.

Once a scanner asks for an active change to the stream, Siphon can no longer just passively delay, reconstruct from, and then forward frames for the corresponding TCP connection; it must now modify some of them. If a scanner asks for some data to be *cut*, Siphon must prune that data from the original packets; doing so requires changes to several TCP and IP fields, and it may require splitting one packet into two. In addition, Siphon must send acknowledgements for the *cut* data once all bytes up to it have been acknowledged by the true receiver. Finally, the sequence numbers and acknowledgements of subsequent packets must be adjusted to account for the *cut* data.

A similar set of active changes are needed for *inject*. Siphon must create, forward, and buffer packets for the injected data, retransmitting as necessary. Subsequent packets must have their sequence numbers and acknowledgements adjusted, and one original packet may have to be split in two if its data spans the *inject* point.

More requests require less work. For small amounts of additional data, the TCP window can be opened further to get the sender to provide more data. Otherwise, Siphon must forge acknowledgements to the source and then handle retransmissions to the destination. In this case, Siphon must also drop redundant acknowledgements from the receiver.

Kill requests are handled by forging packets with the RST flag set and sending one to each end-point.

3.5.4 Issues

Our prototype focuses on exploring the scanner API, and it does set aside two important implementation issues: administrative interface and bounded resource utilization. For a full implementation, one would employ well-established technologies for both issues. Although it is always a dangerous claim, we do not see either of these issues invalidating the experiences or results arising from use of our prototype.

The administrative interface for the prototype consists of a directly-connected terminal interface. Clearly, this is not appropriate for practical management of per-host self-securing NIs. Fortunately, well-established cryptography-based protocols [3Com01b, Arbaugh97, Bartal99, Friedman01, Ioannidis00, Miller96] exist for remotely distributing policy updates and receiving alerts.

This prototype also does not preclude scanners from excessive resource utilization, instead relying on the underlying FreeBSD kernel to timeshare. A real NI kernel implementation would need to explicitly prevent any scanner from using too many resources. With each scanner as a single process with no I/O requirements, such resource management should be relatively straightforward for an OS kernel.

3.6 Evaluation

The main goal of our prototype is to allow us to experiment with NI-embedded scanners. Although it is too early to draw definitive conclusions, we believe that its software architecture is valuable. Our experiences indicate that the scanner API makes writing scanners relatively straightforward, though it could be made more so with Bro-like language support [Nessett98] at the scanner level. More importantly, restricting scanners to this API bounds the damage they can do. Certainly, a scanner with `contain` rights can prevent the flow of traffic that it scans, but its ability to prune other traffic is removed and its ability to manipulate the traffic it scans is reduced.

A scanner with `contain` rights can play a limited form of man-in-the-middle by selectively utilizing the *inject* and *cut* interfaces. The administrator can minimize the danger associated with *inject* by only allowing distinctive messages. (Recall that *inject* can only add pre-registered messages in their entirety. Also, a scanner cannot *cut* portions of *injected* data.) In theory, the ability to transparently *cut* bytes from a TCP stream could allow a rogue scanner to rewrite the stream arbitrarily. Specifically, the scanner could

clip bytes from the existing stream and keep just those that form the desired message. In practice, this is not a problem; unless the stream is already close to the desired output, it will be difficult to construct the desired output without either breaking something or being obvious (e.g., the NI kernel can be extended to watch for such detailed clipping patterns). Still, small *cuts* (e.g., removing the right "not" from an e-mail message) could produce substantial changes that go undetected.

This section evaluates our scanner API and the Siphon prototype. It does so via two very different scanners, exploring how the interface supports their construction. The first scanner examines application-level exchanges for known problems (e-mail viruses). The second scanner looks for a particular suspicious activity (random IP-based propagation) at the network protocol level. Both are easily implemented given Siphon's scanner API.

3.6.1 Basic Overheads

Although its support for scanners is our focus, it is useful to start with Siphon's effect on NI throughput and latency. For all experiments in this chapter, the NI machine runs FreeBSD 4.4 and is equipped with a 300MHz Pentium II, 128MB of main memory, and two 100Mb/s Ethernet cards. After subtracting the CPU power used for packet management functions that could be expected to be hardware-based, we believe that this dated system is a reasonable approximation of a feasible NI. The host machine runs FreeBSD 4.4 and is equipped with a 1.4GHz Pentium III, 512MB of main memory, and a 100Mb/s Ethernet card. Although Siphon is operational, little tuning has been done.

Configuration	Roundtrip	Bandwidth
No NI machine	0.16 ms	11.11 MB/s
No scanners	0.23 ms	11.11 MB/s
Frame scanner	0.23 ms	11.08 MB/s
Stream scanner	0.23 ms	10.69 MB/s

Table 3.2: Base performance of our self-securing NI prototype. Roundtrip latency is measured with 20,000 pings. Throughput is measured by RCPing 100MB. "No NI machine" corresponds to the host machine with no selfsecuring NI in front of it. "No scanners" corresponds to Siphon immediately passing on each packet. "Frame scanner" corresponds to copying all IP packets to a read-only scanner. "Stream scanner" corresponds to reconstructing the TCP stream for a read-only scanner.

Table 3.2 shows results for four configurations: the host machine alone (with no NI machine), the NI machine with no scanners, the NI machine with a read-only frame-level scanner matching every packet, and the NI machine reconstructing all TCP streams for a read-only scanner. We observe a 47% increase in round-trip latency with the insertion of the NI machine into the host's path, but no additional increase with scanners. We observe minimal bandwidth difference among the four configurations, although reconstructing the TCP stream results in a 4% reduction.

3.6.2 Example: E-mail Virus Scanner

A promising activity for self-securing NIs is to examine application-level exchanges for known problems, such as viruses or buffer overflows. As a concrete example, this section describes and evaluates a scanner that examines e-mail traffic.

What the scanner looks for: Commonly, viruses that propagate via e-mail do so in the form of infected attachments. For example, the attachment may be a malicious script or a complex file format (e.g., Excel or Word) with a malicious macro. Our scanner parses incoming and outgoing e-mail messages to identify attachments, which are then passed to virus checking code. By updating the virus checking code, an administrator can immediately identify subsequent attempts to propagate a known virus.

How the scanner works: There are several e-mail protocols. This scanner focuses on two common protocols: Post Office Protocol (POP) and Simple Mail Transport Protocol (SMTP). SMTP [Postel81] is used to send e-mail from one machine to another. POP [Rose88] is used to update a replica of one's main mailbox on a second machine. Both protocols function such that e-mail messages are transfered in their entirety, including all attachments, over a TCP connection. Attachments are encoded in MIME format [Freed96].

The e-mail scanner *subscribes* with `contain` rights to reconstructed TCP streams corresponding to the default port numbers for these protocols (25 for SMTP and 110 for POP). Any communications other than e-mail messages are *pass*ed immediately. When the start of an e-mail message is detected, the scanner waits until it has the entire message before making its decision. If necessary, the scanner uses the *more* request to tell Siphon that it needs more data from the stream. The scanner then parses the message, decodes each attachment, and passes it to the virus checking code. If no viruses are detected, the scanner tells Siphon to *pass* the entire e-mail message.

The current scanner can use either of two virus checking mechanisms. The first is a simple table lookup, in which the MD5 hash is compared to a list of known malicious attachments. This check is time- and space-efficient, but it will only identify non-mutating e-mail viruses. The second is the Sophos Anti-Virus library [Sophos02b]. This library uses modern scanning algorithms, with regular updates, to identify the wide variety of known viruses and even some virus-like signatures.

Configuration	Per-message latency
No scanner	22.4 ms
Null scanner	67.0 ms
Scanner w/MD5	107.1 ms
Scanner w/Sophos	109.9 ms

Table 3.3: Message latency with the e-mail scanner. The average per-message latency is for one pass through a month's worth of e-mail. Each value is an average of three iterations, and all standard deviations are less than 1% of the mean.

When a virus is detected: The scanner does not allow infected attachments to be forwarded. It tells Siphon to *cut* the range of bytes making up the attachment, to *inject* a replacement attachment at the original offset, and to *pass* the remainder of the e-mail message. The pre-registered replacement attachment (MIME-encoded as "text/plain") informs the recipient that an infected attachment was removed. In addition, the original message is sent to the administrative machine in an *alert*, for subsequent analysis. This also ensures that the message data is not lost, since it is possible that the attachment was flagged in error.

Performance data: We investigate the performance overhead involved with our e-mail scanner by using POP to transfer one month's worth of the first author's e-mail (1500 e-mail messages, with an average size of 8240 bytes and a maximum size of 776KB). Transferring the full set of messages at maximum speed, we measure the average time per message for four configurations: no scanner, a null scanner, the e-mail scanner using MD5, and the e-mail scanner using the Sophos library. Table 3.3 shows the results.

The results indicate that the scanner will delay e-mail messages. We observe substantial overhead for parsing the e-mail and exchanging information with Siphon. Much of this is due to an unoptimized scanner. As an anecdotal experiment in scanner programmability, this scanner was written by a recent B.S. graduate who learned POP, SMTP, and socket programming for this project. He observed that constructing an operational e-mail scanner was not difficult, and he reused functions for parsing e-mail, decoding MIME enclosures, and buffering data read from sockets.

Fortunately, the impact of slowed e-mail on user experiences should be minimal. Delaying e-mail delivery by a small amount is unlikely to be noticed. Throughput should also not be a problem, since even a popular user usually gets fewer than 100 messages in a day.

Discussion: Implementing an e-mail scanner for POP and SMTP was straightforward using the scanner API; the scanner simply watches for the beginning of an e-mail message and then examines that message. For more interactive mail exchange protocols, such as IMAP [Crispin96], additional effort will be required. In particular, IMAP transfers e-mail messages in pieces rather than as a whole, and those pieces are not self-identifying. An IMAP scanner will have to track the exchanges to identify when an attachment is being transferred so that it can invoke the virus scanner. Although this requires more application-level logic in the scanner, we do not expect it to be difficult.

3.6.3 Example: Code-Red Scanner

Another promising activity for self-securing NIs is to look for suspicious activity at the network protocol level, such as unanswered SYN+ACK packets, incomplete IP fragments, and unexpected IP addresses. A highly-visible network attack in 2001 was the Code-Red worm (and its follow-ons) that propagated rapidly once started, hitting most susceptable machines in the Internet in less than a day [Moore01].As a concrete example, this section describes and evaluates a scanner that watches for the Code-Red worm's abnormal network behavior.

What the scanner looks for: The Code-Red worm and follow-ons spread exponentially by having each compromized machine target random 32-bit IP addresses. This propagation approach is highly effective because the IP address space is densely populated and relatively small. Although done occasionally, it is uncommon for a host to connect to a new IP addresses without first performing a name translation via the Domain Name System (DNS) [Mockpetris88]. Our scanner watches DNS translations and checks the IP addresses of new connections against them. It flags any sudden rise in the count of "unknown" IP addresses as a potential problem.

How the scanner works: The Code-Red scanner consists of two parts: shadowing the host machine's DNS table and checking new connections against it. Upon initialization, therefore, the scanner *subscribes* to three types of frames. The first two specify UDP packets sent by the host to port 53 and sent by the network from port 53; port 53 is used for DNS traffic.[5] The third specifies TCP packets sent by the host machine with only the SYN flag set, which is the first packet of TCP's connection-setup handshake. Of these, only the third subscription includes `contain` rights.

Each DNS reply can provide several IP addresses, including the addresses of authoritative name servers. When it *reads* a DNS reply packet, the scanner parses it to identify all provided IP addresses and their associated times to live (TTLs). The TTL specifies for how long the given translation is valid. Each IP address is added to the scanner's table and kept at least until the TTL expires. Thus, the scanner's table should contain any valid translations that the host may have in its DNS cache. The scanner prunes expired entries only when it needs space, since host applications may utilize previous results from gethostbyname() even after the DNS translations expire.

The scanner checks the destination IP addresses of the host machine's TCP SYN packets against this table. If there is a match, the packet is *passed*. If not, the scanner considers it a "random" connection. The current policy flags a problem when there are more than two unique random connections in a second or ten in a minute.

When an attack is detected: The scanner's current policy reacts to potential attacks by sending an alert to the administrative system and slowing down excessive random connections. It stays in this mode for the next minute and then re-evaluates and repeats if necessary. The *alert* provides the number of random connections over the last minute and the most recent port to which a connection was opened. Random connections are slowed down by delaying decisions; in attack reaction mode, the scanner tells Siphon *pass* for one of the SYN packets every six seconds. This allows such connections to make progress, somewhat balancing the potential for false positives with the desire for containment. If all susceptible

[5] Although we see none in our networks, DNS traffic can be passed on TCP port 53 as well. Our current scanner will not see this, but could easily be extended to do so.

hosts were equipped with self-securing NIs, this policy would have increased the 14 hour propagation time of Code-Red (version 2) [Moore01] to over a month (assuming the original scan rate was 10/second per infected machine [Weaver01]).

Two extensions to the current scanner are under consideration. First, the scanner can log the exchanges of random connections via the *alert* interface, allowing an administrator to study them at her convenience. Second, when the rate of random connections is very high, SYN packets can simply be dropped; this will prevent connections from being established, but can also cause harm given a false positive.

Performance data: We evaluate two aspects of Code-Red scanner performance: its effect on latency and the required DNS table size. To evaluate the scanner's effect on DNS translation latency, we measured the times for 100 different translations with and without the scanner. The results indicate that the scanner increases the translation latency by 99% (from 1.5ms to 2.9ms) compared to having no NI machine. Since DNS translations and SYN packets are minor parts of overall network activity, we believe that the increased latencies due to scanning would have negligible impact on performance.

We evaluate the table sizes needed for the Code-Red scanner by examining a trace of all DNS translations for 10 desktop machines in our research group over 2 days. Assuming translations are kept only until their TTL's expire, each machine's DNS cache would contain an average of 209 IP addresses. The maximum count observed was 293 addresses. At 16 bytes per entry (for the IP address, the TTL, and two pointers), the DNS table would require less than 5KB. (We also observed that the average latency over the 700,000 translations was 1.6ms, very close to our baseline "no NI machine" measurement above.)

It is interesting to consider the table size required for an aggregate table. As a partial answer, we observe that a combined table for the 10 desktops would require a maximum of 750 entries (average of 568) or 12KB. This matches the results of a recent DNS caching study [Jung01], which finds that caches shared among 5 or more systems exhibit a 80–85% hit rate. They found that aggregating more client caches provides little additional benefit. Thus, one expects an 80–85% overlap among the caches, leaving 15–20% of the entries unique per cache. Thus, 10,000 systems with 250 entries each would yield approximately 375,000–500,000 unique entries (6MB–8MB) in a combined table.

Discussion: The largest danger of the Code-Red scanner is that other mechanisms could be used (legitimately) for name translation. There are numerous research proposals for such mechanisms [Stoica01, Rowstron01, Zhao01], and even experimenting with them would trigger our scanner. Administrators who wish to allow such mechanisms in their environment would need to either disable this scanner or extend it to understand the new name translation mechanisms.

With a scanner like this in place, different tactics will be needed for worms to propagate without being detected quickly. One option is to slow the scan rate and "fly under the radar," but this dramatically reduces the propagation speed, as discussed above. Another approach is to use DNS's reverse lookup support to translate random IP addresses to names, which can then be forward translated to satisfy the scanner's checks. But, extending the scanner to identify such activity would be straightforward. Yet another approach would be to explore the DNS name space randomly[6] rather than the IP address space; this approach would not enjoy the relevant features of the IP address space (i.e., densely populated and relatively small). There are certain to be other approaches as well. The scanner described takes away a highly convenient and effective propagation mechanism; worm writers are thus forced to expend more effort and/or to produce less successful worms. Security is a "game" of escalation, and self-securing NIs arm those in the white hats.

Finally, it is worth noting that all of the Code-Red worms exploited a particular buffer overflow that was well-known ahead of time. An HTTP scanner could easily identify requests that attempt to exploit it and prevent or flag them. The DNS-based scanner, however, will also spot worms, such as the Nimda worm,

[6] The DNS "zone transfer" request could short-circuit the random search by acquiring lists of valid names in each domain. Many domains disable this feature. Also, self-securing NIs could easily notice its use.

that use random propagation but other security holes. Coincidentally, information about the "SQL Slammer" worm [CERT03] indicates that it would be caught by this same scanner.

3.6.4 Detecting Claim-and-hold DoS Attacks

Qie et al. [Qie02] partition DoS attacks into two categories: busy attacks (e.g., overloading network links) and claim-and-hold attacks. In the latter, the attacker causes the victim to allocate a limited resource for an extended period of time. Examples include filling IP fragment tables (by sending many "first IP fragment" frames), filling TCP connection tables (via "SYN bombing"), and exhausting server connection limits (via very slow TCP communication [Qie02]). A host doing such things can be identified by its self-securing NI, which sees what enters and leaves the host when. As a concrete example, this section describes a scanner for SYN bomb attacks.

What the scanner looks for: A SYN bomb attack exploits a characteristic of the state transitions within the TCP protocol [Postel80] to prevent new connections to the victim. The attack consists of repeatedly initiating, but not completing, the three-packet handshake of initial TCP connection establishment, leaving the target with many partially completed sequences that take a long time to "time out." Specifically, an attacker sends only the first packet (with the SYN flag set), ignoring the victim's correct response (a second packet with the SYN and ACK flags set). The scanner watches for instances of inbound SYN/ACK packets not receiving timely responses from the host. A well-behaved host should respond to a SYN/ACK with either an ACK packet (to complete the connection) or a RST packet (to terminate an undesired connection).

How the scanner works: The scanner watches all inbound SYN/ACK packets and all outbound ACK and RST packets. It works by maintaining a table of all SYN/ACKs destined to the host that have not yet been answered. Whenever a new SYN/ACK arrives, it is added to the 'waiting for reply' table with an associated timestamp and expiration time. Retransmitted SYN/ACKs do not change these values. If a corresponding RST packet is sent by the host, the entry is removed. If a corresponding ACK packet is sent, the entry is moved to a 'reply sent' cache, whose role is to identify retransmissions of answered SYN/ACK packets, which may not require responses; entries are kept in this cache until the connection closes or 240 seconds (the official TCP maximum roundtrip time) passes.

If no answer is received by the expiration time, then the scanner considers this to be an ignored SYN/ACK. Currently, the expiration time is hard-coded at 3 seconds. The current policy flags a problem if there are more than two ignored SYN/ACKs in a one minute period.

When an attack is detected: The SYN bomb scanner's current policy reacts to potential attacks only by sending an alert to the administrative system. Other possible responses include delaying or preventing future SYN packets to the observed victim (or all targets) or having Siphon forge RST packets to the host and its victim for the incomplete connection (thereby clearing the held connection state).

Performance data: The SYN bomb scanner maintains a histogram of the observed response latency of its host to SYN/ACK packets. Under a moderate network load, over a one hour period of time, a desktop host replied to SYN/ACKs in an average of 26 milliseconds, with the minimum being under 1 and the maximum being 946 milliseconds. Such data indicates that our current grace period of 3 seconds should result in few false positives.

Discussion: There are two variants of the SYN bomb attack, both of which can be handled by self-securing NIs on the attacking machine. In one variant, the attacker uses its true address in the source fields, and the victim's responses go to the attacker but are ignored. This is the variant targeted by this scanner. In the second variant, the attacker forges false entries in the SYN packets' source fields, so that the victim's replies go to other machines. A self-securing NI on the attacker machine can prevent such spoofing.

3.6.5 Detecting TTL Misuse

Crafty attack tools can hide from NIDSs in a variety of ways. Among them are insertion attacks [Paxson98] based on misuse of the IP TTL field, which determines how many routers a packet may traverse before being dropped.[7] By sending packets with carefully chosen TTL values, an attacker can make a NIDS believe a given packet will reach the destination while knowing that it won't. As a concrete example, the SYN bomb scanner described above is vulnerable to such deception (ACKs could be sent with small TTL values). This section describes a scanner that detects attempts to misuse IP TTL values in this manner.

What the scanner looks for: The scanner looks for unexpected variation in the TTL values of IP packets originating from the host. Specifically, it looks for differing TTL values among packets of a single TCP session. Although TTL values may vary among inbound packets, because different packets may legitimately traverse different paths, such variation should not occur within a session.

How the scanner works: The scanner examines the TTL value for TCP packets originating from a host. The TTL value of the initial SYN packet (for outbound connections) or SYN/ACK packet (for inbound connections) is recorded in a table until the host side of the connection moves to the closed state. The TTL value of each subsequent packet for that connection is compared to the original. Any difference is flagged as TTL misuse, unless it is a RST with TTL=255 (the maximum value). Both Linux and NetBSD use the maximum TTL value for RST packets, presumably to maximize their chance of reaching the destination.

When an attack is detected: The current scanner's policy involves two things. The TTL fields are normalized to the original value, and an alert is generated.

Performance data: We applied the TTL scanner to the traffic of a Linux desktop engaged in typical network usage for over an hour. We observed only 2 different TTL values in packets originating from the desktop: 98.5% of the packets had a TTL of 64 and the remainder had a TTL of 255. All of the TCP packets were among those with TTL of 64, with one exception: a RST packet with TTL=255. The other packets with TTL of 255 were ICMP and other non-TCP traffic.

Discussion: This scanner's detection works well for detecting most NIDS insertion attacks in TCP streams, since there is no vagueness regarding network topology between a host and its NI. It can be extended in several ways. First, it should check for low initial TTL values, which might indicate a non-deterministic insertion attack given some routes being short enough and some not; detecting departure from observed system default values (e.g., 64 and 255) should be sufficient. Second, it should check TTL values for non-TCP packets. This will again rely on observed defaults, with one caveat: tools like traceroute legitimately use low and varying TTL values on non-TCP packets. An augmented scanner would have to understand the pattern exhibited by such tools in order to restrict the non-flagged TTL variation patterns.

3.6.6 Detecting IP Fragmentation Misuse

IP fragmentation can be abused for a variety of attacks. Given known bugs in target machines or NIDSs, IP fragmentation can be used to crash systems or avoid detection; tools like fragrouter [SANS00] exist for testing or exploiting IP fragmentation corner cases. Similarly, different interpretations of overlapping fragments can be exploited to avoid detection. As well, incomplete fragment sets can be used as a capture-and-hold DoS attack.

What the scanner looks for: The scanner looks for five suspicious uses of IP fragmentation. First, overlapping IP fragments are not legitimate—a bug in the host software may cause overlapping, but should not have different data in the overlapping regions—so, the scanner looks for differing data in overlapping regions. Second, incomplete fragmented packets can only cause problems for the receiver, so the scanner looks for them. Third, fragments of a given IP packet should all have the same TTL value. Fourth, only a

[7] This should not be confused with the DNS TTL field used in the Code-Red scanner.

last fragment should ever be smaller than the minimum legal MTU of 68 bytes [ISI81]; many NIDS evasion attacks violate this rule to hide TCP, UDP, or application frame headers from NIDSs that do not reconstitute fragmented packets. Fifth, IP fragmentation of TCP streams is suspicious. This last item is the least certain, but most TCP connections negotiate a "maximum segment size" (`mss`) during setup and modern TCP implementations will also adjust their `mss` field when an ICMP "fragmentation required" message is received.

How the scanner works: The scanner *subscribes* (with `contain` rights) for all outbound IP packets that have either the "More Fragments" bit set or a non-zero value for the IP fragment offset. These two subscriptions capture all Ethernet frames that are part of fragmented IP packets. The first sequential fragmented packet has the "More Fragments" bit set and a zero offset. Fragments in between have the "More Fragments" bit set and a non-zero offset. The last fragment doesn't have the "More Fragments" bit set but it does have a non-zero offset.

The scanner tracks all pending fragments. Each received fragment is compared to held fragments to determine if it completes a full IP packet. If not, it is added to the cache. When all fragments for a packet are received at the NI, the scanner determines whether the IP fragmentation is acceptable. If the full packet is part of a TCP stream, it is flagged. If the fragments have different TTL values, it is flagged. If any fragment other than the last is smaller than 64 bytes, it is flagged. If the fragments overlap and the overlapping ranges contain different data, it is flagged. If nothing is flagged, the fragments are `passed` in ascending order.

Periodically, the fragment structure is checked to determine if an incomplete packet has been held for more than a timeout value (currently one second). If so, the pieces are `cut`. If more than two such timeouts occur in a second or ten in a minute, the host's actions are flagged.

When an attack is detected: There are five cases flagged, all of which result in an alert being generated. In addition, we have the following policies in place: overlapping fragments with mismatching data are dropped, under the assumption that either the host OS is buggy or one of the constructions is an attack; fragments with mismatching TTL fields are sent with all TTLs matching the highest value; incorrectly fragmented packets are dropped; timed out fragments are dropped (as described); fragmented TCP packets are currently passed (if the other rules are not violated).

Performance data: We ran the scanner against a desktop machine, but observed no IP fragmentation during normal operation. With test utilities sending 64KB UDP packets (over Ethernet), we measured the time delay between the first frame's arrival at the NI and the last. The average time before all fragments are received was 0.53ms, with values ranging from 0.46ms to 2.5ms. These values indicate that our timeout period may be too generous.

Discussion: Flagging IP fragmentation of TCP streams is only reasonable for operating systems with modern networking stacks, which can be known by an administrator setting policies. Older systems may actually employ IP fragmentation rather than aggressive `mss` maintenance. Because of this and the possibility of fragmentation by intermediate routers, a rule like this would not be appropriate for a non-host-specific NIDS.

Our original IP fragmentation scanner also watched for out-of-order IP fragments, since this is another possible source of reconstitution bugs. In testing, however, we discovered that at least one OS (Linux) regularly sends its fragments in reverse order. The NI software, therefore, always waits until all fragments are sent and then propagates them in order.

We originally planned to detect unreasonable usage of fragmentation and undersized fragments by caching the MTU values observed (in ICMP "fragmentation required" messages) for various destinations. We encountered several difficulties. First, it was unclear how long to retain the values, since any replacement might cause a false alarm. Second, an external attacker could fill the MTU cache with generated messages, creating state management difficulties. Third, a conspiring external machine with the ability to spoof packets could easily generate the ICMP packets needed to fool the scanner. Since IP fragmentation is legal, we decided to focus on clear misuses of it.

As with most of the scanners described, the IP fragmentation scanner is susceptible to space exhaustion by the host. Specifically, a host could send large numbers of incomplete fragmented packets, filling the NIs buffer capacity. As noted earlier, however, such an attack mainly damages the host itself, denying it access to the network. This seems an acceptable trade-off given the machine's misbehavior. A similar analysis exists for the other scanners.

3.6.7 Other Scanners

Of course, many other scanners are possible. Any traditional NIDS scanning algorithm fits, both inbound and outbound, and can be expected to work better (as described in [Handley01, Malan00]) after the normalization of IP and TCP done by Siphon. For example, NIC-embedded prevention/detection of basic spoofing (e.g., of IP addresses) and sniffing (e.g., by listening with the NI in "promiscuous mode") are appropriate, as is done in 3Com's Embedded Firewall product [3Com01a].

Several other examples of evasion and protocol abuse can be detected as well. For example, misbehaving hosts can increase the rate at which senders transmit data to them by sending early or partial ACKs [Savage99]; sitting on the NI, a scanner could easily see such misbehavior. A TCP abuse of more concern is the use of overlapping TCP segments with different data, much like the overlapping IP fragment example above; usable for NIDS insertion attacks [Ptacek98], such behavior is easily detected by a scanner looking for it.

Finally, we believe that the less aggregated and local view of traffic exhibited at the NI will help with more complex detection schemes, such as those for stepping stones [Donoho02, Zhang00] or general anomaly detection of network traffic.

3.7 Related Work

Self-securing NIs build on much existing technology and borrow ideas from previous work. In particular, network intrusion detection, virus detection, and firewalls are well-established, commonlyused mechanisms [Axelsson98, Cheswick94]. Also, many of the arguments for distributing firewall functions [Friedman01, Ioannidis00, Nessett98] and embedding them into network interface cards [3Com01a, Friedman01] have been made in previous work. This previous work and others [3Com01b, Bartal99, Miller96] also address the issue of remote policy configuration for such systems. Notably, the 3Com Embedded Firewall product [3Com01a] extends NICs with firewall policies such as IP spoofing prevention, promiscuous mode prevention, and selective filtering of packets based on fields like IP address and port number. These systems do not focus on host compromise detection and containment like self-securing NIs do. We extend previous work with examples of more detailed traffic analysis and a system software structure for supporting them.

There are few examples of detailed network intrusion detection documented in the literature, though many system administrators create tools when the need arises. One well-described example is Bro [Paxson98], an extensible, real-time, passive network monitor. Bro provides a scripting language for reacting to pre-programmed network events, which works well for experts. Its clean framework replaced a collection of *ad hoc* scripts, which is how most environments examine network traffic. Our work builds on such previous work by providing a programming model that should accommodate less-expert scanner writers and contain broken scanners. As well, embedding scanning functionality into NIs instead of network taps eliminates several of the challenges described in Bro, such as overload attacks, dropped packets, and crash attacks.

Application proxies, particularly for e-mail and web traffic, can be used as intermediaries between vulnerable LAN systems and particular application services. Some such proxies examine the corresponding dataflows to identify and block dangerous data [Martin97, Sophos02a]. Each such proxy addresses a single protocol, introduces a central bottleneck, and sometimes creates a visibility problem by making all requests appear to come from a single system. Self-securing NIs allow similar checking in a multipurpose, scanner-constraining platform.

There is much ongoing research into addressing Distributed DoS (DDoS) attacks. Most counter-measures start from the victim, using traceback and throttling to get as close to sources as possible. The D-WARD system [Mirkovic02] instead attempts to detect outgoing attacks at source routers, using anomaly detection on traffic flows, and throttle them closer to home. The arguments for this approach bear similarity to those for self-securing NIs, though they focus on a different threat: outgoing DDoS attacks rather than two-stage attacks. The ideas are complementary, and pushing D-WARD all the way to the true sources (individual NIs) is an idea worth exploring.

A substantial body of research has examined the execution of application functionality by network cards [Fiuczynski98, Hitz90] and infrastructure components [Alexander99, Decasper99, Tennenhouse97, Wetherall99]. Although scanners are not fully trusted, they are also not submitted by untrusted clients. Nonetheless, this prior work lays solid groundwork for resource management within network components.

3.8 Summary

Self-securing network interfaces are a promising addition to the network security arsenal. This chapter makes a case for them, identifies NI software design challenges, and describes an NI software architecture to address them. It then goes on to illustrate the potential of self-securing NIs with a prototype NI kernel and example scanners that address several high-profile network security problems: e-mail viruses, insertion and evasion efforts, state holding DoS attacks, and Code-Red style worms.

4 SELF-SECURING STORAGE

4.1 Introduction

Despite the best efforts of system designers and implementers, it has proven difficult to prevent computer security breaches. This fact is of growing importance as organizations find themselves increasingly dependent on wide-area networking (providing more potential sources of intrusions) and computer-maintained information (raising the significance of potential damage). A successful intruder can obtain the rights and identity of a legitimate user or administrator. With these rights, it is possible to disrupt the system by accessing, modifying, or destroying critical data.

Even after an intrusion has been detected and terminated, system administrators still face two difficult tasks: determining the damage caused by the intrusion and restoring the system to a safe state. Damage includes compromised secrets, creation of back doors and Trojan horses, and tainting of stored data. Detecting each of these is made difficult by crafty intruders who understand how to scrub audit logs and disrupt automated tamper detection systems. System restoration involves identifying a clean backup (i.e., one created prior to the intrusion), reinitializing the system, and restoring information from the backup. Such restoration often requires a significant amount of time, reduces the availability of the original system, and frequently causes loss of data created between the safe backup and the intrusion.

Self-securing storage offers a partial solution to these problems by preventing intruders from undetectably tampering with or permanently deleting stored data. Since intruders can take the identity of real users and even the host OS, any resource controlled by the operating system is vulnerable, including the raw storage. Rather than acting as slaves to host OSes, self-securing storage devices view them, and their users, as questionable entities for which they work. These self-contained, self-controlled devices internally version all data and audit all requests for a guaranteed amount of time (e.g., a week or a month), thus providing system administrators time to detect intrusions. For intrusions detected within this window, all of the version and audit information is available for analysis and recovery. The critical difference between self-securing storage and host-controlled versioning (e.g., Elephant [Santry99]) is that intruders can no longer bypass the versioning software by compromising complex OSes or their poorly-protected user accounts. Instead, intruders must compromise single-purpose devices that export only a simple storage interface, and in some configurations, they may have to compromise both.

This chapter describes self-securing storage and our implementation of a self-securing storage server, called S4. A number of challenges arise when storage devices distrust their clients. Most importantly, it may be difficult to keep all versions of all data for an extended period of time, and it is not acceptable to trust the client to specify what is important to keep. Fortunately, storage densities increase faster than most computer characteristics (100%+ per annum in recent years). Analysis of recent workload studies [Santry99, Vogels99] suggests that it is possible to version all data on modern 30–100GB drives for several weeks. Further, aggressive compression and cross-version differencing techniques can extend the intrusion detection window offered by self-securing storage devices. Other challenges include efficiently encoding the many metadata changes, achieving secure administrative control, and dealing with denial-of-service attacks.

The S4 system addresses these challenges with a new storage management structure. Specifically, S4 uses a log-structured object system for data versions and a novel journal-based structure for metadata versions. In addition to reducing space utilization, journal-based metadata simplifies background compaction and reorganization for blocks shared across many versions. Experiments with S4 show that the security and data survivability benefits of self-securing storage can be realized with reasonable performance. Specifically, the performance of S4-enhanced NFS is comparable to FreeBSD's NFS for both micro-benchmarks and application benchmarks. The fundamental costs associated with self-securing storage degrade performance by less than 13% relative to similar systems that provide no data protection guarantees.

4.2 Intrusion Diagnosis and Recovery

Upon gaining access to a system, an intruder has several avenues of mischief. Most intruders attempt to destroy evidence of their presence by erasing or modifying system log files. Many intruders also install back doors in the system, allowing them to gain access at will in the future. They may also install other software, read and modify sensitive files, or use the system as a platform for launching additional attacks. Depending on the skill with which the intruders hide their presence, there will be some detection latency before the intrusion is discovered by an automated intrusion detection system (IDS) or by a suspicious user or administrator. During this time, the intruders can continue their malicious activities while users continue to use the system, thus entangling legitimate changes with those of the intruders. Once an intrusion has been detected and discontinued, the system administrator is left with two difficult tasks: diagnosis and recovery.

Diagnosis is challenging because intruders can usually compromise the "administrator" account on most operating systems, giving them full control over all resources. In particular, this gives them the ability to manipulate everything stored on the system's disks, including audit logs, file modification times, and tamper detection utilities. Recovery is difficult because diagnosis is difficult and because user-convenience is an important issue. This section discusses intrusion diagnosis and recovery in greater detail, and the next section describes how self-securing storage addresses them.

4.2.1 Diagnosis

Intrusion diagnosis consists of three phases: detecting the intrusion, discovering what weaknesses were exploited (for future prevention), and determining what the intruder did. All are difficult when the intruder has free reign over storage and the OS.

Without the ability to protect storage from compromised operating systems, intrusion detection may be limited to alert users and system administrators noticing odd behavior. Examining the system logs is the most common approach to intrusion detection [Denning87], but when intruders can manipulate the log files, such an approach is not useful. Some intrusion detection systems also look for changes to important system files [Kim94]. Such systems are vulnerable to intruders that can change what the IDS thinks is a "safe" copy.

Determining how an intruder compromised the system is often impossible in conventional systems, because he will scrub the system logs. In addition, any exploit tools (utilities for compromising computer systems) that may have been stored on the target machine for use in multi-stage intrusions are usually deleted. The common "solutions" are to try to catch the intruder in the act or to hope that he forgot to delete his exploit tools.

The last step in diagnosing an intrusion is to discover what was accessed and modified by the intruder. This is difficult, because file access and modification times can be changed and system log files can be doctored. In addition, checksum databases are of limited use, since they are effective only for static files.

4.2.2 Recovery

Because it is usually not possible to diagnose an intruder's activities, full system recovery generally requires that the compromised machine be wiped clean and reinstalled from scratch. Prior to erasing the entire state of the system, users may insist that data, modified since the intrusion, be saved. The more effort that went into creating the changes, the more motivation there is to keep this data. Unfortunately, as the size and complexity of the data grows, the likelihood that tampering will go unnoticed increases. Foolproof assessment of the modified data is very difficult, and overlooked tampering may hide tainted information or a back door inserted by the intruder.

Upon restoring the OS and any applications on the system, the administrator must identify a backup that was made prior to the intrusion; the most recent backup may not be usable. After restoring data from a pre-intrusion backup, the legitimately modified data can be restored to the system, and users may resume

using the system. This process often takes a considerable amount of time—time during which users are denied service.

4.3 Self-Securing Storage

Self-securing storage ensures information survival and auditing of all accesses by establishing a security perimeter around the storage device. Conventional storage devices are slaves to host operating systems, relying on them to protect users' data. A self-securing storage device operates as an independent entity, tasked with the responsibility of not only storing data, but protecting it. This shift of storage security functionality into the storage device's firmware allows data and audit information to be safeguarded in the presence of file server and client system intrusions. Even if the OSes of these systems are compromised and an intruder is able to issue commands directly to the self-securing storage device, the new security perimeter remains intact.

Behind the security perimeter, the storage device ensures data survival by keeping previous versions of the data. This history pool of old data versions, combined with the audit log of accesses, can be used to diagnose and recover from intrusions. This section discusses the benefits of self-securing storage and several core design issues that arise in realizing this type of device.

4.3.1 Enabling Intrusion Survival

Self-securing storage assists in intrusion recovery by allowing the administrator to view audit information and quickly restore modified or deleted files. The audit and version information also helps to diagnose intrusions and detect the propagation of maliciously modified data.

Self-securing storage simplifies detection of an intrusion since versioned system logs cannot be imperceptibly altered. In addition, modified system executables are easily noticed. Because of this, self-securing storage makes conventional tamper detection systems obsolete.

Since the administrator has the complete picture of the system's state, from intrusion until discovery, it is considerably easier to establish the method used to gain entry. For instance, the system logs would have normally been doctored, but by examining the versioned copies of the logs, the administrator can see any messages that were generated during the intrusion and later removed. In addition, any exploit tools temporarily stored on the system can be recovered.

Previous versions of system files, from before the intrusion, can be quickly and easily restored by resurrecting them from the history pool. This prevents the need for a complete re-installation of the operating system, and it does not rely on having a recent backup or up-to-date checksums (for tamper detection) of system files. After such restoration, critical data can be incrementally recovered from the history pool. Additionally, by utilizing the storage device's audit log, it is possible to assess which data might have been directly affected by the intruder.

The data protection that self-securing storage provides allows easy detection of modifications, selective recovery of tampered files, prevention of data loss due to out-of-date backups, and speedy recovery since data need not be loaded from an off-line archive.

4.3.2 Device Security Perimeter

The device's security model is what makes the ability to keep old versions more than just a user convenience. The security perimeter consists of self-contained software that exports only a simple storage interface to the outside world and verifies each command's integrity before processing it. In contrast, most file servers and client machines run a multitude of services that are susceptible to attack. Since the self-securing storage device is a single-function device, the task of making it secure is much easier; compromising its firmware is analogous to breaking into an IDE or SCSI disk.

The actual protocol used to communicate with the storage device does not affect the data integrity that the new security perimeter provides. The choice of protocol does, however, affect the usefulness of the audit log in terms of the actions it can record and its correctness. For instance, the NFS protocol provides no

authentication or integrity guarantees; therefore the audit log may not be able to accurately link a request with its originating client. Nonetheless, the principles of self-securing storage apply equally to "enhanced" disk drives, network-attached storage servers, and file servers.

For network-attached storage devices (as opposed to devices attached directly to a single host system), the new security perimeter becomes more useful if the device can verify each access as coming from both a valid user and a valid client. Such verification allows the device to enforce access control decisions and partially track propagation of tainted data. If clients and users are authenticated, accesses can be tracked to a single client machine, and the device's audit log can yield the scope of direct damage from the intrusion of a given machine or user account.

4.3.3 History Pool Management

The old versions of objects kept by the device comprise the history pool. Every time an object is modified or deleted, the version that existed just prior to the modification becomes part of the history pool. Eventually an old version will age and have its space reclaimed. Because clients cannot be trusted to demarcate versions consisting of multiple modifications, a separate version should be kept for every modification. This is in contrast to versioning file systems that generally create new versions only when a file is closed.

A self-securing storage device guarantees a lower bound on the amount of time that a deprecated object remains in the history pool before it is reclaimed. During this window of time, the old version of the object can be completely restored by requesting that the drive copy forward the old version, thus making a new version. The guaranteed window of time during which an object can be restored is called the detection window. When determining the size of this window, the administrator must examine the tradeoff between the detection latency provided by a large window and the extra disk space that is consumed by the proportionally larger history pool.

Although the capacity of disk drives is growing at a remarkable rate, it is still finite, which poses two problems:

- Providing a reasonable detection window in exceptionally busy systems.
- Dealing with malicious users that attempt to fill the history pool. (Note that space exhaustion attacks are not unique to self-securing storage. However, device-managed versioning makes conventional user quotas ineffective for limiting them.)

In a busy system, the amount of data written could make providing a reasonable detection window difficult. Fortunately, the analysis in Section 4.5.2 suggests that multi-week detection windows can be provided in many environments at a reasonable cost. Further, aggressive compression and differencing of old versions can significantly extend the detection window.

Deliberate attempts to overflow the history pool cannot be prevented by simply increasing the space available. As with most denial of service attacks, there is no perfect solution. There are three flawed approaches to addressing this type of abuse. The first is to have the device reclaim the space held by the oldest objects when the history pool is full. Unfortunately, this would allow an intruder to destroy information by causing its previous instances to be reclaimed from the overflowing history pool. The second awed approach is to stop versioning objects when the history pool fills; although this will allow recovery of old data, system administrators would no longer be able to diagnose the actions of an intruder or differentiate them from subsequent legitimate changes. The third awed approach is for the drive to deny any action that would require additional versions once the history pool fills; this would result in denial of service to all users (legitimate or not).

Our hybrid approach to this problem is to try to prevent the history pool from being filled by detecting probable abuses and throttling the source machine's accesses. This allows human intervention before the system is forced to choose from the above poor alternatives. Selectively increasing latency and/or decreasing bandwidth allows well-behaved users to continue to use the system even while it is under attack. Experience will show how well this works in practice.

Since the history pool will be used for intrusion diagnosis and recovery, not just recovering from accidental destruction of data, it is difficult to construct a safe algorithm that would save space in the history pool by pruning versions within the detection window. Almost any algorithm that selectively removes versions has the potential to be abused by an intruder to cover his tracks and to successfully destroy/modify information during a break-in.

4.3.4 History Pool Access Control

The history pool contains a wealth of information about the system's recent activity. This makes accessing the history pool a sensitive operation, since it allows the resurrection of deleted and overwritten objects. This is a standard problem posed by versioning file systems, but is exacerbated by the inability to selectively delete versions.

There are two basic approaches that can be taken toward access control for the history pool. The first is to allow only a single administrative entity to have the power to view and restore items from the history pool. This could be useful in situations where the old data is considered to be highly sensitive. Having a single tightly-controlled key for accessing historical data decreases the likelihood of an intruder gaining access to it. Although this improves security, it prevents users from being able to recover from their own mistakes, thus consuming the administrator's time to restore users' files. The second approach is to allow users to recover their own old objects (in addition to the administrator). This provides the convenience of a user being able to recover their deleted data easily, but also allows an intruder, who obtains valid credentials for a given user, to recover that user's old file versions.

Our compromise is to allow users to selectively make this decision. By choice, a user could thus delete an object, version, or all versions from visibility by anyone other than the administrator, since permanent deletion of data via any other method than aging would be unsafe. This choice allows users to enjoy the benefits of versioning for presentations and source code, while preventing access to visible versions of embarrassing images or unsent e-mail drafts.

4.3.5 Administrative Access

While self-securing storage is able to protect data in a nearly transparent way, it introduces several administrative issues. These additional complexities must be properly managed for self-securing storage to be effective.

Secure administrative control: Self-securing storage devices must have a secure, out-of-band interface for handling administrative tasks such as configuration and intrusion recovery. This interface must use strong authentication to ensure that requests cannot be forged by the OS or other processes on potentially compromised client systems. This precaution is necessary because the administrative interface (necessarily) permits commands that are destructive to data. For example, setting the length of the detection window is handled via this interface. If an intruder were able to change the length of the detection window, he could erase data from the history pool by shrinking it.

Although strong authentication is critical for this interface, confidentiality may be less so. Since history data can be viewed and the audit log read with this interface, some privacy concerns arise. However, most organizations still employ file systems that do not encrypt their normal traffic, thus it is unclear whether there is a need to obscure the data at this point.

Administrative alerts (for intrusion detection) pose another interesting problem. The channel that is used for this device-to-administrator communication must, at a minimum, be tamper-evident. This is necessary to prevent the intruder from successfully preventing the alerts from reaching the administrator. The administrator may also wish to hide the fact that an alert was issued at all. She may even wish to hide the presence of the communication channel completely. These concerns play a significant role in the implementation of the administrative control channel.

The administrative interface may be implemented in a number of ways. For example, there could be a dedicated administrative terminal connection. For network-attached storage (e.g., file servers), one can

use cryptography and well-protected administrative keys. For disks attached to host systems, a secure communication channel between a smart disk drive and a remote administration console can be created by cryptographically tunneling commands through the host's OS. No matter which interface is chosen, it must be the case that obtaining "superuser" privileges on a client computer system is not sufficient to gain administrative access to the storage device.

Setting the detection window. The storage device's detection window is the configurable parameter that determines the duration of history available for detecting, diagnosing, and recovering from intrusions. For this reason, it is desirable that the window be very long. On the other hand, more history requires more disk capacity. Our studies indicate that, with current disk technology, configuring the detection window for between one week and one month of history provides a reasonable balance for a large variety of workloads. The size of the resulting history pool is up to the total number of bytes written or deleted during the detection window. Cross-version differencing and compression can reduce the history pool size [Soules02].

Denial of service. As with any system, an intruder can fill up the storage capacity and prevent forward progress by the system. Such denial-of-service is easy to detect once it occurs, but with self-securing storage, it requires careful administrative intervention. Specifically, restoring the system after such an attack requires true deletion of files or versions from the history pool in order to free space. Such deletion interferes with the basic goals of self-securing storage, so administrative privileges and care are required.

Additionally, conventional file systems use space quotas to prevent a single user from using a disproportionate amount of storage. This concept can be utilized for self-securing storage as well, but it would need to be modified to also contain a rate quota. This second quota would bound the average rate at which a user can write data. Such rate quotas are not likely to completely solve the problem. However, they could allow the device to detect a likely attack and throttle the offending user or client machine. This might provide sufficient time for an administrator to react before all free space is consumed.

Privacy and inability to delete. Some users will object to being unable to delete files whenever they want, but allowing users to permanently delete their files (without waiting for the detection window to elapse) would open a path for intruders to destroy data. The chosen compromise between the need for security and the users' desire for privacy is to allow users to mark files as "unrecoverable." Such files would be retained in the device like normal, but could only be retrieved from the history pool by the administrator, not the user. This prevents an intruder (who could masquerade as a normal user) from recovering the file, but it still allows the administrator to completely recover after an intrusion. While some users may object to it being retained at all, it is often possible for an administrator to recover a deleted file in a conventional system as well [Farmer01]. Also, if the file was stored on the system for any reasonable length of time, it is likely that it was backed up for disaster recovery, making it unlikely that a normal user could force deletion [Hutchinson99].

4.3.6 Version and Administration Tools

Since self-securing storage devices store versions of raw data, users and administrators will need assistance in parsing the history pool. Tools for traversing the history must assist by bridging the gap between standard file interfaces and the raw versions that are stored by the device. By being aware of both the versioning system and formats of the data objects, utilities can present interfaces similar to that of Elephant [Santry99], with "time-enhanced" versions of standard utilities such as `ls` and `cp`. This is accomplished by extending the read interfaces of the device to include an optional time parameter. When this parameter is specified, the drive returns data from the version of the object that was valid at the requested time. Please see Chapter 6 for details on the versioning file systems research done in relation to self-securing storage.

In addition to providing a simple view of data objects in isolation, intrusion diagnosis tools can utilize the audit log to provide an estimate of damage. For instance, it is possible to see all files and directories that a client modified during the period of time that it was compromised. Further estimates of the propagation of data written by compromised clients are also possible, though imperfect. For example, diagnosis tools

may be able to establish a link between objects based on the fact that one was read just before another was written. Such a link between a source file and its corresponding object file would be useful if a user determines that a source file had been tampered with; in this situation, the object file should also be restored or removed.

4.4 S4 Implementation

S4 is a self-securing storage server that transparently maintains an efficient object-versioning system for its clients. It aims to perform comparably with current systems, while providing the benefits of self-securing storage and minimizing the corresponding space explosion.

RPC Type	Allows Time-Based Access	Description
`Create`	no	Create an object
`Delete`	no	Delete an object
`Read`	yes	Read data from an object
`Write`	no	Write data to an object
`Append`	no	Append data to the end of an object
`Truncate`	no	Truncate an object to a specified length
`GetAttr`	yes	Get the attributes of an object (S4-specific and opaque)
`SetAttr`	no	Set the opaque attributes of an object
`GetACLByUser`	yes	Get an ACL entry for an object given a specific UserID
`GetACLByIndex`	yes	Get an ACL entry for an object by its index in the object's ACL table
`SetACL`	no	Set an ACL entry for an object
`PCreate`	no	Create a partition (associate a name with an ObjectID)
`PDelete`	no	Delete a partition (remove a name/ObjectID association)
`PList`	yes	List the partitions
`PMount`	yes	Retrieve the ObjectID given its name
`Sync`	not applicable	Sync the entire cache to disk
`Flush`	not applicable	Removes all versions of all objects between two times
`FlushO`	not applicable	Removes all versions of an object between two times
`SetWindow`	not applicable	Adjusts the guaranteed detection window of the S4 device

Table 4.1: S4 Remote Procedure Call List. Operations that support time-based access accept a time in addition to the normal parameters; this time is used to find the appropriate version in the history pool. Note that all modifications create new versions without affecting the previous versions.

4.4.1 A Self-securing Object Store

S4 is a network-attached object store with an interface similar to recent object-based disk proposals [Gibson99, T10.org99]. This interface simplifies access control and internal performance enhancement relative to a standard block interface.

In S4, objects exist in a at namespace managed by the "drive" (i.e., the object store). When objects are created, they are given a unique identifier (`ObjectID`) by the drive, which is used by the client for all future references to that object. Each object has an access control structure that specifies which entities (users and client machines) have permission to access the object. Objects also have metadata, file data, and opaque attribute space (for use by client file systems) associated with them.

To enable persistent mount points, a S4 drive supports "named objects." The object names are an association of an arbitrary ASCII string with a particular `ObjectID`. The table of named objects is implemented as a special S4 object accessed through dedicated partition manipulation RPC calls. This table is versioned in the same manner as other objects on the S4 drive.

4.4.1.1 S4 RPC interface

Table 4.1 lists the RPC commands supported by the S4 drive. The read-only commands (`read`, `getattr`, `getacl`, `plist`, and `pmount`) accept an optional time parameter. When the *time* is provided, the drive performs the read request on the version of the object that was "most current" at the time specified, provided that the user making the request has sufficient privileges.

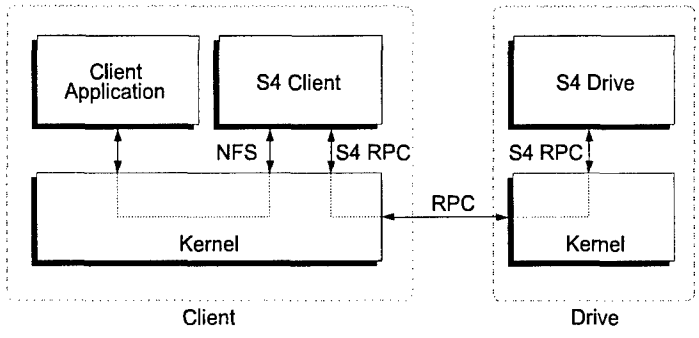
The ACLs associated with objects have the traditional set of flags, with one addition–the `Recovery` flag. The `Recovery` flag determines whether or not a given user may read (recover) an object version from the history pool once it is overwritten or deleted. When this flag is clear, only the device administrator may read this object version once it is pushed into the history pool. The Recovery flag allows users to decide the sensitivity of old versions on a file-by-file basis.

4.4.1.2 S4/NFS Translation

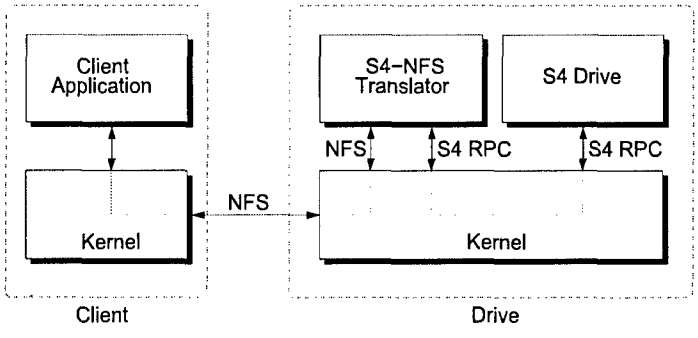
Since one goal of self-securing storage is to provide an enhanced level of security and convenience on existing systems, the prototype minimizes changes to client systems. In keeping with this philosophy, the S4 drive is network-attached and an "S4 client" daemon serves as a user-level file system translator (Figure 4.1a). The S4 client translates requests from a file system on the target OS to S4-specific requests for objects. Because it runs as a user-level process, without operating system modifications, the S4 client should port to different systems easily.

The S4 client currently has the ability to translate NFS version 2 requests to S4 requests. The S4 client appears to the local workstation as a NFS server. This emulated NFS server is mounted via the loopback interface to allow only that workstation access to the S4 client. The client receives the NFS requests and translates them into S4 operations. NFSv2 was chosen over version 3 because its client is well-supported within Linux, and its lack of write caching allows the drive to maintain a detailed account of client actions.

Figure 4.1 shows two approaches to using the S4 client to serve NFS requests with the S4 drive. The first places the S4 client on the client system, as described previously, and uses the S4 drive as a network-attached storage device. The second incorporates the S4 client functionality into the server, as a NFSto-S4 translator. This configuration acts as a S4enhanced NFS server (Figure 4.1b) for normal file system activity, but recovery must still be accomplished through the S4 protocol since the NFS protocol has no notion of "time-based" access.



a) Baseline S4 (network-attached object store)



b) S4-enhanced NFS server

Figure 4.1: Two S4 Configurations. This figure shows two S4 configurations that provide self-securing storage via a NFS interface. (a) shows S4 as a network-attached object store with the S4 client daemon translating NFS requests to S4specific RPCs. (b) shows a self-securing NFS server created by combining the NFS-to-S4 translation and the S4 drive.

The implementation of the NFS file system overlays files and directories on top of S4 objects. Objects used as directories contain a list of ASCII filenames and their associated NFS file handles. Objects used as files and symlinks contain the corresponding data. The NFS attribute structure is maintained within the opaque attribute space of each object.

When the S4 client receives a NFS request, the NFS file handle (previously constructed by the S4 client) can be directly hashed into the ObjectID of the directory or file. The S4 client can then make requests directly to the drive for the desired data.

To support NFSv2 semantics, the client sends an additional RPC to the drive to flush buffered writes to the disk at the end of each NFS operation that modifies the state of one or more objects. Since this RPC does not return until the synchronization is complete, NFSv2 semantics are supported even though the drive normally caches writes.

Because the client overlays a file system on top of the at object namespace, some file system operations require several drive operations (and hence RPC calls). These sets of operations are analogous to the operations that file systems must perform on block-based devices. To minimize the number of RPC calls necessary, the S4 client aggressively maintains attribute and directory caches (for reads only). The drive also supports batching of `setattr`, `getattr`, and sync operations with `create`, `read`, `write`, and append operations.

4.4.2 S4 Drive Internals

The main goals for the S4 drive implementation are to avoid performance overhead and to minimize wasted space, while keeping all versions of all objects for a given period of time. Achieving these goals requires a combination of known and novel techniques for organizing on-disk data.

4.4.2.1 Log-structuring for Efficient Writes

Since data within the history pool cannot be overwritten, the S4 drive uses a log structure similar to LFS [Rosenblum92]. This structure allows multiple data and metadata updates to be clustered into fewer, larger writes. Importantly, it also obviates any need to move previous versions before writing.

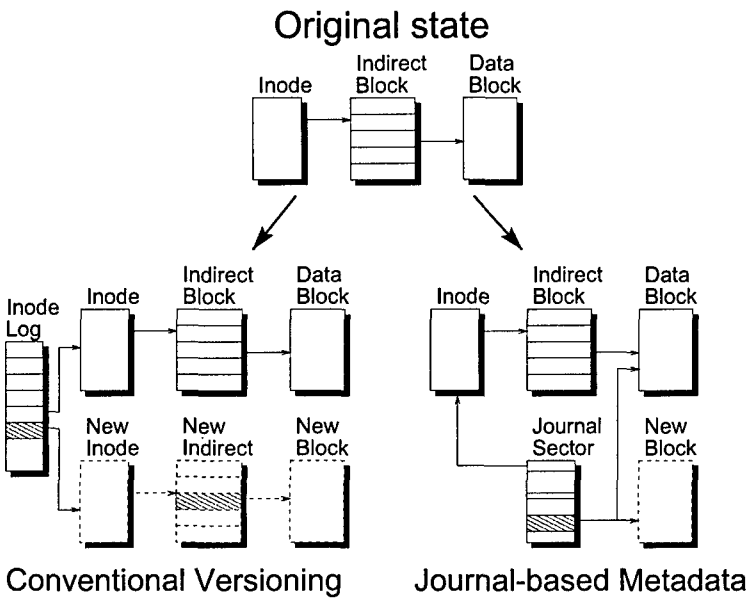


Figure 4.2: Efficiency of Metadata Versioning. The above figure compares metadata management in a conventional versioning system to S4's journal-based metadata approach. When writing to an indirect block, a conventional versioning system allocates a new data block, a new indirect block, and a new inode. Also, the identity of the new inode must be recorded (e.g., in an Elephant-like inode log). With journal-based metadata, a single journal entry suffices, pointing to both the new and old data blocks.

In order to prune old versions and reclaim unused segments, S4 includes a background cleaner. While the goal of this cleaner is similar to that of the LFS cleaner, the design must be slightly different. Specifically, deprecated objects cannot be reclaimed unless they have also aged out of the history pool. Therefore, the obest time greater than the detection window. Once a suitable object is found, the cleaner permanently frees all data and metadata older than the window. If this clears all of the resources within a segment, the segment can be marked as free and used as a fresh segment for foreground activity.

4.4.2.2 Journal-based Metadata

To efficiently keep all versions of object metadata, S4 uses *journal-based metadata*, which replaces most instances of metadata with compact journal entries.

Because clients are not trusted to notify S4 when objects are closed, every update creates a new version and thus new metadata. For example, when data pointed to by indirect blocks is modified, the indirect blocks must be versioned as well. In a conventional versioning system, a single update to a triple-indirect block could require four new blocks as well as a new inode. Early experiments with this type of versioning system showed that modifying a large file could cause up to a 4x growth in disk usage. Conventional versioning file systems avoid this performance problem by only creating new versions when a file is closed.

In order to significantly reduce these problems, S4 encodes metadata changes in a journal that is maintained for the duration of the detection window. By persistently keeping journal entries of all metadata changes, metadata writes can be safely delayed and coalesced, since individual inode and indirect block versions can be recreated from the journal. To avoid rebuilding an object's current state from the journal during normal operation, an object's metadata is checkpointed to a log segment before being evicted from the cache. Unlike conventional journaling, such checkpointing does not prune journal space; only aging may prune space. Figure 4.2 depicts the difference in disk space usage between journal-based metadata and conventional versioning when writing data to an indirect data block.

In addition to the entries needed to describe metadata changes, a checkpoint entry is needed. This checkpoint entry denotes writing a consistent copy of all of an object's metadata to disk. It is necessary to have at least one checkpoint of an object's metadata on disk at all times, since this is the starting point for all time-based and crash recovery recreations.

Storing an object's changes within the log is done using journal sectors. Each journal sector contains the packed journal entries that refer to a single object's changes made within that segment. The sectors are identified by segment summary information. Journal sectors are chained together backward in time to allow for version reconstruction.

Journal-based metadata can also simplify cross-version differential compression [Burns96]. Since the blocks changed between versions are noted within each entry, it is easy to find the blocks that should be compared. Once the differencing is complete, the old blocks can be discarded, and the difference left in its place. For subsequent reads of old versions, the data for each block must be recreated as the entries are traversed. Cross-version differencing of old data will often be effective in reducing the amount of space used by old versions. Adding differencing technology into the S4 cleaner is an area of future work.

4.4.2.3 Audit Log

In addition to maintaining previous object versions, S4 maintains an append-only audit log of all requests. This log is implemented as a reserved object within the drive that cannot be modified except by the drive itself. However, it can be read via RPC operations. The data written to the audit log includes command arguments as well as the originating client and user. All RPC operations (read, write, and administrative) are logged. Since the audit log may only be written by the drive front end, it need not be versioned, thus increasing space efficiency and decreasing performance costs.

4.5 Evaluation of Self-Securing Storage

This section evaluates the feasibility of self-securing storage. Experiments with S4 indicate that comprehensive versioning and auditing can be performed without a significant performance impact. Also, estimates of capacity growth, based on reported workload characterizations, indicate that history windows of several weeks can easily be supported in several real environments.

4.5.1 Performance

The main performance goal for S4 is to be comparable to other networked file systems while offering enhanced security features. This section demonstrates that this goal is achieved and also explores the overheads specifically associated with self-securing storage features.

4.5.1.1 Experimental Setup

The four systems used in the experiments had the following configurations: (1) a S4 drive running on RedHat 6.1 Linux communicating with a Linux client over S4 RPC through the S4 client module (Figure 4.1a), (2) a S4-enhanced NFS server running on RedHat 6.1 Linux communicating with a Linux client over NFS (Figure 4.1b), (3) a FreeBSD 4.0 server communicating with a Linux client over NFS, and (4) a RedHat 6.1 Linux server communicating with a Linux client over NFS. Since Linux NFS does not comply with the NFSv2 semantics of committing data to stable storage before operation completion, the Linux server's file system was mounted synchronously to approximate NFS semantics. In all cases, NFS was configured to use 4KB read/write transfer sizes, the only option supported by Linux. The FreeBSD NFS configuration exports a BSD FFS file system, while the Linux NFS configuration exports an ext2 file system. All experiments were run five times and have a standard deviation of less than 3% of the mean. The S4 drives were configured with a 128MB buffer cache and a 32MB object cache. The Linux and FreeBSD NFS servers' caches could grow to fill local memory (512MB).

In all experiments, the client system has a 550MHz Pentium III, 128MB RAM, and a 3Com 3C905B 100Mb network adapter. The servers have a 600MHz Pentium III, 512MB RAM, a 9GB 10; 000RPM Ultra2 SCSI Seagate Cheetah drive, an Adaptec AIC7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro100 100Mb network adapter. The client and server are on the same subnet and are connected by a 100Mb network switch. All versions of Linux use an unmodified 2.2.14 kernel, and the BSD system uses a stock FreeBSD 4.0 installation.

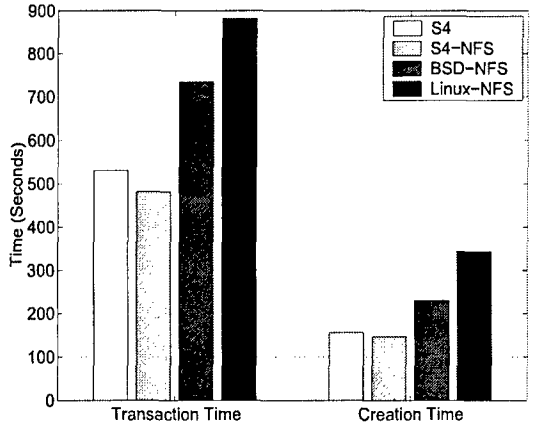


Figure 4.3: PostMark Benchmark

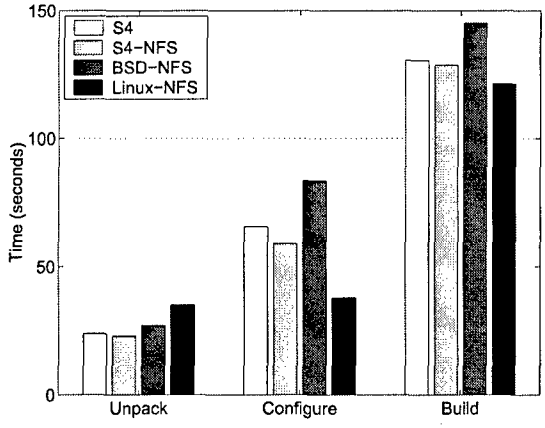


Figure 4.4: SSH-build Benchmark

To evaluate performance for common workloads, results from two application benchmarks are presented: the PostMark benchmark [Katcher97] and the SSH-build benchmark [Ylonen96]. These benchmarks crudely represent Internet server and software development workloads, respectively.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services. It creates a large number of small randomly-sized files (between 512B and 9KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction type are equal.

The SSH-build benchmark was constructed as a replacement for the Andrew file system benchmark [Howard88]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27 (approximately 1MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables.

4.5.1.2 Comparison of the Servers

To gauge the overall performance of S4, the four systems described earlier were compared. As hoped, S4 performs comparably to the existing NFS servers.

Figure 4.3 shows the results of the PostMark benchmark. The times for both the creation (time to create the initial 5000 files) and transaction phases of PostMark are shown for each system. The S4 systems' performance is similar to both BSD and Linux NFS performance, doing slightly better due to their log structured layout.

The times of SSH-build's three phases are shown in Figure 4.4. Performance is similar across the S4 and BSD configurations. The superior performance of the Linux NFS server in the configure stage is due to a much lower number of write I/Os than in the BSD and S4 servers, apparently due to a flaw in the synchronous mount option under Linux.

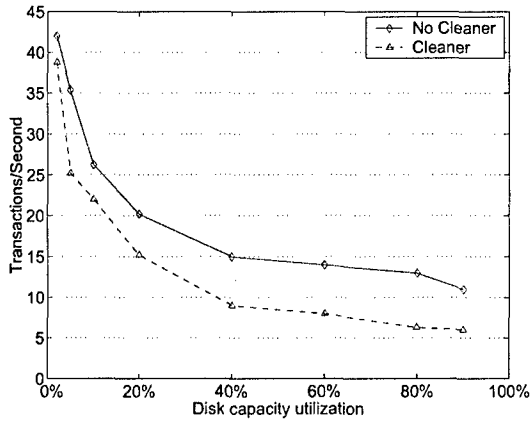


Figure 4.5: Overhead of foreground cleaning in S4. This figure shows the transaction performance of S4 running the PostMark benchmark with varying capacity utilizations. The solid line shows system performance on a system without cleaning. The dashed line shows system performance in the presence of continuous foreground cleaner activity.

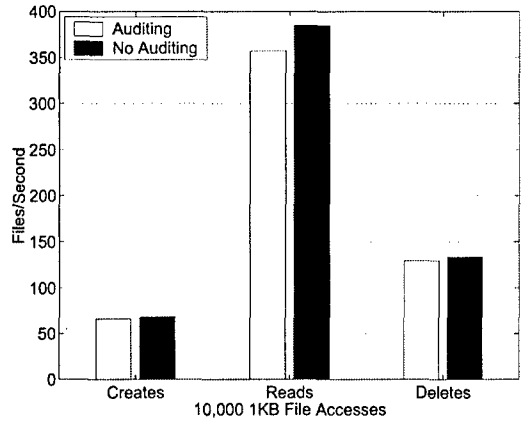


Figure 4.6: Auditing Overhead in S4. This figure shows the impact on small file performance caused by auditing incoming client requests.

4.5.1.3 Overhead of the S4 Cleaner

In addition to the more visible process of creating new versions, S4 must eventually garbage collect data that has expired from the history pool. This garbage collection comes at a cost. The potential overhead of the cleaner was measured by running the PostMark benchmark with 50,000 transactions on increasingly large sets of initial files. For each set of initial files, the benchmark was run once with the cleaner disabled and once with the cleaner competing with foreground activity.

The results shown in Figure 4.5 represent PostMark running with the initial set of files filling between 2% and 90% of a 2GB disk. As expected, when the working set increases, performance of the normal S4 system degrades due to increasingly poor cache and disk locality. The sharp drop in the graph from 2% to 10% is caused by the fact that the set of files and data expands beyond the bounds of the drive's cache.

Although the S4 cleaner is slightly different, it was expected to behave similarly to a standard LFS cleaner, which has up to an approximate 34% decrease in performance [Seltzer95]. The S4 cleaner is slightly more intrusive, degrading performance by approximately 50% in the worst case. The greater degradation is attributed mainly to the additional reads necessary when cleaning objects rather than segments. In addition, the S4 cleaner has not been tuned and does not include known techniques for reducing cleaner performance problems [Matthews97].

4.5.1.4 Overhead of the S4 Audit Log

In addition to versioning, self-securing storage devices keep an audit log of all connections and commands sent to the drive. Recording this audit log of events has some cost. In the worst case, all data written to the disk belongs to the audit log. In this case, one disk write is expected approximately every 750 operations. In the best case, large writes, the audit log overhead is almost non-existent, since the writes of the audit log blocks are hidden in the segment writes of the requests. For the macrobenchmarks, the performance penalty ranged between 1% and 3%.

For a more focused view of this overhead, a set of micro-benchmarks were run with audit logging enabled and disabled. The micro-benchmarks proceed in three phases: creation of 10,000 1KB files (split across 10 directories), reads of the newly created files in creation order, and deletion of the files in creation order.

Figure 4.6 shows the results. The create and delete phases exhibit a 2.8% and 2.9% decrease in performance, respectively, and the read phase exhibits a 7.2% decrease in performance. Read performance suffers a larger penalty because the audit log and data log blocks become interwoven in the create phase reducing the number of files packed into each segment. This in turn increases the number of segment reads needed.

4.5.1.5 Fundamental Performance Costs

There are three fundamental performance costs of self-securing storage: versioning, auditing, and garbage collection. Versioning can be achieved at virtually no cost by combining journal-based metadata with the LFS structure. Auditing creates a small performance penalty of 1% to 3%, according to application benchmarks. The final performance cost, garbage collection, is more difficult to quantify. The extra overhead of S4 cleaning in comparison to standard LFS cleaning comes mainly from the difference in utilized space due to the history pool.

The worst-case performance penalty for garbage collection in S4 can be estimated by comparing the cleaning overhead at two space utilizations: the space utilized by the active set of objects and the space utilized by the active set combined with the history pool. For example, assume that the active set utilizes 60% of the drive's space and the history pool another 20%. For PostMark, the cleaning overhead is the difference between cleaning performance and standard performance seen at a given space utilization in Figure 4.5. For 60% utilization, the cleaning overhead is 43%. For 80% utilization, it is 53%. Thus, in this example, the extra cleaning overhead caused by keeping the history pool is 10%.

There are several possibilities for reducing cleaner overhead for all space utilizations. With expected detection windows ranging into the hundreds of days, it is likely that the history pool can be extended until such a time that the drive becomes idle. During idle time, the cleaner can run with no observable overhead [Blackwell95]. Also, recent research into technologies such as freeblock scheduling offer standard LFS cleaning at almost no cost [Lumb00]. This technique could be extended for cleaning in S4.

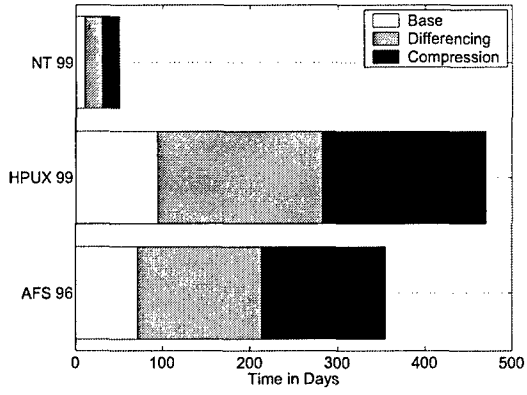


Figure 4.7. Projected Detection Window. The expected detection window that could be provided by utilizing 10GB of a modern disk drive. This conservative history pool would consume only 20% of a 50GB disk's total capacity. The baseline number represents the projected number of days worth of history information that can be maintained within this 10GB of space. The gray regions show the projected increase that cross-version differencing would provide. The black regions show the further increase expected from using compression in addition to differencing.

4.5.2 Capacity Requirements

To evaluate the size of the detection window that can be provided, three recent workload studies were examined. Figure 4.7 shows the results of approximations based on worst-case write behavior. Spasojevic and Satyanarayanan's AFS trace study [Spasojevic96] reports approximately 143MB per day of write traffic per file server. The AFS study was conducted using 70 servers (consisting of 32; 000 cells) distributed across the wide area, containing a total of 200GB of data. Based on this study, using just 20% of a modern 50GB disk would yield over 70 days of history data. Even if the writes consume 1GB per day per server, as was seen by Vogels' Windows NT file usage study [Vogels99], 10 days worth of history data can be provided. The NT study consisted of 45 machines split into personal, shared, and administrative domains running workloads of scientific processing, development, and other administrative tasks. Santry, et al. [Santry99] report a write data rate of 110MB per day. In this case, over 90 days of data could be kept. Their environment consisted of a single file system holding 15GB of data that was being used by a dozen researchers for development.

Much work has been done in evaluating the efficiency of differencing and compression [Burns96, Burrows94, Burrows 92]. To briefly explore the potential benefits for S4, its code base was retrieved from the CVS repository at a single point each day for a week. After compiling the code, both differencing and differencing with compression were applied between each tree and its direct neighbor in time using Xdelta [MacDonald00, MacDonald98]. After applying differencing, the space efficiency increased by 200%. Applying compression added an additional 200% for a total space efficiency of 500%. These results are in line with previous work. Applying these estimates to the above workloads indicates that a 10GB history pool can provide a detection window of between 50 and 470 days.

4.6 Metadata Efficiency in Versioning File Systems

Self-securing storage [Strunk00] uses versioning to enable storage servers to internally retain file versions and provide detailed information for post-intrusion diagnosis and recovery of compromised client systems [Strunk02]. We envision self-securing storage servers that retain every version of every file, where every modification (e.g., a WRITE operation or an attribute change) creates a new version. Such *comprehensive versioning* maximizes the information available for post-intrusion diagnosis. Specifically, it avoids pruning away file versions, since this might obscure intruder actions. For self-securing storage, *pruning techniques* are particularly dangerous when they rely on client-provided information, such as CLOSE operations—the versioning is being done specifically to protect stored data from malicious clients.

Obviously, finite storage capacities will limit the duration of time over which comprehensive versioning is possible. To be effective for intrusion diagnosis and recovery, this duration must be greater than the intrusion detection latency (i.e., the time from an intrusion to when it is detected). We refer to the desired duration as the *detection window*. In practice, the duration is limited by the rate of data change and the space efficiency of the versioning system. The rate of data change is an inherent aspect of a given environment, and an analysis of several real environments suggests that detection windows of several weeks or more can be achieved with only a 20% cost in storage capacity [Strunk00].

Sections 4.1 and 4.2 describe a prototype self-securing storage system. By using standard copy-on-write and a log-structured data organization, the prototype provided comprehensive versioning with minimal performance overhead (<10%) and reasonable space efficiency. In that work, we discovered that a key design requirement is efficient encoding of metadata versions (the additional information required to track the data versions). While copy-on-write reduces data versioning costs, conventional versioning implementations still involve one or more new metadata blocks per version. On average, the metadata versions require as much space as the versioned data, halving the achievable detection window. Even with less comprehensive versioning, such as Elephant [Santry99] or VMS [McCoy90], the metadata history can become almost (≈80%) as large as the data history.

This half of the chapter describes and evaluates two methods of storing metadata versions more compactly: journal-based metadata and multiversion b-trees. Journal-based meta-data encodes each version of a file's metadata in a journal entry. Each entry describes the difference between two versions, allowing the system to roll-back to the earlier version of the metadata. Multiversion b-trees retain all versions of a metadata structure within a single tree. Each entry in the tree is marked with a timestamp indicating the time over which the entry is valid.

The two mechanisms have different strengths and weaknesses. We discuss these and describe how both techniques are integrated into a comprehensive versioning file system called CVFS. CVFS uses journal-based meta-data for inodes and indirect blocks to encode changes to attributes and file data pointers; doing so reduces the space used for their histories by 80%. CVFS implements directories as multiversion b-trees to encode additions and removals of directory entries; doing so reduces the space used for their histories by 99%. Combined, these mechanisms nearly double the potential detection window over conventional versioning mechanisms, without increasing the access time to current versions of the data.

Journal-based metadata and multiversion b-trees are also valuable for conventional versioning systems. Using these mechanisms with on-close versioning and snapshots would provide similar reductions in versioned metadata. For on-close versioning, this reduces the total space required by nearly 35%, thereby reducing the pressure to prune version histories. Identifying solid heuristics for such pruning remains an open area of research [Santry99], and less pruning means fewer opportunities to mistakenly prune important versions.

4.7 Versioning and Space Efficiency

Every modification to a file inherently results in a new version of the file. Instead of replacing the previous version with the new one, a *versioning file system* retains both. Users of such a system can then access any historical versions that the system keeps as well as the most recent one. This section discusses uses of versioning, techniques for managing the associated capacity costs, and our goal of minimizing the metadata required to track file versions.

4.7.1 Uses of Versioning

File versioning offers several benefits to both users and system administrators. These benefits can be grouped into three categories: recovery from user mistakes, recovery from system corruption, and analysis of historical changes. Each category stresses different features of the versioning system beneath it.

Recovery from user mistakes: Human users make mistakes, such as deleting or erroneously modifying files. Versioning can help [Hagmann87, McCoy90, Santry99]. Recovery from such mistakes usually starts with some a priori knowledge about the nature of the mistake. Often, the exact file that should be recovered is known. Additionally, there are only certain versions that are of any value to the user; intermediate versions that contain incomplete data are useless. Therefore, versioning aimed at recovery from user mistakes should focus on retaining key versions of important files.

Recovery from system corruption: When a system becomes corrupted, administrators generally have no knowledge about the scope of the damage. Because of this, they restore the entire state of the file system from some well-known "good" time. A common versioning technique to help with this is the online *snapshot*. Like a backup, a snapshot contains a version of every file in the system at a particular time. Thus, snapshot systems present sets of known-valid system images at a set of well-known times.

Analysis of historical changes: A history of versions can help answer questions about how a file reached a certain state. For example, version control systems (e.g., RCS [Tichy80], CVS [Grune]) keep a complete record of committed changes to specific files. In addition to selective recovery, this record allows developers to figure out who made specific changes and when those changes were made. Similarly, self-securing storage seeks to enable post-intrusion diagnosis by providing a record of what happened to stored files before, during, and after an intrusion. We believe that every version of every file should be kept. Otherwise, intruders who learn the pruning heuristic will leverage this information to prune any file versions that might disclose their activities. For example, intruders may make changes and then quickly revert them once damage is caused in order to hide their tracks. With a complete history, administrators can determine which files were changed and estimate damage. Further, they can answer (or at least construct informed hypotheses for) questions such as "When and how did the intruder get in?" and "What was their goal?" [Strunk02].

4.7.2 Pruning Heuristics

A true comprehensive versioning system keeps all versions of all files for all time. Such a system could support all three goals described above. Unfortunately, storing this much information is not practical. As a result, all versioning systems use *pruning heuristics*. These pruning heuristics determine when versions should be created and when they should be removed. In other words, pruning heuristics determine which versions to keep from the total set of versions that would be available in a comprehensive versioning system.

Common Heuristics: A common pruning technique in versioning file systems is *on-close* versioning. This technique keeps only the last version of a file from each session; that is, each CLOSE of a file creates a distinct version. For example, the VMS file system [McCoy90] retains a fixed number of versions for each file. VMS's pruning heuristic creates a version after each CLOSE of a file and, if the file already has the maximum number of versions, removes the oldest remaining version of the file. The more recent Elephant file system [Santry99] also creates new versions after each CLOSE; however, it makes additional pruning decisions based on a set of rules derived from observed user behavior.

Version control systems prune in two ways. First, they retain only those versions explicitly committed by a user. Second, they retain versions for only an explicitly-chosen subset of the files on a system.

By design, snapshot systems like WAFL [Hitz94] and Venti/Plan9 [Quinlan02] prune all of the versions of files that are made between snapshots. Generally, these systems only create and delete snapshots on request, meaning that the system's administrator decides most aspects of the pruning heuristic.

Information Loss: Pruning heuristics act as a form of lossy compression. Rather than storing every version of a file, these heuristics throw some data away to save space. The result is that, just as a JPEG file loses some of its visual clarity with lossy compression, pruning heuristics reduce the clarity of the actions that were performed on the file.

Although this loss of information could result in annoyances for users and administrators attempting to recover data, the real problem arises when versioning is used to analyze historical changes. When versioning for intrusion survival, as in the case of self-securing storage, pruning heuristics create holes in the administrator's view of the system. Even creating a version on every CLOSE is not enough, as malicious users can leverage this heuristic to hide their actions (e.g., storing exploit tools in an open file and then truncating the file to zero before closing it).

To avoid traditional pruning heuristics, self-securing storage employs comprehensive versioning over a fixed window of time, expiring versions once they become older than the given window. This detection window can be thought of as the amount of time that an administrator has to detect, diagnose, and recover from an intrusion. As long as an intrusion is detected within the window, the administrator has access to the entire sequence of modifications since the intrusion.

4.7.3 Lossless Version Compression

For a system to avoid pruning heuristics, even over a fixed window of time, it needs some form of loss-less version compression. Lossless version compression can also be combined with pruning heuristics to provide further space reductions in conventional systems. To maximize the benefits, a system must attempt to compress both versioned data and versioned metadata.

Data: Data block sharing is a common form of loss-less compression in versioning systems. Unchanged data blocks are shared between versions by having their individual metadata point to the same physical block. Copy-on-write is used to avoid corrupting old versions if the block is modified.

An improvement on block sharing is byte-range differencing between versions. Rather than keeping the data blocks that have changed, the system keeps the bytes that have changed [MacDonald98]. This is especially useful in situations where a small change is made to the file. For example, if a single byte is inserted at the beginning of a file, a block sharing system keeps two full copies of the entire file (since the data of every block in the file is shifted forward by one byte); for the same scenario, a differencing system only stores the single byte that was added and a small description of the change.

Another recent improvement in data compression is hash-based data storage [Muthitacharoen01, Quinlan02]. These methods recognize identical blocks or ranges of data across the system and store only one copy of the data. This method is quite effective for snapshot versioning systems, and could likely be applied to other versioning systems with similar results.

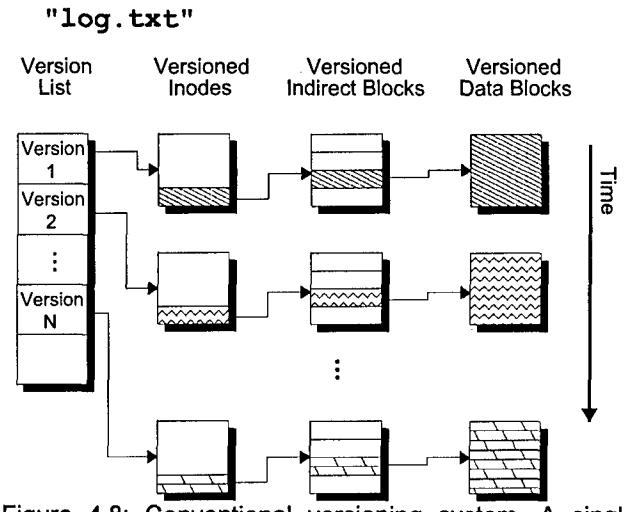


Figure 4.8: Conventional versioning system. A single logical block of file "log.txt" is overwritten several times. With each new version, new versions of the indirect block and inode that reference it are created. Although only a single pointer has changed in both the indirect block and the inode, they must be rewritten entirely, since they require new versions. The system tracks each version with a pointer to that version's inode.

Metadata: Conventional versioning file systems keep a full copy of the file metadata with each version. While it simplifies version access, this method quickly exhausts capacity, since even small changes to file data or attributes result in a new copy of the metadata.

Figure 4.8 shows an example of how the space overhead of versioned metadata can become a problem in a conventional versioning system. In this example, a program is writing small log entries to the end of a large file. Since several log entries fit within a single data block, appending entries to the end of the file produces several different versions of the same block. Because each versioned data block has a different location on disk, the system must create a new version of the indirect block to track its location. In addition, the system must write a new version of the inode to track the location of the versioned indirect block. Since any data or metadata change will always result in a new version of the inode, each version is tracked using a pointer to that version's inode. Thus, writing a single data block results in a new indirect block, a new inode, and an entry in the version list, resulting in more metadata being written than data.

Access patterns that create such metadata versioning problems are common. Many applications create or modify files piece by piece. In addition, distributed file systems such as NFS create this behavior by breaking large updates of a file into separate, block-sized updates. Since there is no way for the server to determine if these block-sized writes are one large update or several small ones, each must be treated as a separate update, resulting in several new versions of the file.

Again, the solution to this problem is some form of differencing between the versions. Mechanisms for creating and storing differences of metadata versions are the main focus of this work.

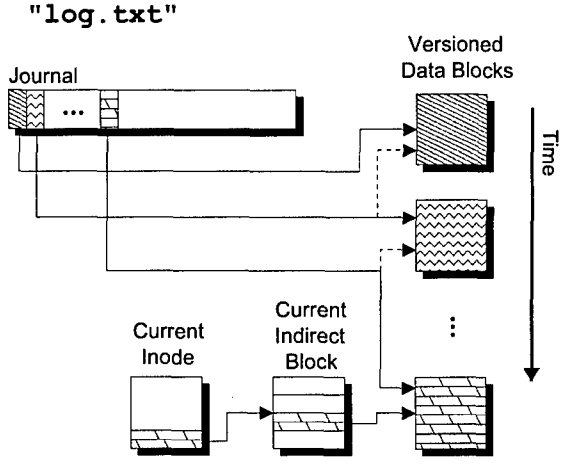
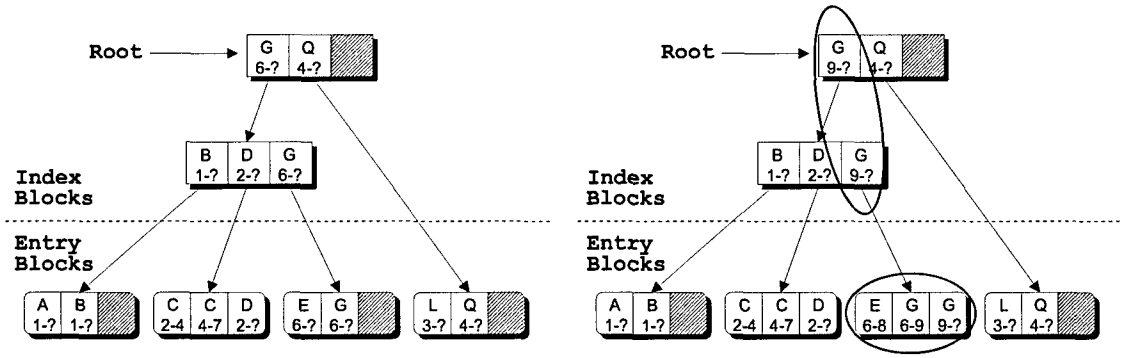


Figure 4.9: Journal-based metadata system. As in Figure 4.8, a single logical block of file "log.txt" is being overwritten several times. All versions of the data block are retained by recording each in a journal entry, which points to both the new block and the overwritten one. Only the current version of the inode and indirect block are kept, significantly reducing the amount of space required for metadata.

4.7.4 Objective

In a perfect world we could keep all versions of all files for an infinite amount of time with no impact on performance. This is obviously not possible. The objective of this work is to minimize the space overhead of versioned metadata. For self-securing storage, doing so will increase



(a) initial tree structure (b) After removal of E and update of G.

Figure 4.10: Multiversion b-tree. This figure shows the layout of a multiversion b-tree. Each entry of the tree is designated by a <user-key, timestamp> tuple which acts as a key for the entry. A question mark (?) in the timestamp indicates that the entry is valid through the current time. Different versions of an entry are separate entries using the same user-key with different timestamps. Entries are packed into entry blocks, which are tracked using index blocks. Each index pointer holds the key of the last entry along the subtree that it points to.

the detection window. For other versioning purposes, doing so will reduce the pressure to prune. Because this space reduction will require compressing metadata versions, it is also important that the performance overhead of both version creation and version access be minimized.

4.8 Efficient Metadata Versioning

One characteristic of versioned metadata is that the actual changes to the metadata between versions are generally quite small. In Figure 4.8, although an inode and an indirect block are written with each new version of the file, the only change to the metadata is an update to a single block pointer. The system can leverage these small changes to provide much more space-efficient metadata versioning. This section describes two methods that leverage small metadata modifications, and Section 4.9 describes an implementation of these solutions.

4.8.1 Journal-based Metadata

Journal-based metadata maintains a full copy of the current version's metadata and a journal of each previous metadata change. To recreate old versions of the meta-data, each change is undone backward through the journal until the desired version is recreated. This process of undoing metadata changes is referred to as *journal rollback*.

Figure 4.9 illustrates how journal-based metadata works in the example of writing log entries. Just as in Figure 4.8, the system writes a new data block for each version; however, in journal-based metadata, these blocks are tracked using small journal entries that track the locations of the new and old blocks. By keeping the current version of the metadata up-to-date, the journal entries can be rolled-back to any previous version of the file.

In addition to storing version information, the journal can be used as a write-ahead log for metadata consistency, just as in a conventional journaling file system. To do so, the new block pointer must be recorded in addition to the old. Using this, a journal-based metadata implementation can safely maintain the current version of the metadata in memory, flushing it to disk only when it is forced from the cache.

Space vs. Performance: Journal-based metadata is more space efficient than conventional versioning. However, it must pay a performance penalty for recreating old versions of the meta-data. Since each entry written between the current version and the requested version must be read and rolled-back, there is a linear relation between the number of changes to a file and the performance penalty for recreating old versions.

One way the system can reduce this overhead is to *checkpoint* a full copy of a file's metadata to the disk occasionally. By storing checkpoints and remembering their locations, a system can start journal roll-back from the closest checkpoint in time rather than always starting with the current version. The frequency with which these checkpoints are written dictates the space/performance trade-off. If the system keeps a checkpoint with each modification, journal-based metadata performs like a conventional versioning scheme (using the most space, but offering the best back-in-time performance). However, if no checkpoints are written, the only full instance of the metadata is the current version, resulting in the lowest space utilization but reduced back-in-time performance.

4.8.2 Multiversion B-trees

A multiversion b-tree is a variation on standard b-trees that keeps old versions of entries in the tree [Becker96]. As in a standard b-tree, an entry in a multiversion b-tree contains a key/data pair; however, the key consists of both a user-defined key and the time at which the entry was written. With the addition of this time-stamp, the key for each version of an entry becomes unique. Having unique keys means that entries within the tree are never overwritten; therefore, multiversion b-trees can have the same basic structure and operations as a standard b-tree. To facilitate current version lookups, entries are sorted first by the user-defined key and then by the timestamp.

Figure 4.10a shows an example of a multiversion b-tree. Each entry contains both the user-defined key and the time over which the entry is valid. The entries are packed into entry blocks, which act as the leaf nodes of the tree. The entry blocks are tracked using index blocks, just as in standard b+trees. In this example, each pointer in the index block references the last entry of the subtree beneath it. So in the case of the root block, the *G* subtree holds all entries with values less than or equal to *G*, with ⟨*G*, 6–?⟩ as its last entry. The *Q* subtree holds all entries with values between *G* and *Q*, with ⟨*Q*, 4–?⟩ as its last entry.

Figure 4.10b shows the tree after a remove of entry *E* and an update to entry *G*. When entry *E* is removed at time *8*, the only change is an update to the entry's timestamp. This indicates that *E* is only valid from time *6* through time *8*. When entry *G* is updated at time *9*, a new entry is created and associated with the new data. Also, the old entry for *G* must be updated to indicate its bounded window of validity. In this case, the index blocks must also be updated to reflect the new state of the subtree, since the last entry of the subtree has changed.

Since both current and history entries are stored in the same tree, accesses to old and current versions have the same performance. For this reason, large numbers of history entries can decrease the performance of accessing current entries.

4.8.3 Solution Comparison

Both journal-based metadata and multiversion b-trees reduce the space utilization of versioning but incur some performance penalty. Journal-based metadata pays with reduced back-in-time performance. Multiversion b-trees pay with reduced current version performance.

Because the two mechanisms have different drawbacks, they each perform certain operations more efficiently. As mentioned above, the number of history entries in a multiversion b-tree can adversely affect the performance of accessing the current version. This emerges in two situations: linear scan operations and files with a large number of versions. The penalty on lookup operations is reduced by the logarithmic nature of the tree structure, but large numbers of history entries can increase tree depth. Linear scanning of all current entries requires accessing every entry in the tree, which becomes expensive if the number of history entries is high. In both of these cases, it is better to use journal-based metadata.

When lookup of a single entry is common or history access time is important, it is preferable to use multiversion b-trees. Using a multiversion b-tree, all versions of the entry are located together in the tree and have logarithmic lookup time (for both current and history entries), giving a performance benefit over the linear roll-back operation required by journal-based metadata.

4.9 Implementation

We have integrated journal-based metadata and multiversion b-trees into a comprehensive versioning file system, called CVFS. CVFS provides comprehensive versioning within our self-securing NFS server prototype. Because of this, some of our design decisions (such as the implementation of a strict detection window) are specific to self-securing storage. Regardless, these structures would be effective in most versioning systems.

4.9.1 Overview

Since current versions of file data must not be overwritten in a comprehensive versioning system, CVFS uses a log-structured data layout similar to LFS [Rosenblum91]. Not only does this eliminate overwriting of old versions on disk, but it also improves update performance by combining data and metadata updates into a single disk write.

CVFS uses both mechanisms described in Section 4.8. It uses journal-based metadata to version file data pointers and file attributes, and multiversion b-trees to version directory entries. We chose this division of methods based on the expected usage patterns of each. Assuming many versions of file attributes and a need to access them in their entirety most of the time, we decided that journal-based metadata would be more efficient. Directories, on the other hand, are updated less frequently than file meta-data and a large fraction of operations are entry lookup rather than full listing. Thus, the cost of having history entries within the tree is expected to be lower.

Since the only pruning heuristic in CVFS is expiration, it requires a cleaner to find and remove expired versions. Although CVFS's background cleaner is not described in detail here, its implementation closely resembles the cleaner in LFS. The only added complication is that, when moving a data block in a versioning system, the cleaner must update all of the metadata versions that point to the block. Locating and modifying all of this metadata can be expensive. To address this problem, each data block on the disk is assigned a virtual block number. This allows us to move the physical location of the data and only have to update a single pointer within a virtual indirection table, rather than all of the associated metadata.

4.9.2 Layout and Allocation

Because of CVFS's log-structured format, disk space is managed in contiguous sets of disk blocks called *segments*. At any particular time, there is a single *write segment*. All data block allocations are done within this segment. Once the segment is completely allocated, a new write segment is chosen. Free segments on the disk are tracked using a bitmap.

As CVFS performs allocations from the write segment, the allocated blocks are marked as either journal blocks or data blocks. Journal blocks hold journal entries, and they contain pointers that string all of the journal blocks together into a single contiguous journal. Data blocks contain file data or metadata checkpoints.

CVFS uses inodes to store a file's metadata, including file size, access permissions, creation time, modification time, and the time of the oldest version still stored on the disk. The inode also holds direct and indirect data pointers for the associated file or directory. CVFS tracks inodes with a unique inode number. This inode number indexes into a table of inode pointers that are kept at a fixed location on the disk. Each pointer holds the block number of the most current metadata checkpoint for that file, which is guaranteed to hold the most current version of the file's inode. The in-memory copy of an inode is always kept up-to-date with the current version, allowing quick access for standard operations. To ensure that the current version can always be accessed directly off the disk, CVFS checkpoints the inode to disk on a cache flush.

Entry Type	Description	Cause
Attribute	Holds new inode attribute information	Inode change
Delete	Holds inode number and delete time	Inode change
Truncate	Holds the new size of the file	File data change
Write	Points to the new file data	File data change
Checkpoint	Points to checkpointed metadata	Metadata checkpoint / Inode change

Table 4.2: Journal entry types. This table lists the five types of journal entry. Journal entries are written when inodes are modified, file data is modified, or file metadata is flushed from the cache.

4.9.3 The Journal

The string of journal blocks that runs through the segments of the disk is called the *journal*. Each journal block holds several time-ordered, variably-sized journal entries. CVFS uses the journal to implement both conventional file system journaling (a.k.a. write-ahead logging) and journal-based metadata.

Each journal entry contains information specific to a single change to a particular file. This information must be enough to do both roll-forward and roll-back of the metadata. Roll-forward is needed for update consistency in the face of failures. Roll-back is needed to reconstruct old versions. Each entry also contains the time at which the entry was written and a pointer to the location of the previous entry that applies to this particular file. This pointer allows us to trace the changes of a single file through time.

Table 4.2 lists the five different types of journal entries. CVFS writes entries in three different cases: inode modifications (creation, deletion, and attribute updates), data modifications (writing or truncating file data), and meta-data checkpoints (due to a cache flush or history optimization).

4.9.4 Metadata

There are three types of file metadata that can be altered individually: inode attributes, file data pointers, and directory entries. Each has characteristics that match it to a particular method of metadata versioning.

Inode Attributes: There are four operations that act upon inode attributes: creation, deletion, attribute updates, and attribute lookups.

CVFS creates inodes by building an initial copy of the new inode and checkpointing it to the disk. Once this checkpoint completes and the inode pointer is updated, the file is accessible. The initial checkpoint entry is required because the inode cannot be read through the inode pointer table until a checkpoint occurs. CVFS's default checkpointing policy bounds the back-in-time access performance to approximately 150ms as is described in Section 4.10.

To delete an inode, CVFS writes a "delete" journal entry, which notes the inode number of the file being deleted. A flag is also set in the current version of the inode, specifying that the file was deleted, since the deleted inode cannot actually be removed from the disk until it expires.

CVFS stores attribute modifications entirely within a journal entry. This journal entry contains the value of the changed inode attributes both before and after the modification. Therefore, an attribute update involves writing a single journal entry, and updating the current version of the inode in memory.

CVFS accesses the current version of the attributes by reading in the current inode, since all of the attributes are stored within it. To access old versions of the attributes, CVFS traverses the journal entries searching for modifications that affect the attributes of that particular inode. Once roll-back is complete, the system is left with a copy of the attributes at the requested point in time.

File Data Pointers: CVFS tracks file data locations using direct and indirect pointers [McKusick84]. Each file's inode contains thirty direct pointers, as well as one single, one double and one triple indirect pointer.

When CVFS writes to a file, it allocates space for the new data within the current write segment and creates a "write" journal entry. The journal entry contains pointers to the data blocks within the segment, the range of logical block numbers that the data covers, the old size of the file, and pointers to the old data blocks that were overwritten (if there were any). Once the journal entry is allocated, CVFS updates the current version of the meta-data to point at the new data.

If a write is larger than the amount of data that will fit within the current write segment, CVFS breaks the write into several data/journal entry pairs across different segments. This compartmentalization simplifies cleaning.

To truncate a file, CVFS first checkpoints the file to the log. This is necessary because CVFS must be able to locate truncated indirect blocks when reading back-in-time. If they are not checkpointed, then the information in them will be lost during the truncate; although earlier journal entries could be used to recreate this information, such entries could expire and leave the detection window, resulting in lost information. Once the checkpoint is complete, a "truncate" journal entry is created containing both a pointer to the checkpointed metadata and the new size of the file.

To access current file data, CVFS finds the most current inode and reads the data pointers directly, since they are guaranteed to be up-to-date. To access historical data versions, CVFS uses a combination of checkpoint tracking and journal roll-back to recreate the desired version of the requested data pointers. CVFS's checkpoint tracking and journal roll-back work together in the following way. Assume a user wishes to read data from a file at time T. First, CVFS locates the oldest checkpoint it is tracking with time T_c such that $T_c \geq T$. Next, it searches backward from that checkpoint through the journal looking for changes to the block numbers being read. If it finds an older version of a block that applies, it will use that. Otherwise, it reads the block from the checkpointed metadata.

To illustrate this journal rollback, Figure 4.11 shows a sequence of updates to block 3 of inode 4 interspersed with checkpoints of inode 4. Each block update and inode checkpoint is labeled with the time that it was written. To read block 3 at time $T_1 = 12$, CVFS first reads the checkpoint at time $t = 18$, then reads journal entries to see if a different data block should be used. In this case, it finds that the block was overwritten at time $t = 15$, and so returns the older block written at time $t = 10$. In the case of time $T_2 = 5$, CVFS starts with the checkpoint at time $t = 7$, and then reads the journal entry, and realizes that no such block existed at time $t = 5$.

Directory Entries: Each directory in CVFS is implemented as a multiversion b-tree. Each entry in the tree represents a directory entry; therefore, each b-tree entry must contain the entry's name, the inode number of the associated file, and the time over which the entry is valid. Each entry also contains a fixed-size hash of the name. Although the actual name must be used as the key while searching through the entry blocks, this fixed-size hash allows the index blocks to use space-efficient fixed-size keys.

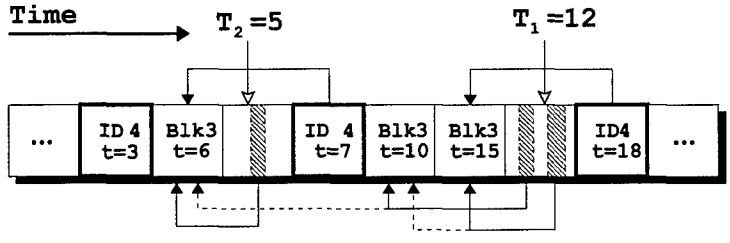


Figure 4.11: Back-in-time access. This figure shows a series of checkpoints of inode 4 (highlighted with dark border) and updates to block 3 of inode 4. Each checkpoint and update is marked with a time t at which the event occurred. Each checkpoint holds a pointer to the block that is valid at the time of the checkpoint. Each update is accompanied by a journal entry (marked by thin, grey boxes) which holds a pointer to the new block (solid arrow) and the old block that it overwrote (dashed arrow, if one exists).

CVFS uses a full data block for each entry block of the tree, and sorts the entries within it first by hash and then by time. Index nodes of the tree are also full data blocks consisting of a set of index pointers also sorted by hash and then by time. Each index pointer is a ⟨subtree, hash, time-range⟩ tuple, where *subtree* is a pointer to the appropriate child block, *hash* is the name hash of the last entry along the subtree, and *time-range* is the time over which that same entry is valid.

With this structure, lookup and listing operations on the directory are the same as with a standard b-tree, except that the requested time of the operation becomes part of the key. For example, in Figure 4.10a, a lookup of ⟨*C,5*⟩ searches through the tree for entries with name C, and then checks the time-ranges of each to determine the correct entry to return (in this case ⟨*C,4 – 7*⟩). A listing of the directory at time *5* would do an in-order tree traversal (just as in a standard b-tree), but would exclude any entries that are not valid at time *5*.

Insert, remove, and update are also very similar. Insert is identical, with the time-range of the new entry starting at the current time. Remove is an update of the time-range for the requested name. For example, in Figure 4.10b, entry *E* is removed at time *8*. Update is an atomic remove and insert of the same entry name. Also in Figure 4.10b, entry *G* is updated at time *9*. This involves atomically removing the old entry *G* at time *9* (updating the time-range), and inserting entry *G* at time *9* (the new entry ⟨*G,9 – ?*⟩).

Labyrinth Traces					
		Versioned Data	**Versioned Metadata**	**Metadata Savings**	**Total Savings**
Files:	Conventional versioning Journal-based metadata	123.4 GB 123.4 GB	142.4 GB 4.2 GB	97.1%	52.0%
Directories:	Conventional versioning Multiversion b-trees	— —	9.7 GB 0.044 GB	99.6%	99.6%
Total:	Conventional versioning CVFS	123.4 GB 123.4 GB	152.1 GB 4.244 GB	**97.2%**	**53.7%**
Lair Traces					
		Versioned Data	**Versioned Metadata**	**Metadata Savings**	**Total Savings**
Files:	Conventional versioning Journal-based metadata	74.5 GB 74.5 GB	34.8 GB 1.1 GB	96.8%	30.8%
Directo-ries:	Conventional versioning Multiversion b-trees	— —	1.8 GB 0.0064 GB	99.7%	99.7%
Total:	Conventional versioning CVFS	74.5 GB 74.5 GB	36.6 GB 1.1064 GB	**97.0%**	**32.0%**

Table 4.3: Space utilization. This table compares the space utilization of conventional versioning with CVFS, which uses journal-based metadata and multiversion b-trees. The space utilization for versioned data is identical for conventional versioning and journal-based metadata because neither address data beyond block sharing. Directories contain no versioned data because they are entirely a metadata construct.

4.10 Evaluation

The objective of this work is to reduce the space overheads of versioning without reducing the performance of current version access. Therefore, our evaluation of CVFS is done in two parts. First, we analyze the space utilization of CVFS. We find that using journal-based metadata and multiversion b-trees reduces space overhead for versioned metadata by over 80%. Second, we analyze the performance characteristics of CVFS. We find that it performs similarly to non-versioning systems for current version access, and that back-in-time performance can be bounded to acceptable levels.

4.10.1 Experimental Setup

For the evaluation, we used CVFS as the underlying file system for S4, our self-securing NFS server. S4 is a user-level NFS server written for Linux that uses the SCSI-generic interface to directly access the disk. S4 exports an NFSv2 interface and treats it as a security perimeter between the storage system and the client operating system. Although the NFSv2 specification requires that all changes be synchronous,

S4 also has an asynchronous mode of operation, allowing us to more thoroughly analyze the performance overheads of our metadata versioning techniques.

In all experiments, the client system has a 550 MHz Pentium III, 128 MB RAM, and a 3Com 3C905B 100 MB network adapter. The servers have two 700 MHz Pentium IIIs, 512 MB RAM, a 9 GB 10,000 RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro100 100 Mb network adapter. The client and server are on the same 100 MB network switch.

4.10.2 Space Utilization

We used two traces, labeled *Labyrinth* and *Lair*, to evaluate the space utilization of our system. The *Labyrinth* trace is from an NFS server at Carnegie Mellon holding the home directories and CVS repository that support the activities of approximately 30 graduate students and faculty; it records approximately 164 GB of data traffic to the NFS server over a one-month period. The *Lair* trace [Ellard03] is from a similar environment at Harvard; it records approximately 103 GB of data traffic over a one-week period. Both were captured via passive network monitoring.

We replayed each trace onto both a standard configuration of CVFS and a modified version of CVFS. The modified version simulates a conventional versioning system by checkpointing the metadata with each modification. It also performs copy-on-write of directory blocks, overwriting the entries in the new blocks (that is, it uses normal b-trees). By observing the amount of allocated data for each request, we calculated the exact overheads of our two metadata versioning schemes as compared to a conventional system.

Table 4.3 compares the space utilization of versioned files for the two traces using conventional versioning and journal-based metadata. There are two space overheads for file versioning: versioned data and versioned meta-data. The overhead of versioned data is the overwritten or deleted data blocks that are retained. In both conventional versioning and journal-based metadata, the versioned data consumes the same amount of space, since both schemes use block sharing for versioned data. The overhead of versioned metadata is the information needed to track the versioned data. For *Labyrinth*, the versioned metadata consumes as much space as the versioned data. For *Lair*, it consumes only half as much space as the versioned data, because *Lair* uses a larger block size; on average, twice as much data is overwritten with each WRITE.

Journal-based metadata: Journal-based metadata reduces the space required for versioned file metadata substantially. For the conventional system, versioned meta-data consists of copied inodes and sometimes indirect blocks. For journal-based metadata, it is the log entries that allow recreation of old versions plus any checkpoints used to improve back-in-time performance (see Section 4.10.3). For both traces, this results in 97% reduction of space required for versioned metadata.

Multiversion b-trees: Using multiversion b-trees for directories provides even larger space utilization reductions. Because directories are a metadata construct, there is no versioned data. The overhead of versioned meta-data in directories is the space used to store the overwritten and deleted directory entries. In a conventional versioning system, each entry creation, modification, or removal results in a new block being written that contains the change. Since the entire block must be kept over the detection window, it results in approximately

9.7 GB of space for versioned entries in the *Labyrinth* trace and 1.8 GB in the *Lair* trace. With multiversion b-trees, the only overhead is keeping the extra entries in the tree, which results in approximately 45 MB and 7 MB of space for versioned entries in the respective traces.

Labyrinth Traces							
		Versioned Data	**Versioned File Metadata**	**Versioned Directories**	**Version Ratio**	**Metadata Savings**	**Total Savings**
Comprehensive:	Conventional	123.4 GB	142.4 GB	9.7 GB			
	CVFS	123.4 GB	4.2 GB	0.044 GB	1:1	97%	54%
On close():	Conventional	55.3 GB	30.6 GB	2.4 GB			
	CVFS	55.3 GB	2.1 GB	0.012 GB	1:2.8	94%	35%
6 minute Snapshots:	Conventional	53.2 GB	11.0 GB	2.4 GB			
	CVFS	53.2 GB	1.3 GB	0.012 GB	1:11.7	90%	18%
1 hour Snapshots:	Conventional	49.7 GB	5.1 GB	2.4 GB			
	CVFS	49.7 GB	0.74 GB	0.012 GB	1:20.8	90%	12%
Lair Traces							
		Versioned Data	**Versioned File Metadata**	**Versioned Directories**	**Version Ratio**	**Metadata Savings**	**Total Savings**
Comprehensive:	Conventional	74.5 GB	34.8 GB	1.79 GB			
	CVFS	74.5 GB	1.1 GB	0.0064 GB	1:1	97%	32%
On close():	Conventional	40.3 GB	6.1 GB	0.75 GB			
	CVFS	40.3 GB	0.57 GB	0.0032	1:2.9	91%	13%
6 minute Snapshots:	Conventional	38.2 GB	3.0 GB	0.75 GB			
	CVFS	38.2 GB	0.36 GB	0.0032	1:11.2	88%	8%
1 hour Snapshots:	Conventional	36.2 GB	2.0 GB	0.75 GB			
	CVFS	36.2 GB	0.26 GB	0.0032	1:15.6	87%	6%

Table 4.4: Benefits for different versioning schemes. This table shows the benefits of journal-based meta-data for three versioning schemes that use pruning heuristics. For each scheme it compares conventional versioning with CVFS's journal-based metadata and multiversion b-trees, showing the versioned meta-data sizes, the corresponding metadata savings, and the total space savings. It also displays the ratio of versions to file modifications; more modifications per version generally reduces both the importance and the compressibility of versioned metadata.

Other Versioning Schemes: We used the *Labyrinth* and *Lair* traces to compute the space that would be required to track versions in three other versioning schemes: versioning on every file CLOSE, taking systems snapshots every 6 minutes, and taking system snapshots every hour. In order to simulate open-close semantics with our NFS server, we insert a CLOSE call after sequences of operations on a given file that are followed by 500ms of inactivity to that file.

Table 4.4 shows the benefits of CVFS's mechanisms for the three versioning schemes mentioned above. For each scheme, the table also shows the ratio of file versions-to-modifications (e.g., in comprehensive versioning, each modification results in a new version, so the ratio is 1:1). For on-close versioning in the *Labyrinth* trace, conventional versioning requires 55% as much space for versioned metadata as versioned data, meaning that reduction can still provide large benefits. As the versioned metadata to versioned data ratio decreases and as the version ratio increases, the overall benefits of versioned metadata compression drop.

Table 4.4 identifies the benefits of both journal-based meta-data (for "Versioned File Metadata") and multiversion b-trees (for "Versioned Directories"). For both, the meta-data compression ratios are similar to those for comprehensive versioning. The journal-based metadata ratio drops slightly as the version ratio increases, because capturing more changes to the file metadata moves the journal entry size closer to the actual metadata size. The multiversion b-tree ratio is lower because a most of the directory updates fall into one of two categories: entries that are permanently added or temporary entries that are created and then rapidly renamed or deleted. For this reason, the number of versioned entries is lower for other versioning schemes; although multiversion b-trees use less space, the effect on overall savings is reduced.

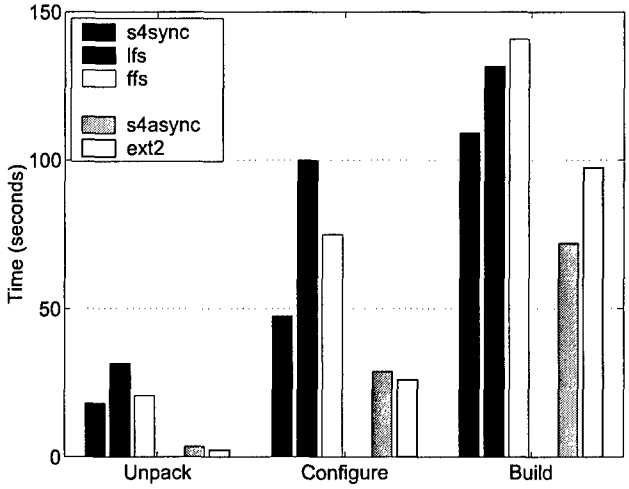


Figure 4.12: SSH comparison. This figure shows the performance of five systems on the unpack, configure, and build phases of the SSH-build benchmark. Performance is measured in the elapsed time of the phase. Each result is the average of 15 runs, and all variances are under .5s with the exception of the build phases of ffs and lfs which had variances of 37.6s and 65.8s respectively.

4.10.3 Performance Overheads

The performance evaluation is done in three parts. First, we compare the S4 prototype to non-versioning systems using several macro benchmarks. Second, we measure the back-in-time performance characteristics of journal-based metadata. Third, we measure the general performance characteristics of multiversion b-trees.

General Comparison: The purpose of the general comparison is to verify that the S4 prototype performs comparably to non-versioning systems. Since part of our objective is to avoid undue performance overheads for versioning, it is important that we confirm that the prototype performs reasonably relative to similar systems. To evaluate the performance relationship between S4 and non-versioning systems, we ran two macro benchmarks designed to simulate realistic workloads.

For both, we compare S4 in both synchronous and asynchronous modes against three other systems: a NetBSD NFS server running FFS, a NetBSD NFS server running LFS, and a Linux NFS server running EXT2. We chose to compare against BSD's LFS because it uses a log-structured layout similar to S4's. BSD's FFS and Linux's EXT2 use similar, more "traditional" file layout techniques that differ from S4's log-structured layout. It is not our intent to compare a LFS layout against other layouts, but rather to confirm that our implementation does not have any significant performance anomalies. To ensure this, a small discussion of the performance differences between the systems is given for each benchmark.

Each of these systems was measured using an NFS client running on Linux. Our S4 measurements use the S4 server and a Linux client. For "Linux," we run RedHat 6.1 with a 2.2.17 kernel. For "NetBSD," we run a stock NetBSD 1.5 installation.

To focus comparisons, the five setups should be viewed in two groups. BSD LFS, BSD FFS, and S4-sync all push updates to disk synchronously, as required by the NFSv2 specification. Linux EXT2 and S4-async do not; instead, updates are made in memory and propagated to disk in the background.

SSH-build [Seltzer00] was constructed as a replacement for the Andrew file system benchmark [Howard88]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

Figure 4.12 shows the SSH-build results for each of the five different systems. As we hoped, our S4 prototype performs similarly to the other systems measured.

LFS does significantly worse on unpack and configure because it has poor small write performance. This is due to the fact that NetBSD's LFS implementation uses a 1 MB segment size, and NetBSD's NFS

server requires a full sync of this segment with each modification; S4 uses a 64kB segment size, and supports partial segments. Adding these features to NetBSD's LFS implementation would result in performance similar to S4. FFS performs worse than S4 because FFS must update both a data block and inode with each file modification, which are in separate locations on the disk. EXT2 performs more closely to S4 in asynchronous mode because it fails to satisfy NFS's requirement of synchronous modifications. It does slightly better in the unpack and configure stages because it maintains no consistency guarantees, however it does worse in the build phase due to S4's segment-sized reads.

Postmark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [Katcher97]. It creates a large number of small randomly-sized files (between 512 B and 9 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction types are equal.

Figure 4.13 shows the Postmark results for the five server configurations. These show similar results to the SSH-build benchmark. Again, S4 performs comparably. In particular, LFS continues to perform poorly due to its small write performance penalty caused by its interaction with NFS. FFS still pays its performance penalty due to multiple updates per file create or delete. EXT2 performs even better in this benchmark because the random, small file accesses done in Postmark are not assisted by aggressive prefetching, unlike the sequential, larger accesses done during a compilation; however, S4 continues to pay the cost of doing larger accesses, while EXT2 does not.

Journal-based Metadata: Because the metadata structure of a file's current version is the same in both journal-based metadata and conventional versioning systems, their current version access times are identical. Given this, our performance measurements focus on the performance of back-in-time operations with journal-based metadata.

There are two main factors that affect the performance of back-in-time operations: checkpointing and clustering. Checkpointing refers to the frequency of metadata checkpoints. Since journal roll-back can begin with any checkpoint, CVFS keeps a list of metadata checkpoints for each file, allowing it to start roll-back from the closest checkpoint. The more frequently CVFS creates checkpoints, the better the back-in-time performance.

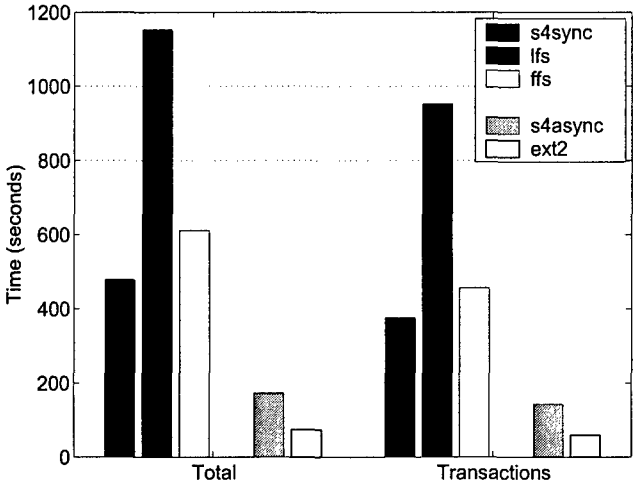


Figure 4.13: Postmark comparison. This figure shows the elapsed time for both the entire run of postmark and the transactions phase of postmark for the five test systems. Each result is the average of 15 runs, and all variances are under 1.5s.

Clustering refers to the physical distance between relevant journal entries. With CVFS's log-structured layout, if several changes are made to a file in a short span of time, then the journal entries for these changes are likely to be clustered together in a single segment. If several journal entries are clustered in a single segment together, then they are all read together, speeding up journal rollback. The "higher" the clustering, the better the performance is expected to be.

Figure 4.14 shows the back-in-time performance characteristics of journal-based metadata. This graph shows the access time in milliseconds for a particular version number of a file back-in-time. For example, in the worst-case, accessing the 60th version back-in-time would take

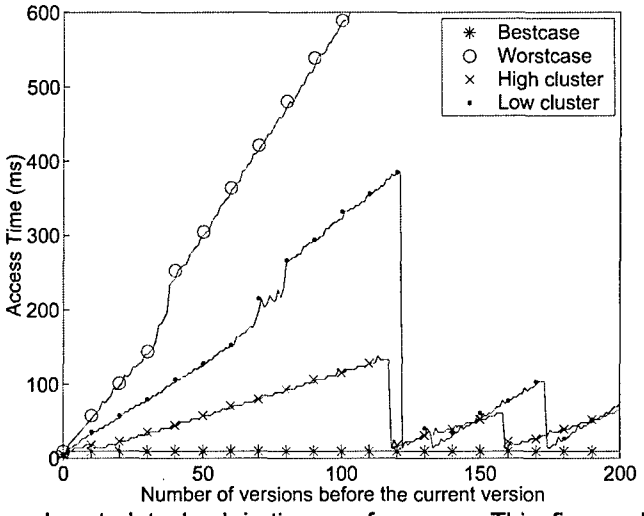


Figure 4.14: Journal-based metadata back-in-time performance. This figure shows several potential curves for back-in-time performance of accessing a single 1KB file. The worst-case is when journal roll-back is used exclusively, and each journal entry is in a separate segment on the disk. The best-case is if a checkpoint is available for each version, as in conventional versioning. The high and low clustering cases are examples of how random checkpointing and access patterns can affect back-in-time performance. In the "high cluster" case, there is an average of 5 versions in a segment. In the "low cluster" case, there is an average of 2 versions in a segment. The cliffs in these curves indicate the locations of checkpoints, since the access time for a checkpointed version drops to the best-case performance. As the level of clustering increases, the slope of the curve decreases, since multiple journal entries are read together in a single segment. Each curve is the average of 5 runs, and all variances are under 1ms.

350ms. The graph examines four different scenarios: best-case behavior, worst-case behavior, and two potential cases (one involving low clustering and one involving high clustering).

The best-case back-in-time performance is the situation where a checkpoint is kept for each version of the file, and so any version can be immediately accessed with no journal roll-back. This is the performance of a conventional versioning system. The worst-case performance is the situation where no checkpoints are kept, and every version must be created through journal roll-back. In addition there is no clustering, since each journal entry is in a separate segment on the disk. This results in a separate disk access to read each entry. In the high clustering case, changes are made in bursts, causing journal entries to be clustered together into segments. This reduces the slope of the back-in-time performance curve. In the low clustering case, journal entries are spread more evenly across the segments, giving a higher slope. In both the low and high clustering cases, the points where the performance drops back to the best-case are the locations of checkpoints.

Using this knowledge of back-in-time performance, a system can perform a few optimizations. By tracking checkpoint frequency and journal entry clustering, CVFS can predict the back-in-time performance of a file while it is being written. With this information, CVFS bounds the performance of the back-in-time operations for a particular file by forcing a checkpoint whenever back-in-time performance is expected to be poor. For example, in Figure 4.14, the high-clustering case keeps checkpoints in such a way as to bound back-in-time performance to around 100ms at worst. In our S4 prototype, we bound the back-in-time performance to approximately 150ms. Another possibility is to keep checkpoints at the point at which one believes the user would wish to access the file. Using a heuristic such as in the Elephant FS [Santry99] to decide when to create file checkpoints might closely simulate the back-in-time performance of conventional versioning.

Multiversion B-trees: Figure 4.15 shows the average access time of a single entry from a directory given some fixed number of entries currently stored within the directory (notice the log scale of 13 the x-axis).

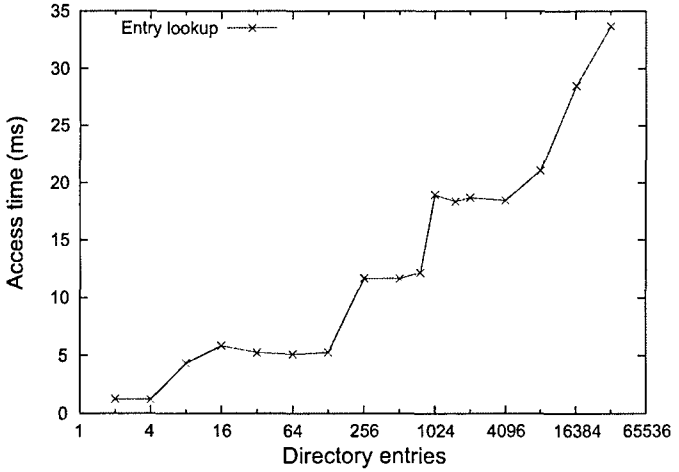


Figure 4.15: Directory entry performance. This figure shows the average time to access a single entry out of a directory given the total number of entries within the directory. History entries affect performance by increasing the effective number of entries in the directory. The larger the ratio of history entries to current entries, the more current version performance will suffer. This curve is the average of 15 runs and the variance for each point is under .2ms.

To see how a multiversion b-tree performs as compared to a standard b-tree, we must compare two different points on the graph. The point on the graph corresponding to the number of current entries in the directory represents the access time of a standard b-tree. The point on the graph corresponding to the combined number of current and history entries represents the access time of a multiversion b-tree. The difference between these values is the lookup performance lost by keeping the extra versions.

Using the traces gathered from our NFS server, we found that the average number of current entries in a directory is approximately 16. Given a detection window of one month, the number of history entries is less than 100 over 99% of the time, and between zero and five over 95% of the time. Since approximately 200 entries can fit into a block, there is generally no performance lost by keeping the history. This block-based performance explains the stepped nature of Figure 4.15.

4.10.4 Summary

Our results show that CVFS reduces the space utilization of versioned metadata by more than 80% without causing noticeable performance degradation to current version access. In addition, through intelligent checkpointing, it is possible to bound back-in-time performance to within a constant factor of conventional versioning systems.

4.11 Metadata Versioning in Non-Log-Structured Systems

Most file systems today use a layout scheme similar to that of BSD FFS rather than a log-structured layout. Such systems can be extended to support versioning relatively easily; Santry et al. [Santry99] describe how this was done for Elephant, including tracking of versions and block sharing among versions. For non-trivial pruning policies, such as those used in Elephant and CVFS, a cleaner is required to expire old file versions. In an FFS-like system, unlike in LFS, the cleaner does not necessarily have the additional task of coalescing free space.

Both journal-based metadata and multiversion b-trees can be used in a versioning FFS-like file system in much the same way as in CVFS. Replacing conventional directories with multiversion b-trees is a self-contained change, and the characteristics should be as described in the paper. Replacing replicated metadata versions with journal-based metadata requires effort similar to adding write-ahead logging support. Experience with adding such support [Hagmann87, Seltzer00] suggests that relatively little effort is involved, little change to the on-disk structures is involved, and substantial benefits accrue. If such logging is already present, journal-based metadata can piggyback on it.

Given write-ahead logging support, journal-based meta-data requires three things. First, updates to the write-ahead log, which are the journal entries, must contain enough information to roll-back as well as roll-forward. Second, the journal entries must be kept until the cleaner removes them; they cannot be re-

moved via standard log checkpointing. This will increase the amount of space required for the log, but by much less than the space required to instead retain metadata replicas. Third, the metadata replica support must be retained for use in tracking metadata version checkpoints.

With a clean slate, we think an LFS-based design is superior if many versions are to be retained. Each version committed to disk requires multiple updates, and LFS coalesces those updates into a single disk write. LFS does come with cleaning and fragmentation issues, but researchers have developed sufficiently reasonable solutions to them to make the benefits outweigh the costs in many environments. FFS-type systems that employ copy-on-write versioning have similar fragmentation issues.

4.12 Discussion

This section overviews several important implications of self-securing storage.

Selective versioning: There are data that users would prefer not to have backed up at all. The common approach to this is to store them in directories known to be skipped by the backup system. Since one of the goals of S4 is to allow recovery of exploit tools, it does not support designating objects as non-versioned. A system may be configured with non-S4 partitions to support selective versioning. While this would provide a way to prevent versioning of temporary files and other non-critical data, it would also create a location where an intruder could temporarily store exploit tools without fear that they will be recovered.

Versioning vs. snapshots: Self-securing storage can be implemented with frequent copy-on-write snapshots [Hitz94, Howard88, Lee96] instead of versioning, so long as snapshots are kept for the full detection window. Although the audit log can still provide a record of what blocks are changed, snapshots often will not allow administrators to recover short-lived files (e.g., exploit tools) or intermediate versions (e.g., system log file updates). Also, legitimate changes are only guaranteed to survive malicious activity if they survive to the next snapshot time. Of course, the potential scope of such problems can be reduced by shrinking the time between snapshots. The comprehensive versioning promoted in here represent the natural end-point of such shrinking – every modification creates a new snapshot.

Versioning file systems vs. self-securing storage: Versioning file systems excel at providing users with a safety net for recovery from accidents. They maintain old file versions long after they would be reclaimed by the S4 system, but they provide little additional system security. This is because they rely on the host's OS for security and aggressively prune apparently insignificant versions. By combining self-securing storage with long-term landmark versioning [Santry99], recovery from users' accidents could be enhanced while also maintaining the benefits of intrusion survival.

Self-securing storage for databases: Most databases log all changes in order to protect internal consistency in the face of system crashes. Some institutions also retain these logs for long-term auditing purposes. All information needed to understand and recover from malicious behavior can be kept, in database-specific form, in these logs. Self-securing storage can increase the post-intrusion recoverability of database systems in two ways: (1) by preventing undetectable tampering with stored log records, and (2) by preventing undetectable changes to data that bypass the log. After an intrusion, self-securing storage allows a database system to verify its log's integrity and confirm that all changes are correctly reflected in the log – the database system can then safely use its log for subsequent recovery.

Client-side cache effects: In order to improve efficiency, most client systems use caches to minimize storage latencies. This is at odds with the desire to have storage devices audit users' accesses and capture exploit tools. Client-side read caches hide data dependency information that would otherwise be available to the drive in the form of reads followed quickly by writes. However, this information could be provided by client systems as (questionable) hints during writes. Write caches cause a more serious problem when files are created then quickly deleted, thus never being sent to the drive. This could cause difficulties with capturing exploit tools, since they may never be written to the drive. Although client cache effects may obscure some of the activity in the client system, data that are stored on a self-securing storage device are still completely protected.

Object-based vs. block-based storage: Implementing a self-securing storage device with a block interface adds several difficulties. Since objects are designed to contain one data item (file or directory), enforcing access control at this level is much more manageable than attempting to assign permissions on a per-block basis. In addition, maintaining versions of objects as a whole, rather than having to collect and correlate individual blocks, simplifies recovery tools and internal reorganization mechanisms.

Multi-device coordination: Multi-device coordination is necessary for operations such as striping data or implementing RAID across multiple self-securing disks or file servers. In addition to the coordination necessary to ensure that multiple copies of data are synchronized, recovery operations must also coordinate old versions. On the other hand, clusters of self-securing storage devices could maintain a single history pool and balance the load of versioning objects. Note that a self-securing storage device containing several disks (e.g., a self-securing disk array) does not have these issues. Additionally, it has the ability to keep old versions and current data on separate disks.

4.13 Conclusions

Self-securing storage ensures data and audit log survival in the presence of successful intrusions and even compromised host operating systems. Experiments with the S4 prototype show that self-securing storage devices can achieve performance that is comparable to existing storage appliances. In addition, analysis of recent workload studies suggest that complete version histories can be kept for several weeks on state-of-the-art disk drives.

Our work on versioning shows that journal-based metadata and multiversion b-trees address the space-inefficiency of conventional versioning. Integrating them into the CVFS file system has nearly doubled the detection window that can be provided with a given storage capacity. Further, current version performance is affected minimally, and back-in-time performance can be bounded reasonably with checkpointing.

5 STORAGE-BASED INTRUSION DETECTION, DIAGNOSIS AND RECOVERY

5.1 Introduction

Many intrusion detection systems (IDSs) have been developed over the years [Axelsson98, Lunt88, Porras97], with most falling into one of two categories: network-based or host-based. Network IDSs (NIDS) are usually embedded in sniffers or firewalls, scanning traffic to, from, and within a network environment for attack signatures and suspicious traffic [Cheswick94, NFR02]. Host-based IDSs (HIDS) are fully or partially embedded within each host's OS. They examine local information (such as system calls [Forrest96]) for signs of intrusion or suspicious behavior. Many environments employ multiple IDSs, each watching activity from its own vantage point.

The storage system is another interesting vantage point for intrusion detection. Several common intruder actions [Denning99, p. 218; Scambray01, pp. 363–365] are quite visible at the storage interface. Examples include manipulating system utilities (e.g., to add backdoors or Trojan horses), tampering with audit log contents (e.g., to eliminate evidence), and resetting attributes (e.g., to hide changes). By design, a storage server sees all changes to persistent data, allowing it to transparently watch for suspicious changes and issue alerts about the corresponding client systems. Also, like a NIDS, a storage IDS must be compromise-independent of the host OS, meaning that it cannot be disabled by an intruder who only successfully gets past a host's OS-level protection.

This chapter motivates and describes storage-based intrusion detection, diagnosis and recovery. It presents several kinds of suspicious behavior that can be spotted by a storage IDS. Using sixteen "rootkits" and two worms as examples, we describe how fifteen of them would be exposed rapidly by our storage IDS. (The other three do not modify stored files.) Most of them are exposed by modifying system binaries, adding files to system directories, scrubbing the audit log, or using suspicious file names. Of the fifteen detected, three modify the kernel to hide their presence from host-based detection including FS integrity checkers like Tripwire [Kim94]. In general, compromises cannot hide their changes from the storage device if they wish to persist across reboots; to be re-installed upon reboot, the tools must manipulate stored files.

A storage IDS could be embedded in many kinds of storage systems. The extra processing power and memory space required should be feasible for file servers, disk array controllers, and perhaps augmented disk drives. Most detection rules will also require FS-level understanding of the stored data. Such understanding exists trivially for a file server, and may be explicitly provided to block-based storage devices. This understanding of a file system is analogous to the understanding of application protocols used by a NIDS [Paxson98], but with fewer varieties and structural changes over time.

As a concrete example with which to experiment, we have augmented an NFS server with a storage IDS that supports online, rule-based detection of suspicious modifications. This storage IDS supports the detection of four categories of suspicious activities. First, it can detect unexpected changes to important system files and binaries, using a rule-set very similar to Tripwire's. Second, it can detect patterns of changes like non-append modification (e.g., of system log files) and reversing of inode times. Third, it can detect specifically proscribed content changes to critical files (e.g., illegal shells inserted into /etc/passwd). Fourth, it can detect the appearance of specific file names (e.g., hidden "dot" files) or content (e.g., known viruses or attack tools). An administrative interface supplies the detection rules, which are checked during the processing of each NFS request. When a detection rule triggers, the server sends the administrator an alert containing the full pathname of the modified file, the violated rule, and the offending NFS operation. Experiments show that the runtime cost of such intrusion detection is minimal. Further analysis indicates that little memory capacity is needed for reasonable rulesets (e.g., only 152KB for an example containing 4730 rules).

5.2 Storage-based Intrusion Detection

Storage-based intrusion detection enables storage devices to examine the requests they service for suspicious client behavior. Although the world view that a storage server sees is incomplete, two features combine to make it a well-positioned platform for enhancing intrusion detection efforts. First, since storage devices are independent of host OSes, they can continue to look for intrusions after the initial compromise, whereas a host-based IDS can be disabled by the intruder. Second, since most computer systems rely heavily on persistent storage for their operation, many intruder actions will cause storage activity that can be captured and analyzed. This section expands on these two features and identifies limitations of storage-based intrusion detection.

5.2.1 Threat Model and Assumptions

Storage IDSs focus on the threat on of an attacker who has compromised a host system in a managed computing environment. By "compromised," we mean that the attacker subverted the host's software system, gaining the ability to run arbitrary software on the host with OS-level privileges. The compromise might have been achieved via technical means (e.g., exploiting buggy software or a loose policy) or non-technical means (e.g., social engineering or bribery). Once the compromise occurs, most administrators wish to detect the intrusion as quickly as possible and terminate it. Intruders, on the other hand, often wish to hide their presence and retain access to the machine.

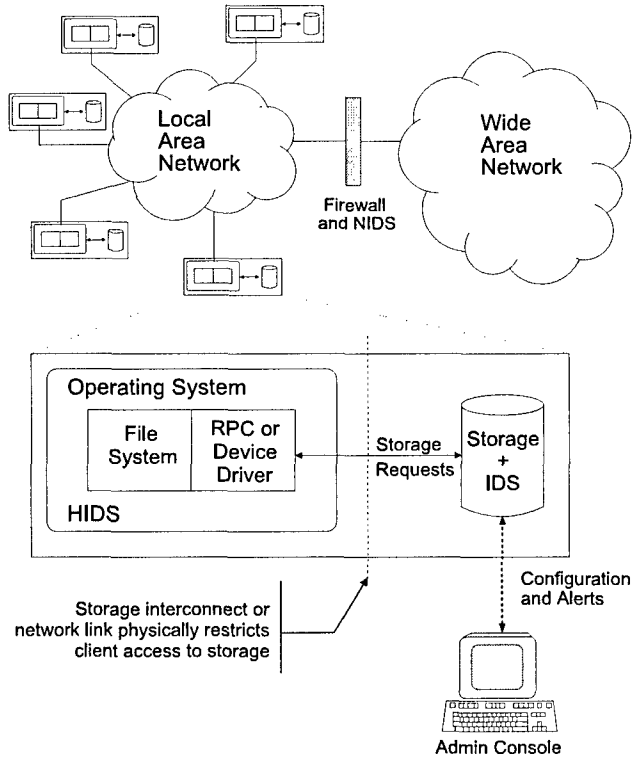


Figure 5.1: The compromise independence of a storage IDS. The storage interface provides a physical boundary behind which a storage server can observe the requests it is asked to service. Note that this same picture works for block protocols, such as SCSI or IDE/ATA, and distributed file system protocols, such as NFS or CIFS. Also note that storage IDSs do not replace existing IDSs, but simply offer an additional vantage point from which to detect intrusions.

Unfortunately, once an intruder compromises a machine, intrusion detection with conventional schemes becomes much more difficult. Host-based IDSs can be rendered ineffective by intruder software that disables them or feeds them misinformation, for which many tools exist. Network IDSs can continue to look for suspicious behavior, but are much less likely to find an already successful intruder—most NIDSs look for attacks and intrusion attempts rather than for system usage by an existing intruder [Ganger03a]. A storage IDS can help by offering a vantage point on a system component that is often manipulated in suspicious ways *after* the intruder compromises the system.

A key characteristic of the described threat model is that the attacker has software control over the host, but does not have physical access to its hardware. We are not specifically trying to address insider attacks, in which the intruder would also have physical access to the hardware and its storage components. Also, for the storage IDS to be effective, we assume that neither the storage device nor the admin console are compromised.

5.2.2 Compromise Independence

A storage IDS will continue watching for suspicious activity even when clients' OSes are compromised. It capitalizes on the fact that

storage devices (whether file servers, disk array controllers, or even IDE disks) run different software on separate hardware, as illustrated in Figure 5.1. This fact enables server-embedded security functionality that cannot be disabled by any software running on client systems (including the OS kernel). Further, storage devices often have fewer network interfaces (e.g., RPC+SNMP+HTTP or just SCSI) and no local users. Thus, compromising a storage server should be more difficult than compromising a client system. Of course, such servers have a limited view of system activity, so they cannot distinguish legitimate users from clever impostors. But, from behind the physical storage interface, a storage IDS can spot many common intruder actions and alert administrators.

Administrators must be able to communicate with the storage IDS, both to configure it and to receive alerts. This administrative channel must also be compromise-independent of client systems, meaning that no user (including root) and no software (including the OS kernel) on a client system can have administrative privileges for the storage IDS. Section 5.4 discusses deployment options for the administrative console, including physical consoles and cryptographic channels from a dedicated administrative system.

All of the warning signs discussed here could also be spotted from within a HIDS, but host-based IDSs do not enjoy the compromise independence of storage IDSs. A host-based IDS is vulnerable to being disabled or bypassed by intruders that compromise the OS kernel. Another interesting place for a storage IDS is the virtual disk module of a virtual machine monitor [Sugerman01]; such deployment would enjoy compromise independence from the OSes running in its virtual machines [Chen01].

5.2.3 Warning Signs for Storage IDSs

Successful intruders often modify stored data. For instance, they may overwrite system utilities to hide their presence, install Trojan horse daemons to allow for reentry, add users, modify startup scripts to reinstall kernel modifications upon reboot, remove evidence from the audit log, or store illicit materials. These modifications are visible to the storage system when they are made persistent. This section describes four categories of warning signs that a storage IDS can monitor: data and attribute modifications, update patterns, content integrity, and suspicious content.

5.2.3.1 Data/Attribute Modification

In managed computing environments, the simplest (and perhaps most effective) category of warning signs consists of data or meta-data changes to files that administrators expect to remain unchanged except during explicit upgrades. Examples of such files include system executables and scripts, configuration files, and system header files and libraries. Given the importance of such files and the infrequency of updates to them, any modification is a potential sign of intrusion. A storage IDS can detect all such modifications on-the-fly, before the storage device processes each request, and issue an alert immediately.

In current systems, modification detection is sometimes provided by a checksumming utility (e.g., Tripwire [Kim94]) that periodically compares the current storage state against a reference database stored elsewhere. Storage-based intrusion detection improves on this current approach in three ways: (1) it allows immediate detection of changes to watched files; (2) it can notice short-term changes, made and then undone, which would not be noticed by a checksumming utility if the changes occurred between two periodic checks; and (3) for local storage, it avoids trusting the host OS to perform the checks, which many rootkits disable or bypass.

5.2.3.2 Update Patterns

A second category of warning signs consists of suspicious access patterns, particularly updates. There are several concrete examples for which storage IDSs can be useful in watching. The clearest is client system audit logs; these audit logs are critical to both intrusion detection [Denning87] and diagnosis [Schneier99], leading many intruders to scrub evidence from them as a precaution. Any such manipulation will be obvious to a storage IDS that understands the well-defined update pattern of the specific audit log. For instance, audit log files are usually append-only, and they may be periodically "rotated." This rotation consists of renaming the current log file to an alternate name (e.g., `logfile` to `logfile.0`)

and creating a new "current" log file. Any deviation in the update pattern of the current log file or any modification of a previous log file is suspicious.

Another suspicious update pattern is timestamp reversal. Specifically, the data modification and attribute change times commonly kept for each file can be quite useful for post-intrusion diagnosis of which files were manipulated [Farmer00a]. By manipulating the times stored in inodes (e.g., setting them back to their original values), an intruder can inhibit such diagnosis. Of course, care must be taken with IDS rules, since some programs (e.g., `tar`) legitimately set these times to old values. One possibility would be to only set off an alert when the modification time is set back long after a file's creation. This would exclude `tar`-style activity but would catch an intruder trying to obfuscate a modified file. Of course, the intruder could now delete the file, create a new one, set the date back, and hide from the storage IDS—a more complex rule could catch this, but such escalation is the nature of intrusion detection.

Detection of storage denial-of-service (DoS) attacks also falls into the category of suspicious access patterns. For example, an attacker can disable specific services or entire systems by allocating all or most of the free space. A similar effect can be achieved by allocating inodes or other metadata structures. A storage IDS can watch for such exhaustion, which may be deliberate, accidental, or coincidental (e.g., a user just downloaded 10GB of multimedia files). When the system reaches predetermined thresholds of unallocated resources and allocation rate, warning the administrator is appropriate even in non-intrusion situations—attention is likely to be necessary soon. A storage IDS could similarly warn the administrator when storage activity exceeds a threshold for too long, which may be a DoS attack or just an indication that the server needs to be upgraded.

Although specific rules can spot expected intruder actions, more general rules may allow larger classes of suspicious activity to be noticed. For example, some attribute modifications, like enabling "set UID" bits or reducing the permissions needed for access, may indicate foul play. Additionally, many applications access storage in a regular manner. As two examples: word processors often use temporary and backup files in specific ways, and UNIX password management involves a pair of inter-related files (`/etc/passwd` and `/etc/shadow`). The corresponding access patterns seen at the storage device will be a reflection of the application's requests. This presents an opportunity for anomaly detection based on how a given file is normally accessed. This could be done in a manner similar to learning common patterns of system calls [Forrest96] or starting with rules regarding the expected behavior of individual applications [Ko97]. Deviation from the expected pattern could indicate an intruder attempting to subvert the normal method of accessing a given file. Of course, the downside is an increase (likely substantial) in the number of false alarms. Our focus to date has been on explicit detection rules, but anomaly detection within storage access patterns is an interesting topic for future research.

5.2.3.3 Content Integrity

A third category of warning signs consists of changes that violate internal consistency rules of specific files. This category builds on the previous examples by understanding the application-specific semantics of particularly important stored data. Of course, to verify content integrity, the device must understand the format of a file. Further, while simple formats may be verified in the context of the write operation, file formats may be arbitrarily complex and verification may require access to additional data blocks (other than those currently being written). This creates a performance vs. security trade-off made by deciding which files to verify and how often to verify them. In practice, there are likely to be few critical files for which content integrity verification is utilized.

As a concrete example, consider a UNIX system password file (`/etc/passwd`), which consists of a set of well-defined records. Records are delimited by a line-break, and each record consists of seven colon-separated fields. Further, each of the fields has a specific meaning, some of which are expected to conform to rules of practice. For example, the seventh field specifies the "shell" program to be launched when a user logs in, and (in Linux) the file `/etc/shells` lists the legal options. During the "Capture the Flag" information warfare game at the 2002 DEF CON conference [Lemos02], one tactic used was to

change the root shell on compromised systems to `/sbin/halt`; once a targeted system's administrator noted the intrusion and attempted to become root on the machine (the common initial reaction), considerable down-time and administrative effort was needed to restore the system to operation. A storage IDS can monitor changes to `/etc/passwd` and verify that they conform to a set of basic integrity rules: 7-field records, non-empty password field, legal default shell, legal home directory, non-overlapping user IDs, etc. The attack described above, among others, could be caught immediately.

5.2.3.4 Suspicious Content

A fourth category of warning signs is the appearance of suspicious content. The most obvious suspicious content is a known virus or rootkit, detectable via its signature. Several high-end storage servers (e.g., from EMC [EMC-McAfee02] and Network Appliance [Phillips06]) now include support for internal virus scanning. By executing the scans within the storage server, viruses cannot disable the scanners even after infecting clients.

Two other examples of suspicious content are large numbers of "hidden" files or empty files. Hidden files have names that are not displayed by normal directory listing interfaces [Denning99, p. 217], and their use may indicate that an intruder is using the system as a storage repository, perhaps for illicit or pirated content. A large number of empty files or directories may indicate an attempt to exploit a race condition [Bishop96, Purczynski02] by inducing a time-consuming directory listing, search, or removal.

5.2.4 Limitations, Costs and Weaknesses

Although storage-based intrusion detection contributes to security efforts, of course it is not a silver bullet.

Like any IDS, a storage IDS will produce some false positives. With very specific rules, such as "watch these 100 files for any modification," false positives should be infrequent; they will occur only when there are legitimate changes to a watched file, which should be easily verified if updates involve a careful procedure. The issue of false alarms grows progressively more problematic as the rules get less exact (e.g., the time reversal or resource exhaustion examples). The far end of the spectrum from specific rules is general anomaly detection.

Also like any IDS, a storage IDS will fail to spot some intrusions. Fundamentally, a storage IDS cannot notice intrusions whose actions do not cause odd storage behavior. For example, three of the eighteen intrusion tools examined in the next section modify the OS but change no files. Also, an intruder may manipulate storage in unwatched ways. Using network-based and host-based IDSs together with a storage IDS can increase the odds of spotting various forms of intrusion.

Intrusion detection, as an aspect of information warfare, is by nature a "game" of escalation. As soon as one side takes away an avenue of attack, the other starts looking for the next. Since storage-based intrusion detection easily sees several common intruder activities, crafty intruders will change tactics. For example, an intruder can make any number of changes to the host's memory, so long as those modifications do not propagate to storage. A reboot, however, will reset the system and remove the intrusion, which argues for proactive restart [Castro00, Huang96, Vaidyanathan02]. To counter this, attackers must have their changes re-established automatically after a reboot, such as by manipulating the various boot-time (e.g., `rc.local` in UNIX-like systems) or periodic (e.g., `cron` in UNIX-like systems) programs. Doing so exposes them to the storage IDS, creating a traditional intrusion detection game of cat and mouse.

As a practical consideration, storage IDSs embedded within individual components of decentralized storage systems are unlikely to be effective. For example, a disk array controller is a fine place for storage-based intrusion detection, but individual disks behind software striping are not. Each of the disks has only part of the file system's state, making it difficult to check non-trivial rules without adding new inter-device communication paths.

Finally, storage-based intrusion detection is not free. Checking rules comes with some cost in processing and memory resources, and more rules require more resources. In configuring a storage IDS, one must balance detection efforts with performance costs for the particular operating environment.

5.3 Case Studies

This section explores how well a storage IDS might fare in the face of actual compromises. To do so, we examined eighteen intrusion tools (Table 5.1) designed to be run on compromised systems. All were downloaded from public websites, most of them from Packet Storm [PSS03].

Most of the actions taken by these tools fall into two categories. Actions in the first category involve hiding evidence of the intrusion and the rootkit's activity. The second provides a mechanism for reentry into a system. Twelve of the tools operate by running various binaries on the host system and overwriting existing binaries to continue gaining control. The other six insert code into the operating system kernel.

For the analysis here, we focus on a subset of the rules supported by our prototype storage-based IDS described in Section 5.5. Specifically, we include the file/directory modification (Tripwire-like) rules, the append-only logfile rule, and the hidden filename rules. We do not consider any "suspicious content" rules, which may or may not catch a rootkit depending on whether its particular signature is known.[8] In these 18 toolkits, we did not find any instances of resource exhaustion attacks or of reverting inode times.

5.3.1 Detection Results

Of the eighteen toolkits tested, storage IDS rules would immediately detect fifteen based on their storage modifications. Most would trigger numerous alerts, highlighting their presence. The other three make no changes to persistent storage. However, they are removed if the system reboots; all three modify the kernel, but would have to be combined with system file changes to be re-inserted upon reboot.

Non-append changes to the system audit log. Seven of the eighteen toolkits scrub evidence of system compromise from the audit log. All of them do so by selectively overwriting entries related to their intrusion into the system, rather than by truncating the logfile entirely. All cause alerts to be generated in our prototype.

System file modification. Fifteen of the eighteen toolkits modify a number of watched system files (ranging from 1 to 20). Each such modification generates an alert. Although three of the rootkits replace the files with binaries that match the size and CRC checksum of the previous files, they do not foil cryptographically-strong hashes. Thus, Tripwire-like systems would be able to catch them as well, though the evasion mechanism described in Section 5.3.2 defeats Tripwire.

Many of the files modified are common utilities for system administration, found in `/bin`, `/sbin`, and `/usr/bin` on a UNIX machine. They are modified to hide the presence and activity of the intruder. Common changes include modifying `ps` to not show an intruder's processes, `ls` to not show an intruder's files, and `netstat` to not show an intruder's open network ports and connections. Similar modifications are often made to `grep`, `find`, `du`, and `pstree`.

The other common reason for modifying system binaries is to create backdoors for system reentry. Most commonly, the target is `telnetd` or `sshd`, although one rootkit added a backdoored PAMmodule [Samar95] as well. Methods for using the backdoor vary and do not impact our analysis.

Hidden file or directory names. Twelve of the rootkits make a hard-coded effort to hide their non-executable and working files (i.e., the files that are not replacing existing files). Ten of the kits use directories starting in a '.' to hide from default ls listings. Three of these generate alerts by trying to make a hidden directory look like the reserved '.' or '..' directories by appending one or more spaces ('. ' or '.. '). This also makes the path harder to type if a system administrator does not know the number of spaces.

[8] An interesting note is that rootkit developers reuse code: four of the rootkits use the same audit log scrubbing program (`sauber`), and another three use a different program (`zap2`).

5.3.2 Kernel-inserted Evasion Techniques

Six of the eighteen toolkits modified the running operating system kernel. Five of these six "kernel rootkits" include loadable kernel modules (LKMs), and the other inserts itself directly into kernel memory by use of the `/dev/kmem` interface. Most of the kernel modifications allow intruders to hide as well as reenter the system, similarly to the file modifications described above. Especially interesting for this analysis is the use of `exec()` redirection by four of the kernel rootkits. With such redirection, the `exec()` system call uses a replacement version of a targeted program, while other system calls return information about or data from the original. As a result, any tool relying on the accuracy of system calls to check file integrity, such as Tripwire, will be fooled.

Name	Description	Syscall redir.	Log scrub	Hidden dirs	Watched files	Total alerts
Ramen	Linux worm			X	2	3
lion	Linux worm				10	10
FK 0.4	Linux LKM rootkit and trojan ssh	X			1	1
Taskigt	Linux LKM rootkit				1	1
SK 1.3a	Linux kernel rootkit via `/dev/kmem`	X				-
Darkside 0.2.3	FreeBSD LKM rootkit	X				-
Knark 0.59	Linux LKM rootkit	X		X	1	2
Adore	Linux LKM rootkit	X				-
lrk5	User level rootkit from source		X	X	20	22
Sun rootkit	SunOS rootkit with trojan rlogin				1	1
FreeBSD Rootkit 2	User level FreeBSD rootkit		X	X	15	17
t0rn	Linux user level rootkit		X	X	20	22
Advanced Rootkit	Linux user level rootkit			X	10	11
ASMD	Rootkit w/SUID binary trojan			X	1	2
Dica	Linux user level rootkit		X	X	9	11
Flea	Linux user level rootkit		X	X	20	22
Ohara	Rootkit w/PAM trojan		X	X	4	6
TK 6.66	Linux user level rootkit		X	X	10	12

Table 5.1: Visible actions of several intruder toolkits. For each of the tools, the table shows which of the following actions are performed: redirecting system calls, scrubbing the system log files, and creating hidden directories. It also shows how many of the files watched by our rule set are modified by a given tool. The final column shows the total number of alerts generated by a given tool.

All of these rootkits are detected using our storage IDS rules—they all put their replacement programs in the originals' directories (which are watched), and four of the six actually move the original file to a new name and store their replacement file with the original name (which also triggers an alert). However, future rootkits could be modified to be less obvious to a storage IDS. Specifically, the original files could be left untouched and replacement files could be stored someplace not watched by the storage IDS, such as a random user directory—neither would generate an alert. With this approach, file modification can be completely hidden from a storage IDS unless the rootkit wants to reinstall the kernel modification after a reboot. To accomplish this, some original files would need to be changed, which forces intruders to make an interesting choice: hide from the storage IDS or persist beyond the next reboot.

5.4 Design of a Storage IDS

To be useful in practice, a storage IDS must simultaneously achieve several goals. It must support a useful set of detection rules, while also being easy for human administrators to understand and configure. It must be efficient, minimizing both added delay and added resource requirements; some user communities still accept security measures only when they are "free." Additionally, it should be invisible to users at least until an intrusion detection rule is matched.

This section describes four aspects of storage IDS design: specifying detection rules, administering a storage IDS securely, checking detection rules, and responding to suspicious activity.

5.4.1 Specifying Detection Rules

Specifying rules for an IDS is a tedious, error prone activity. The tools an administrator uses to write and manipulate those rules should be as simple and straightforward as possible. Each of the four categories of suspicious activity presented earlier will likely need a unique format for rule specification.

The rule format used by Tripwire seems to work well for specifying rules concerned with data and attribute modification. This format allows an administrator to specify the pathname of a file and a list of properties that should be monitored for that file. The set of watchable properties are codified, and they include most file attributes. This rule language works well, because it allows the administrator to manipulate a well understood representation (pathnames and files), and the list of attributes that can be watched is small and well-defined.

The methods used by virus scanners work well for configuring an IDS to look for suspicious content. Rules can be specified as signatures that are compared against files' contents. Similarly, filename expression grammars (like those provided in scripting languages) could be used to describe suspicious filenames.

Less guidance exists for the other two categories of warning signs: update patterns and content integrity. We do not currently know how to specify general rules for these categories. Our approach has been to fall back on Tripwire-style rules; we hard-code checking functions (e.g., for non-append update or a particular content integrity violation) and then allow an administrator to specify on which files they should be checked (or that they should be checked for every file). More general approaches to specifying detection rules for these categories of warning signs are left for future work.

5.4.2 Secure Administration

The security administrator must have a secure interface to the storage IDS. This interface is needed for the administrator to configure detection rules and to receive alerts. The interface must prevent client systems from forging or blocking administrative requests, since this could allow a crafty intruder to sneak around the IDS by disarming it. At a minimum, it must be tamper-evident. Otherwise, intruders could stop rule updates or prevent alerts from reaching the administrator. To maintain compromise independence, it must be the case that obtaining "superuser" or even kernel privileges on a client system is insufficient to gain administrative access to the storage device.

Two promising architectures exist for such administration: one based on physical access and one based on cryptography. For environments where the administrator has physical access to the device, a local administration terminal that allows the administrator to set detection rules and receive the corresponding alert messages satisfies the above goals.

In environments where physical access to the device is not practical, cryptography can be used to secure communications. In this scenario, the storage device acts as an endpoint for a cryptographic channel to the administrative system. The device must maintain keys and perform the necessary cryptographic functions to detect modified messages, lost messages, and blocked channels. Architectures for such trust models in storage systems exist [Gobioff99]. This type of infrastructure is already common for administration of other network-attached security components, such as firewalls or network intrusion detection systems. For direct-attached storage devices, cryptographic channels can be used to tunnel administrative requests and alerts through the OS of the host system, as illustrated in Figure 5.2. Such tunneling simply treats the host OS as an untrusted network component.

For small numbers of dedicated servers in a machine room, either approach is feasible. For large numbers of storage devices or components operating in physically insecure environments, cryptography is the only viable solution.

5.4.3 Checking the Detection Rules

Checking detection rules can be non-trivial, because rules generally apply to full pathnames rather than inodes. Additional complications arise because rules can watch for files that do not yet exist. For simple operations that act on individual files (e.g., READ and WRITE), rule verification is localized. The device need only check that the rules pertaining to that specific file are not violated (usually a simple flag comparison, sometimes a content check). For operations that affect the file system's namespace, verification is more complicated. For example, a rename of a directory tree may impact a large number of individual files, any of which could have IDS rules that must be checked. Renaming a directory requires examining all files and directories that are children of the one being renamed.

In the case of rules pertaining to files that do not currently exist, this list of rules must be consulted when operations change the namespace. For example, the administrator may want to watch for the existence of a file named `/a/b/c` even if the directory named `/a` does not yet exist. However, a single file system operation (e.g., `mv /z /a`) could cause the watched file to suddenly exist, given the appropriate structure for `z`'s directory tree.

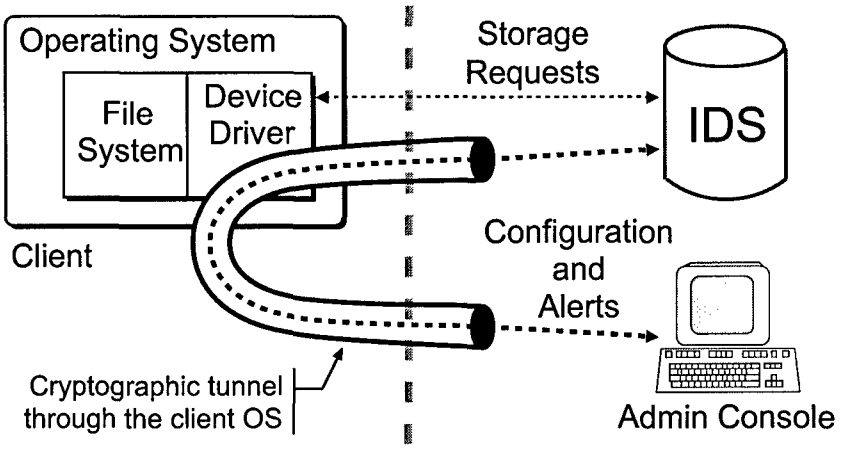


Figure 5.2: Tunneling administrative commands through client systems. For storage devices attached directly to client systems, a cryptographic tunnel can allow the administrator to securely manage a storage IDS. This tunnel uses the untrusted client OS to transport administrative commands and alerts.

5.4.4 Responding to Rule Violations

Since a detected "intruder action" may actually be legitimate user activity (i.e., a false alarm), our default response is simply to send an alert to the administrative system or the designated alert log file. The alert message should contain such information as the

file(s) involved, the time of the event, the action being performed, the action's attributes (e.g., the data written into the file), and the client's identity. Note that, if the rules are set properly, most false positives should be caused by legitimate updates (e.g., upgrades) from an administrator. With the right information in alerts, an administrative system that also coordinates legitimate upgrades could correlate the generated alert (which can include the new content) with the in-progress upgrade; if this were done, it could prevent the false alarm from reaching the human administrator while simultaneously verifying that the upgrade went through to persistent storage correctly.

There are more active responses that a storage IDS could trigger upon detecting suspicious activity. When choosing a response policy, of course, the administrator must weigh the benefits of an active response against the inconvenience and potential damage caused by false alarms.

One reasonable active response is to slow down the suspected intruder's storage accesses. For example, a storage device could wait until the alert is acknowledged before completing the suspicious request. It could also artificially increase request latencies for a client or user that is suspected of foul play. Doing so would provide increased time for a more thorough response, and, while it will cause some annoyance in false alarm situations, it is unlikely to cause damage. The device could even deny a request entirely if it violates one of the rules, although this response to a false alarm could cause damage and/or application failure. For some rules, like append-only audit logs, such access control may be desirable.

Liu, et al. proposed a more radical response to detected intrusions: isolating intruders, via versioning, at the file system level [Liu00]. To do so, the file system forks the version trees to sandbox suspicious users until the administrator verifies the legitimacy of their actions. Unfortunately, such forking is likely to interfere with system operation, unless the intrusion detection mechanism yields no false alarms. Specifically, since suspected users modify different versions of files from regular users, the system faces a difficult reintegration [Kumar98, Terry95] problem, should the updates be judged legitimate. Still, it is interesting to consider embedding this approach, together with a storage IDS, into storage systems for particularly sensitive environments.

A less intrusive storage-embedded response is to start versioning all data and auditing all storage requests when an intrusion is detected. Doing so provides the administrator with significant information for post-intrusion diagnosis and recovery. Of course, some intrusion-related information will likely be lost unless the intrusion is detected immediately, which is why Strunk et al. [Strunk00] argue for always doing these things (just in case). Still, IDS-triggered employment of this functionality may be a useful trade-off point.

5.5 Storage-based Intrusion Detection in an NFS Server

To explore the concepts and feasibility of storage-based intrusion detection, we implemented a storage IDS in an NFS server. Unmodified client systems access the server using the standard NFS version 2 protocols [Sun89][9], while storage-based intrusion detection occurs transparently. This section describes how the prototype storage IDS handles detection rule specification, the structures and algorithms for checking rules, and alert generation.

The base NFS server is called S4, and its implementation is described and evaluated elsewhere [Strunk00]. It internally performs file versioning and request auditing, using a log-structured file system [Rosenblum92], but these features are not relevant here. For our purposes, it is a convenient NFS file server with performance comparable to the Linux and FreeBSD NFS servers. Secure administration is performed via the server's console, using the physical access control approach.

5.5.1 Specifying Detection Rules

Our prototype storage IDS is capable of watching for a variety of data and metadata changes to files. The administrator specifies a list of Tripwire-style rules to configure the detection system. Each administrator-

[9] The use of the NFSv2 protocol is an artifact of the server implementation the IDS is built into, but makes no difference in the areas we care about.

supplied rule is of the form: {*pathname, attribute-list*}—designating which attributes to monitor for a particular file. The list of attributes that can be watched is shown in Table 5.2. In addition to standard Tripwire rules, we have added two additional functions that can be specified on a per-file basis. The first watches for non-append changes, as described earlier; any truncation or write anywhere but at the previous end of a file will generate an alert. The second checks a file's integrity against the password file integrity rule discussed earlier. After every write, the file must conform to the rigid structure of a password file (7 colons per line), and all of the shells must be contained in the "acceptable" list.

In addition to per-file rules, an administrator can choose to enable any of three system-wide rules: one that matches on any operation that rolls-back a file's modification time, one that matches on any operation that creates a "hidden" directory (e.g., a directory name beginning with '.' and having spaces in it), and one that looks for known (currently hard-coded) intrusion tools by their sizes and SHA-1 digests. Although the system currently checks the digests on every file update, periodic scanning of the system would likely be more practical. These rules apply to all parts of the directory hierarchy and are specified as simply ON or OFF.

Metadata	
inode modification time	data modification time
access time	file permissions
link count	device number
file owner	inode number
file type	file owner group
file size	
Data	
any modification	append only
password structure	

Table 5.2: Attribute list. Rules can be established to watch these attributes in real-time on a file-by-file basis.

Rules are communicated to the server through the use of an administrative RPC. This RPC interface has two commands (see Table 5.3). The `setRule()` RPC gives the IDS two values: the path of the file to be watched, and a set of flags describing the specific rules for that file. Rules are removed through the same mechanism, specifying the path and an empty rule set.

5.5.2 Checking the Detection Rules

This section describes the core of the storage IDS. It discusses how rules are stored and subsequently checked during operation.

5.5.2.1 Data Structures

Three new structures allow the storage IDS to efficiently support the detection rules: the reverse lookup table, the inode watch flags, and the non-existent names table.

Reverse lookup table: The reverse lookup table serves two functions. First, it serves as a list of rules that the server is currently enforcing. Second, it maps an inode number to a pathname. The alert generation mechanism uses the latter to provide the administrator with file names instead of inode numbers, without resorting to a brute-force search of the namespace.

The reverse lookup table is populated via the `setRule()` RPC. Each rule's full pathname is broken into its component names, which are stored in distinct rows of the table. For each component, the table records four fields: *inode-number*, *directory-inode-number*, *name*, and *rules*. Indexed by *inode-number*, an entry

contains the *name* within a parent directory (identified by its *directory-inode-number*). The *rules* associated with this *name* are a bitmask of the attributes and patterns to watch. Since a particular inode number can have more than one name, multiple entries for each inode may exist. A given inode number can be translated to a full pathname by looking up its lowest-level name and recursively looking up the name of the corresponding directory inode number. The search ends with the known inode number of the root directory. All names for an inode can be found by following all paths given by the lookup of the inode number.

Command	Purpose	Direction
setRule(path, rules)	Changes the watched characteristics of a file. This command is used to both set and delete rules.	admin⇒server
listRules()	Retrieves the server's rule table as a list of {`pathname`, `rules`} records.	admin⇒server
alert(path, rules, operation)	Delivers a warning of a rule violation to the administrator.	server⇒admin

Table 5.3: Administrative commands for our storage IDS. This table lists the small set of administrative commands needed for an administrative console to configure and manage the storage IDS. The first two are sent by the console, and the third is sent by the storage IDS. The pathname refers to a file relative to the root of an exported file system. The *rules* are a description of what to check for, which can be any of the changes described in Table 5.2. The *operation* is the NFS operation that caused the rule violation.

Inode `watchflags` field: During the `setRule()` RPC, in addition to populating the reverse lookup table, a rule mask of 16 bits is computed and stored in the `watchflags` field of the watched file's inode. Since multiple pathnames may refer to the same inode, there may be more than one rule for a given file, and the mask contains the union. The inode `watchflags` field is a performance enhancement designed to co-locate the rules governing a file with that file's metadata. This field is not necessary for correctness since the pertinent data could be read from the reverse lookup table. However, it allows efficient verification of detection rules during the processing of an NFS request. Since the inode is read as part of any file access, most rule checking becomes a simple mask comparison.

Non-existent names table: The non-existent names table lists rules for pathnames that do not currently exist. Each entry in the table is associated with the deepest-level (existing) directory within the pathname of the original rule. Each entry contains three fields: *directory-inode-number*, *remaining-path*, and *rules*. Indexed by *directory-inode-number*, an entry specifies the *remaining-path*. When a file or directory is created or removed, the non-existent names table is consulted and updated, if necessary. For example, upon creation of a file for which a detection rule exists, the *rules* are checked and inserted in the `watchflags` field of the inode. Together, the reverse lookup table and the nonexistent names table contain the entire set of IDS rules in effect.

5.5.2.2 Checking Rules During NFS Operations

We now describe the flow of rule checking, much of which is diagrammed in Figure 5.3, in two parts: changes to individual files and changes to the namespace.

Checking rules on individual file operations: For each NFS operation that affects only a single file, a mask of rules that might be violated is computed. This mask is compared, bitwise, to the corresponding `watchflags` field in the file's inode. For most of the rules, this comparison quickly determines if any alerts should be triggered. If the "password file" or "append only" flags are set, the corresponding verification function executes to determine if the rule is violated.

Checking rules on namespace operations: Namespace operations can cause watched pathnames to appear or disappear, which will usually trigger an alert. For operations that create watched pathnames, the storage IDS moves rules from the non-existent names table to the reverse lookup table. Conversely, operations that delete watched pathnames cause rules to move between tables in the opposite direction.

When a name is created (via CREATE, MKDIR, LINK, or SYMLINK) the non-existent names table is checked. If there are rules for the new file, they are checked and placed in the `watchflags` field of the new inode. In addition, the corresponding rule is removed from the non-existent names table and is added to the reverse lookup table. During a MKDIR, any entries in the non-existent names table that include the new directory as the next step in their remaining path are replaced; the new entries are indexed by the new directory's inode number and its name is removed from the remaining path.

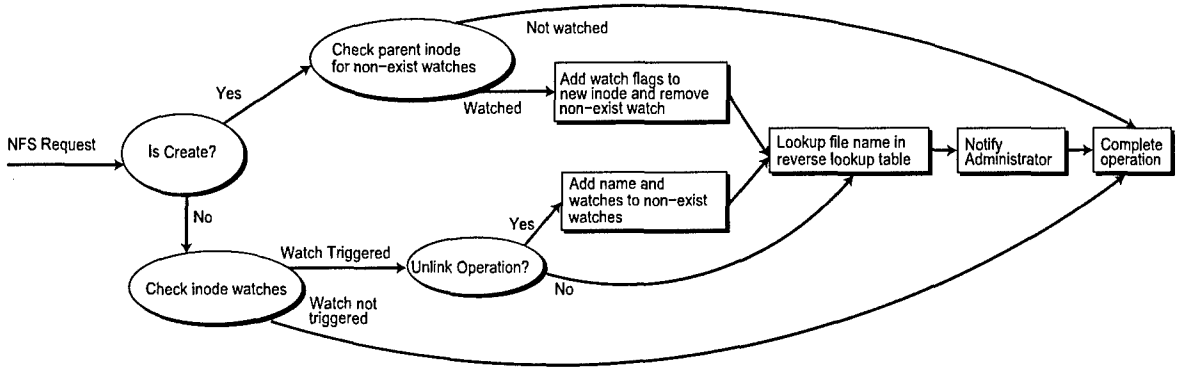


Figure 5.3: Flowchart of our storage IDS. Few structures and decision points are needed. In the common case (no rules for the file), only one inode's `watchflags` field is checked. The picture does not show RENAME operations here due to their complexity.

When a name is removed (via UNLINK or RMDIR), the `watchflags` field of the corresponding inode is checked for rules. Most such rules will trigger an alert, and an entry for them is also added to the non-existent names table. For RMDIR, the reverses of the actions for MKDIR are necessary. Any non-existent table entries parented on the removed directory must be modified. The removed directory's name is added to the beginning of each remaining path, and the directory inode number in the table is modified to be the directory's parent.

By far, the most complex namespace operation is a RENAME. For a RENAME of an individual file, modifying the rules is the same as a CREATE of the new name and a REMOVE of the old. When a directory is renamed, its subtrees must be recursively checked for watched files. If any are found, and once appropriate alerts are generated, their rules and pathname up to the parent of the renamed directory are stored in the non-existent names table, and the `watchflags` field of the inode is cleared. Then, the nonexistent names table must be checked (again recursively) for any rules that map into the directory's new name and its children; such rules are checked, added to the inode's `watchflags` field, and updated as for name creation.

5.5.3 Generating Alerts

Alerts are generated and sent immediately when a detection rule is triggered. The alert consists of the original detection rule (pathname and attributes watched), the specific attributes that were affected, and the RPC operation that triggered the rule. To get the original rule information, the reverse lookup table is consulted. If a single RPC operation triggers multiple rules, one alert is sent for each.

5.5.4 Storage IDS Rules in a NIDS

Because NFS traffic goes over a traditional network, the detection rules described for our prototype storage IDS could be implemented in a NIDS. However, this would involve several new costs. First, it would require the NIDS to watch the LAN links that carry NFS activity. These links are usually higher band-

width than the Internet uplinks on which most NIDSs are used.[10] Second, it would require that the NIDS replicate a substantial amount of work already performed by the NFS server, increasing the CPU requirements relative to an in-server storage IDS. Third, the NIDS would have to replicate and hold substantial amounts of state (e.g. mappings of file handles to their corresponding files). Our experiences checking rules against NFS traces indicate that this state grows rapidly because the NFS protocol does not expose to the network (or the server) when such state can be removed. Even simple attribute updates cannot be checked without caching the old values of the attributes, otherwise the NIDS could not distinguish modified attributes from reapplied values. Fourth, rules cannot always be checked by looking only at the current command. The NIDS may need to read file data and attributes to deal with namespace operations, content integrity checks, and update pattern rules. In addition to the performance penalty, this requires giving the NIDS read permission for all NFS files and directories.

Given all of these issues, we believe that embedding storage IDS checks directly into the storage component is more appropriate.

5.6 Detection Evaluation

This section evaluates the costs of our storage IDS in terms of performance impact and memory required—both costs are minimal.

5.6.1 Experimental Setup

All experiments use the S4 NFS server, with and without the new support for storage-based intrusion detection. The client system is a dual 1 GHz Pentium III with 128MB RAM and a 3Com 3C905B 100 Mbps network adapter. The server is a dual 700MHz Pentium III with 512MB RAM, a 9GB 10,000RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro 100Mb network adapter. The client and server are on the same 100Mb network switch. The operating system on all machines is Red Hat Linux 6.2 with Linux kernel version 2.2.14.

SSH-build was constructed as a replacement for the Andrew file system benchmark [Howard88, Seltzer00]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v. 1.2.27 (approximately 1MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [Katcher97]. It creates a large number of small randomly-sized files (between 512B and 16 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 100,000 transactions on 20,000 files, and the biases for transaction types are equal.

5.6.2 Performance Impact

The storage IDS checks a file's rules before any operation that could possibly trigger an alert. This includes READ operations, since they may change a file's last access time. Additionally, namespace-modifying operations require further checks and possible updates of the nonexistent names table. To understand the performance consequences of the storage IDS design, we ran PostMark and SSH-Build tests. Since our main concern is avoiding a performance loss in the case where no rule is violated, we ran these benchmarks with no relevant rules set. As long as no rules match, the results are similar with 0 rules,

[10] Tapping a NIDS into direct-attached storage interconnects, such as SCSI and FibreChannel, would be more difficult.

1000 rules on existing files, or 1000 rules on non-existing files. Table 5.4 shows that the performance impact of the storage IDS is minimal. The largest performance difference is for the configure and build phases of SSH-build, which involve large numbers of namespace operations.

Benchmark	Baseline	With IDS	Change
SSH untar	27.3 (0.02)	27.4 (0.02)	0.03%
SSH config.	42.6 (0.68)	43.2 (0.37)	1.3%
SSH build	85.9 (0.18)	86.8 (0.17)	1.0%
PostMark	4288 (11.9)	4290 (13.0)	0.04%

Table 5.4: Performance of macro benchmarks. All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 10 trials in seconds (with the standard deviation in parenthesis).

Benchmark	Baseline	With IDS	Change
Create	4.32	4.35	0.7%
Remove	4.50	4.65	3.3%
Mkdir	4.36	4.38	0.5%
Rmdir	4.52	4.59	1.5%
Rename file	3.81	3.91	2.6%
Rename dir	3.91	4.04	3.3%

Table 5.5: Performance of micro benchmarks. All benchmarks were run with and without the storage IDS functionality. Each number represents the average of 1000 trials in milliseconds.

Microbenchmarks on specific filesystem actions help explain the overheads. Table 5.5 shows results for the most expensive operations, which all affect the namespace. The performance differences are caused by redundancy in the implementation. The storage IDS code is kept separate from the NFS server internals, valuing modularity over performance. For example, name removal operations involve a redundant directory lookup and inode fetch (from cache) to locate the corresponding inode's `watchflags` field.

Rules take very little time to generate alerts. For example, a write to a file with a rule set takes 4.901 milliseconds if no alert is set off. If an alert is set off the time is 4.941 milliseconds. These represent the average over 1000 trials, and show a 0.8% overhead.

5.6.3 Space Efficiency

The storage IDS structures are stored on disk. To avoid extra disk accesses for most rule checking, though, it is important that they fit in memory.

Three structures are used to check a set of rules. First, each inode in the system has an additional two-byte field for the bitmask of the rules on that file. There is no cost for this, because the space in the inode was previously unused. Linux's ext2fs and BSD's FFS also have sufficient unused space to store such data without increasing their inode sizes. If space were not available, the reverse lookup table can be used instead, since it provides the same information. Second, for each pathname component of a rule, the reverse lookup table requires 20 + N bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and N+2bytes for a pathname component of length N. Third, the nonexistent names table contains one entry for every file being watched that does not currently exist. Each entry consumes 274 bytes: a 16-byte inode number, 2 bytes for the rule bitmask, and 256 bytes for the maximum pathname supported.

To examine a concrete example of how an administrator might use this system, we downloaded the open source version of Tripwire [Tripwire02]. Included with it is an example rule file for Linux, containing (after expanding directories to lists of files) 4730 rules. We examined a Red Hat Linux 6.1 [RedHat99] desktop machine to obtain an idea of the number of watched files that actually exist on the hard drive. Of the 4730watched files, 4689 existed on our example system. Using data structure sizes from above, reverse lookup entries for the watched files consume 141 KB. Entries in the non-existent name table for the remaining 41watched files consume 11 KB. In total, only 152KB are needed for the storage IDS.

5.6.4 False Positives

We have explored the false positive rate of storage-based intrusion detection in several ways.

To evaluate the file watch rules, two months of traces of all file system operations were gathered on a desktop machine in our group. We compared the files modified on this system with the watched file list from the open source version of Tripwire. This uncovered two distinct patterns where files were modified. Nightly, the user list (`/etc/passwd`) on the machine was overwritten by a central server. Most nights it does not change but the create and rename performed would have triggered an alert. Additionally, multiple binaries in the system were replaced over time by the administrative upgrade process. In only one case was a configuration file on the system changed by a local user.

For alert-triggering modifications arising from explicit administrative action, a storage IDS can provide an added benefit. If an administrator pre-informs the admin console of updated files before they are distributed to machines, the IDS can verify that desired updates happen correctly. Specifically, the admin console can read the new contents via the admin channel and verify that they are as intended. If so, the update is known to have succeeded, and the alert can be suppressed.

We have also performed two (much) smaller studies. First, we have evaluated our "hidden filename" rules by examining the entire filesystems of several desktops and servers—we found no uses of any of them, including the '.' or '..' followed by any number of spaces discussed above. Second, we evaluated our "inode time reversal" rules by examining lengthy traces of NFS activity from our environment and from two Harvard environments [Ellard03]—we found a sizable number of false positives, caused mainly by unpacking archives with utilities like `tar`. Combined with the lack of time reversal in any of the toolkits, use of this rule may be a bad idea.

5.7 Intrusion Detection in Workstation Disks

There are four main design issues for a storage-based intrusion detection system [Pennington03], which have been outlined in section 5.4. These include specifying access policies, securely administering the IDS, monitoring storage activity for policy violations, and responding to policy violations. This section discusses the aspects of these that specifically relate to the challenging environment of workstation disks.

5.7.1 Specifying Access Policies

For the sake of usability and correctness, there must be a simple and straightforward syntax for human administrators to state access policies to a disk-based IDS. Although a workstation disk operates using a block-based interface, it is imperative that the administrator be able to refer to higher-level file system objects contained on the disk when stating policies. As an example, an appropriate statement might be: *Warn me if anything changes in the directory /sbin.* In our experience, the Tripwire-like rules used by Pennington et al. to specify access policies for their server-based IDS [Pennington03] work well for specifying policies to a disk-based IDS.

A disk-based IDS must be capable of mapping such high-level statements into a set of violating interface actions. This set of violating actions may include writes, reads (e.g., of honeytokens [Spitzner03] such as `creditcards.txt`), and interface-specific commands (such as the FORMAT UNIT command for SCSI). One such mapping for the above "no-change" rule for `/sbin` could be: *Generate the alert "the file /sbin/fsck was modified" when a write to block #280 causes the 7 contents of block #280 to change.*

To accomplish this mapping, the IDS must be able to read and interpret the on-disk structures used by the file system. However, the passive nature of intrusion detection means it is not necessary for a disk-based IDS to be able to modify the file system, which simplifies implementation.

Workstation disks are frequently powered down. These mappings must be restored to the IDS (for example, from an on-disk copy) before regular operation commences. This requires a few megabytes of private disk space. This approach is no different from the other tables kept by disk firmware, such as for tracking defective sectors and predicting service times efficiently.

A disk-based IDS is capable of watching for writes to free space that do not correspond with updates to the file system's block allocation table. Storing object and data files in unallocated disk blocks is one method used by attackers to hide evidence of a successful intrusion [Grugg02]. Such hidden files are difficult to detect, but are accessible by processes (such as an IRC or FTP server) initiated by the attacker. Depending on the file system, watching free space may cause extra alerts to be generated for short-lived files which are deleted after the contents are written to disk but before the allocation table is updated.

5.7.2 Disk-based IDS Administration

For effective real-time operation of a disk-based IDS, there must be a secure method for communication between an administrator's computer and the IDS. This communication includes transmitting access policies to the IDS, receiving alerts generated by the IDS, and acknowledging the receipt of policies and alerts. Unlike in a server-based environment, however, a workstation disk will likely not have direct access to a communications network to handle such administrative traffic. In this case, this traffic must be routed through the host and over the existing physical link connecting the host with the disk, as shown in Figure 5.2. In other words, the host must be actively involved in bridging the gap between the administrator and the disk.

This communications model presents several challenges. For one, in most block-based storage interconnects, the disk takes the role of a passive target device and is unable to initiate a data transfer to the administrator's computer. Instead, the administrator must periodically poll the IDS to query if any alerts have been generated. Also, the administrative communication path is vulnerable to disabling by an attacker who has compromised the host system. In response to such disabling, both the administrator and the IDS could treat a persistent loss of communication as a policy violation. The IDS could alter its behavior as described in Section 5.7.4, while additionally logging any subsequent alerts for later transmission to the administrator.

The communications channel between the disk-based IDS and the administrator must be protected both from eavesdropping and tampering by an attacker. Such a secure channel can be implemented using standard cryptographic techniques. For the network intruder and rogue software concerns in our security model, it is sufficient for secret keys to be kept in the disk's firmware. If physical attacks are an issue, disk secure coprocessors can be used [Zhang02]. Such additions are not unreasonable for future system components [Gobioff99, Lie00].

5.7.3 Monitoring for Policy Violations

Once the administrative policy is received by a disk-based IDS, all storage requests arriving at the disk should be checked against the set of violating interface actions. A check should be performed for every block in a request: a write to block 72 of length 8 blocks should check for violating actions on any of blocks 72–79. As this check is in the critical path of every request, it should be implemented as efficiently as possible.

Some file system objects, such as the directory listing for a directory containing many files, will span multiple sequential disk blocks. Space for such objects will generally be allocated by the file system in a multiple of the file system block (hereafter, *fs-block*) size. We found it convenient for our disk-based IDS prototype to evaluate incoming requests by their impact on entire fs-blocks instead of on individual disk blocks: *Generate the alert "the file /sbin/fsck was modified" when a write to fs-block #35 causes the contents of fs-block #35 to change.* When monitoring at the level of fs-blocks, a disk-based IDS may need to

momentarily queue several pending requests (all of which affect a single fs-block) in order to evaluate their combined effect on the fs-block atomically.

When checking the legality of write requests, a disk-based IDS may need to first fetch the previously written (old) data for that block from the disk. The old data can then be compared with the new data in the write request to determine which bytes, if any, are actually changed by the write. Such a check would also be necessary when a block contains more than one file system object—for example, a single block could contain several file fragments or inode structures—and different administrative policies apply to the different objects. A similar check allows the system to quell alerts that might otherwise be generated during file system defragmentation or reorganization operations; no alerts are generated unless the actual contents or attributes of the files are modified.

5.7.4 Responding to Policy Violations

For an intrusion detection system in a workstation disk, the default response to a policy violation should be to prepare an administrative alert while allowing the request to complete. This is because the operating system may halt its forward progress when a locally-attached disk returns an error or fails to complete a request, especially at boot time or during crash recovery.

Pennington et al. discuss several other possible responses for a storage-based IDS after a policy violation [Pennington03]. These include artificially slowing storage requests while waiting for an administrative response, and internally versioning all subsequent modifications to aid the administrator in post-intrusion analysis and recovery [Strunk00]. To support versioning without needing to modify the file system, a disk-based IDS can copy the previous contents of written blocks to a private area on the disk that is inaccessible from the host computer. The administrator can later read the contents of this area over the administrative communication channel.

After a policy violation, the IDS should make note of any dynamic changes to the on-disk file system structure and adjust its behavior accordingly. For example, for the policy specified in Section 5.7.1, when a new file is created in the `/sbin` directory the system should first generate an alert about the file creation and then begin monitoring the new file for changes.

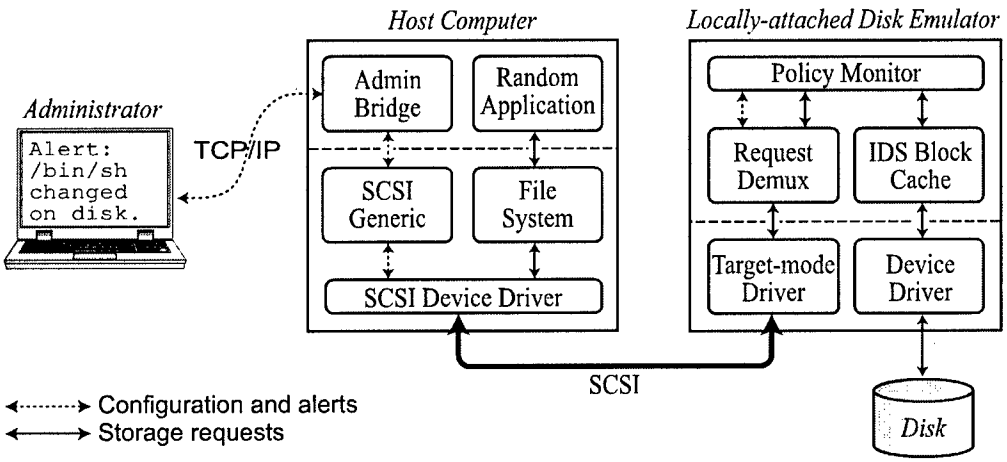


Figure 5.4: Disk-based IDS prototype architecture. This figure shows the communications flow between a Linux-based host computer and the locally-attached, FreeBSD-based IDD. The shaded boxes are key components that support disk-based IDS operation. Ordinary storage traffic is initiated by application processes, passes across the SCSI bus, is checked by the policy monitor, and is finally serviced by the disk. Administrative traffic is initiated by the administrator, passes across a TCP/IP network, is received by the bridge process on the host computer, passes across the SCSI bus, and is finally serviced by the policy monitor. The sample alert displayed on the administrator's console originated in the policy monitor.

5.8 Prototype Implementation of IDD

We built a prototype disk-based IDS called *IDD* (Intrusion Detection for Disks). The IDD takes the form of a PC masquerading as a SCSI disk, enhanced with storage-based intrusion detection. From the perspective of the host computer, the IDD looks and behaves like an actual disk. This section describes architectural and implementation details of this prototype.

5.8.1 Architecture

Figure 5.4 shows the high-level interactions between the IDD, the administrator, and the intermediary host computer. The three major components that implement the disk-based IDS functionality are the bridge process on the host computer and the request demultiplexer and policy manager on the IDD.

The bridge process forwards commands from the administrator to the IDD and conveys administrative alerts from the IDD to the administrator. The request demultiplexer identifies which incoming SCSI requests contain administrative data and handles the receipt of administrative policy and the transmission of alerts. Together, these components implement the administrative communications channel, as described in Section 5.8.2.

The policy monitor handles the mapping of administrative policy into violating interface actions. It also monitors all ordinary storage traffic for real-time violations, and generates alerts to be fetched by the administrator. This management of administrative policies is discussed further in Section 5.8.3.

5.8.2 Administrative Communication

The administrative communications channel is implemented jointly by the bridge process and the request demultiplexer. The administrator sends its traffic directly to the bridge process over a TCP/IP-based network connection. The bridge process immediately repackages that traffic in the form of specially-marked SCSI requests and sends those across the SCSI bus. When these marked requests arrive inside the IDD, they are identified and intercepted by the request demultiplexer. Encryption of messages is handled by the administrator's computer and the request demultiplexer.

The repackaging in the bridge process takes different forms depending on whether the administrator is sending new policies (outgoing traffic) or polling for new alerts (incoming traffic). For outgoing traffic, the bridge creates a single SCSI WRITE 10 request containing the entire message. The request is marked as containing administrative data by setting a flag in the SCSI command descriptor block[11]. The request is then sent to the bus using the Linux SCSI Generic pass through driver interface.

For incoming traffic, the bridge creates either one or two SCSI READ 10 requests. The first request is always of fixed-size (we used 8 KB) and is used to determine the number of bytes of alert data waiting in the IDD to be fetched: The first 32 bits received from the IDD indicate the integer number of pending bytes. The remaining space in the first request is filled with waiting data. If there is more data waiting than fits in the first request, a second request immediately follows. This second request is of appropriate size to fetch all the remaining data. These requests are marked as containing administrative data and sent in the same manner as for outgoing traffic. Once the bridge has fetched all the waiting data, it forwards the data to the administrator over the network.

Outgoing and incoming messages contain sequence and acknowledgment numbers, to ensure that policies or alerts are not mistakenly or otherwise dropped. Our implementation sends one pair of messages (outgoing and incoming) per second by default. In order to reduce administrator perceived lag, this frequency is temporarily increased whenever recent messages contained policies or alerts.

[11] We set bit 7 of byte 1 in the Write 10 and Read 10 command descriptor blocks. This bit is otherwise unused and is marked as reserved in the SCSI-3 specification. This method was the simplest of several options we considered—other options included using vendor-specific operation codes for administrative reads and writes, or using Mode Sense and Mode Select to access administrative mode pages.

5.8.3 Administrative Policy Management

The policy monitor bridges the semantic gap between the administrator's policy statements and violating interface actions, and it audits all storage traffic from the host computer in real time. The policy monitor operates at the file system block (fs-block) level, as discussed in Section 5.7.3. Request block numbers are converted to the relevant partition and fs-block numbers upon request arrival. (Hereafter in this section, "block" refers to a fs-block.)

The IDD has a relatively simple semantically-smart [Sivathanu03] understanding of the on-disk file system structures. The IDD currently understands the ext2 file system used by Linux-based host computers [Card94]; to support this we hard-coded the structure of on-disk metadata into the policy manager. For ext2 this included the ext2 superblock, inode, indirect block, and directory entry structures. As an alternative to hard-coding, we envision a more flexible administrative interface over which the relevant file system details could be downloaded.

As administrative policy is specified, the IDD receives a list of files whose contents should be watched in particular ways (e.g., for any change, for reads, and for non-append updates). For each of these watched files, the IDD traverses the on-disk directory structure to determine which metadata and data blocks are associated with the file. Each such block is then associated with an access check function (ACF) that can evaluate whether a block access violates a given rule. For example, the ACF for a data block and the "any change" policy would simply compare the old contents of the block with the new. The ACF for an inode block and the "non-append updates" policy would compare the old and new contents to ensure that the access time field and the file size field only increased. The ACF for a directory entry block and the "any change" policy would check the old and new contents to determine if a particular filename-to-inode-number mapping changed (e.g., `mv file1 file2` when `file2` is watched) and, if so, that the new inode's file contents match the old inode's file contents.

After a block is associated with an ACF, it is added to the watched block table (WBT). The WBT is the primary structure used by the policy manager for storage request auditing. It is stored in private, reserved disk space for persistence and paging. A WBT entry contains a monitored block number, a pointer to the associated ACF, and a human-understandable explanation (e.g., *the contents of the file /bin/netstat were modified*) to be sent whenever the ACF reports a violation. The IDD maintains a separate WBT for each monitored partition, as well as a WBT for the partition table and other unpartitioned disk space.

When a storage request arrives from the host computer, its blocks are checked against the appropriate partition's WBT. If any blocks are indeed watched, the associated ACFs for those blocks are invoked to determine whether an alert should be generated. As discussed in Section 5.7.3, checking the validity of a write request may require the old data to be read from the disk. Blocks that have a high probability of being checked, such as those containing monitored inodes and 15 directory entries, are cached internally by the IDD to reduce IDS-related delays. Note that this cache is primarily needed to quickly execute ACFs on block updates that will not generate an alert; generally, we are unconcerned with performance when alerts are to be generated. Examples of appropriate cache uses include directory blocks in which a subset of names are watched and inode blocks in which a subset of inodes are watched.

5.9 IDD Evaluation

This section examines the performance and memory overheads incurred by IDD, our prototype disk-based IDS. As hoped, we find that these overheads are not unreasonable for inclusion in workstation disks.

5.9.1 Experimental Setup

Both the host computer and the locally-attached disk emulator are 2 GHz Pentium 4-based computers with 512MB RAM. The disk emulator runs the FreeBSD 5.1 distribution and makes use of the FreeBSD target-mode SCSI support in order to capture SCSI requests initiated by the host computer. The host computer runs the Red Hat Linux 7.3 distribution. The machines are connected point-to-point using QLogic QLA2100 fibre channel adapters. The backing store disk on the emulator is a Fujitsu

MAN3184MP connected to an Adaptec 29160N Ultra160 SCSI controller. In an effort to exercise the worst-case storage performance, the disk emulator was mounted synchronously by the host computer and caching was turned off inside the backing store disk.

We do not argue that embedded disk processors will have a 2 GHz clock frequency; this is perhaps an order of magnitude larger than one might expect. However, an actual disk-based IDS would be manually tuned to the characteristics of the disk it runs on and would therefore run more 16 efficiently than the IDD, perhaps by as much as an order of magnitude. To compensate for this uncertainty, we report processing overheads both in elapsed time and in processor cycle counts, the latter of which provides a reasonably portable estimate of the amount of work performed by the IDD.

Our experiments use microbenchmarks and two macrobenchmarks: PostMark and SSH-build. The PostMark benchmark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [Katcher97]. It creates a large number of small randomly-sized files (between 512 B and 16 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 10,000 transactions on 200 files, and the biases for transaction types are equal.

The SSH-build benchmark [Seltzer00] was constructed as a replacement for the Andrew file system benchmark [Howard88]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v. 1.2.27 (approximately 1MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables.

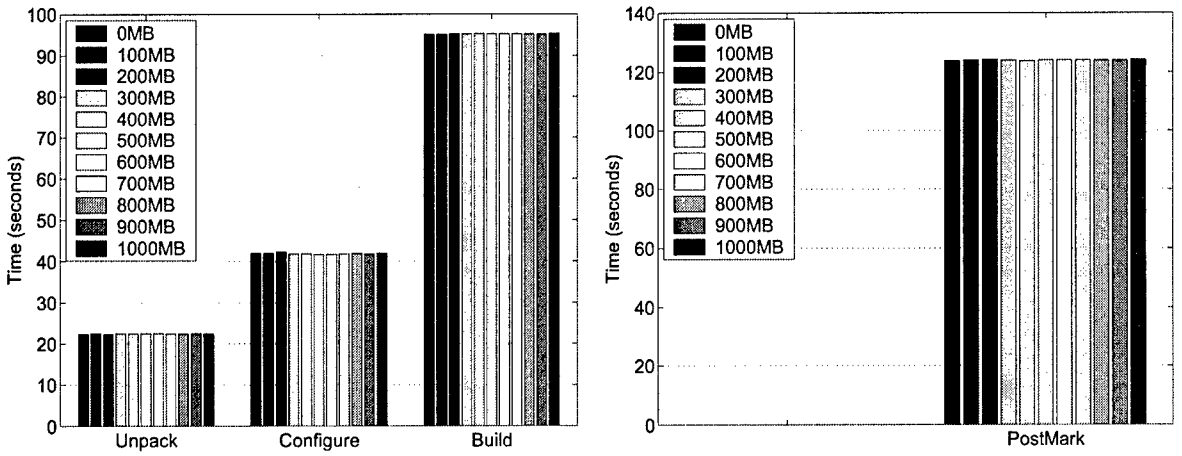
5.9.2 Common-case Performance

When new requests arrive from the host computer, the IDD checks whether any of the request's blocks are in the WBT. In the (very) common case, no matches are found. This means there are no policy implications for that request, so no further IDS processing is required.

As shown in the second column of Table 5.6, the WBT lookup takes very little time and requires less than 1500 cycles. As expected, the added latency has a minimal effect on application performance: Figures 4(a) and 4(b) show that the application run times for PostMark and SSH-build do not change when varying the number of monitored blocks over a large range.

The overheads of the IDS infrastructure itself are also small. The difference in PostMark run times between running the disk emulator with no IDS and running it with an empty IDS (with no policy set, and therefore no WBT lookup) was less than 1%.

The size of the WBT is approximately 20 bytes per watched block. To put this in context, Tripwire's default ruleset for Red Hat Linux expanded on our testbed host to 29,308 files, which in turn translates to approximately 225,000 watched file system blocks. This would require 4.5MB to keep the entire WBT in core. This number can be substantially reduced, with a small slowdown for the non-common-case IDS performance, by only keep 4 bytes per watched block in core (or, better yet, keeping lists extents of watched blocks in core) and demand paging the remainder of the WBT. This would reduce the memory cost to 900KB or less. (At time of this writing, our prototype does not yet demand page the WBT.)



a) SSH-build benchmark

b) Postmark benchmark

Figure 5.5: Application benchmarks. These graphs show the impact of the initial WBT check on application performance. These experiments were run with the IDS engaged and watching different amounts of data; the file system was constructed such that no policy violations were generated by the disk accesses. The data indicate virtually no scaling of the overhead as a function of the amount of watched data. 0MB is the leftmost thin bar in each group.

Old data status	WBT check time	Fetch old data	ACF check time	Disk access time	Total time
Data block write: Changes the data in one block of a monitored file.					
Not cached	0.655 µs (0.076 µs) 1310 cycles	4,210 µs (1,580 µs)	0.447 µs (1.48 µs) 893 cycles	5,840 µs (21.0 µs)	10.1 ms
Cached	0.554 µs (0.102 µs) 1110 cycles	3.21 µs (0.340 µs) 6410 cycles	0.270 µs (0.033 µs) 539 cycles	4,390 µs (1,500 µs)	4.39 ms
Metadata block write: Changes the last-modify-time in a monitored inode.					
Not cached	0.548 µs (0.094 µs) 1100 cycles	3,860 µs (1,420 µs)	0.250 µs (0.029 µs) 501 cycles	5,700 µs (716 µs)	9.56 ms
Cached	0.594 µs (0.134 µs) 1190 cycles	3.07 µs (0.477 µs) 6140 cycles	0.772 µs (2.59 µs) 1540 cycles	3,810 µs (1,780 µs)	3.81 ms

Table 5.6: Microbenchmarks. This table decomposes the service time of write requests that set off rules either on file data or metadata. The numbers in parentheses show the standard deviation of the average. Two cases for each are shown, where the requested blocks either are or are not already present in the IDD block cache. With caching enabled, the total time is dominated by the main disk access. When blocks are not cached, the service time is roughly doubled because an additional disk access is required to fetch the old data for the block. The three phases of an IDS-watched block evaluation are described in Section 5.8.3. Numbers shown in bold represent the dominating times in these experiments.

5.9.3 Additional Checks on Watched Blocks

When one or more of a write request's blocks are found in the WBT, additional checks are made to determine whether the request violates administrative policy. If necessary, the old data is fetched from the disk in order to determine which bytes are changed by the write. The ACF is then executed; if this determines that the request illegally modifies the data, an alert is generated and queued to be sent to the administrator.

We used microbenchmarks to measure the performance of various types of rules being matched, which we show in Table 5.6. Each alert was triggered 200 times, accessing files at random which had rules set for them. In order to determine if a rule has been violated, and to send an alert, several operations need to take place. For any write to a file data block, for example, the IDD needs to determine first if the given block is being watched; this takes 0.665 µs. It then reads the modified block on the disk to see if the write results in a modification; this takes 4,207 µs if the IDD does not have the block to be modified in cache, and 3.208 µs if it does. If the write is to an inode block, it takes 0.25 µs to determine if it changes a watched field in that inode. Finally, an alert is generated and the write is allowed to complete, this takes 5,841 µs if the block wasn't in cache (one revolution from the completion of the read request of the same block), and 4,386 µs if it was in cache. Table 5.6 summarizes the time taken by all of these actions, as well as the number of CPU cycles. We view these numbers as conservative estimates of CPU costs, because the IDS code is untuned and known to have inefficiencies. Nonetheless, the results are quite promising.

5.9.4 Frequency of IDS Invocation

To understand the frequency of overheads beyond the common-case performance, we examined 11 months worth of local file system traces from managed desktop machines in CMU's ECE Department. The traces included 820,145,133 file operations, of which 1.8% were modifications to the disk. Using these traces, we quantify the frequency of two cases: actual rule violations and nonrule- violating updates (specifically, incidental updates to shared inode blocks or directory blocks).

We examine the traces for violations of the ruleset used by Pennington et al. [Pennington03], which includes Tripwire's default rule set for Red Hat Linux and a few additional rules regarding hidden names and append-only audit logs. This expanded out to rules watching 29,308 files, which in turn translates to approximately 225,000 blocks being watched. In the traces, 5350 operations (0.0007% of the total) impacted files with rules set on them. All but 10 of these were the result of nightly updates to configuration files such as /etc/passwd and regular updates to system binaries. As discussed in [Pennington03], a method for coordinating administrative updates with the IDD would convert the response to planned updates from alerts to confirmations.

The first class of non-rule-violating updates that require ACF execution is shared inode blocks. Our prototype notices any changes to an inode block containing a watched inode, so it must also determine if any given modification impacts an inode being watched. In the case of the ext2 file system [Card94], 32 inodes share a given block. If any inode in a given block is watched, an update to one of the 31 remaining inodes will incur some additional overhead. To quantify this effect, we looked at the number of times an inode was changed which was in the same block as a watched inode. For this analysis, the local file systems of 15 computers were used as examples of which inodes share blocks with watched files. For our traces, 1.9% of I/Os resulted in changes to inode blocks. Of these, 8.1% update inode blocks that are being watched (with a standard deviation of 2.9% over the 15 machines), for a total of 0.15% of I/Os requiring ACF execution. Most of the inode block overlap resulted from these machines' `/etc/passwd` being updated nightly. This caused its inode to be in close proximity with many short-lived files in `/tmp`. On one machine, which had its own partition for /tmp, we found that only 0.013% of modifications caused writes to watched inode blocks. Using the values from Table 5.6, we compute that the extra work would result in a 0.01–0.04% overhead (depending on the IDD cache hit rate).

Similarly, the IDD needs to watch directories between a watched file and the root directory. We looked at the number of namespace changes the IDD would have to process given our traces. Using the same traces, we found that 0.22% of modifications to the file system result in namespace changes that an ACF would need to process in order to verify that no rule was violated. Based on the Table 5.6 measurements, these ACF invocations would result in a 0.004–0.01% performance impact, depending on IDD cache hit rate.

5.10 Discussion of IDD Feasibility

Workstation disks are extremely cost-sensitive components, making feature extensions a tough proposition. Security features, however, are sufficiently important and marketable today that feature extensions are not impossible. To make for a viable business case, uninterested customers must observe zero cost. the cost of any hardware support needed must be low enough that The profits from the subset of customers utilizing (and paying for) the IDS features must compensate for the marginal hardware costs incurred on all disks produced. A similar situation exists in the network interface card (NIC) industry, where 3com embedded sufficient hardware in their standard NICs to allow them to sell firewall-on-NIC extensions to the subset of interested security-sensitive customers [3Com01a]; the purchased software is essentially an administrative application that enables the support already embedded in each NIC [3Com01b] plus per-seat licenses.

Evaluation of our disk-based IDS prototype suggests that IDS processing and memory requirements are not unreasonable. In the common case of no ACF invocations, even with our untuned code, we observe just a few thousand cycles per disk I/O. Similarly, for a thorough ruleset, the memory required for IDS structures and sufficient cache to avoid disk reads for non-alert ACF executions (e.g., shared inode blocks) is less than a megabyte. Both are within reasonable bounds for modern disks. They may slightly reduce performance, for example by reducing the amount of disk cache from 2–8MB (representative values for today's ATA drives) by less than one megabyte. The overall effect of such changes should be minor in practice, since host caches capture reuse while disk caches help mainly with prefetching. Moreover, neither the memory nor the CPU costs need be incurred by any disk that does not actually initialize and use its IDS functionality.

In addition to the IDS functionality, a disk-based IDS requires the disk to be able to perform the cryptographic functions involved with the secure administrative channel. This requires a key management mechanism and computation support for the cryptography. Again referring to the 3com NIC example, these costs can be very small. Further, various researchers have proposed the addition of such functionality to disks to enable secure administration of access control functions [Aguilera03, Gobioff99], and it can also be used to assist secure bootstrapping [Arbaugh97].

In summary, storage-based intrusion detection would be most effective if embedded in the local storage components of individual workstations. From experiences developing and analyzing a complete disk-based IDS, implemented in a disk emulator, we conclude that such embedding is feasible. The CPU and memory costs are quite small, particularly when marginal hardware costs are considered, and would be near-zero for any disk not using the IDS functionality. The promise of enhanced intrusion detection capabilities in managed computing environments, combined with the low cost of including it, makes disk-based intrusion detection a functionality that should be pursued by disk vendors.

5.11 Diagnosis of Intrusions

Once an intrusion is detected and stopped, the administrator would like to understand what happened. There are several goals of post-intrusion diagnosis. These goals include determining how the intruder gained access to the system, when they gained access, and what they did once they got in.

In current systems, the administrator is poorly equipped to answer these questions because, once an intruder gains control of the computer system, no information can be trusted; the intruder has the ability to erase and obfuscate incriminating evidence. As a result, administrators usually perform only a cursory

review of the post-intrusion system, hoping that the intruder overlooked some obviously incriminating evidence.

The administrator who wishes to dig deeper into the system in search of answers to these questions is faced with a daunting task. First, she must scour the free space of the storage system in search of disk blocks from deleted data and log files that have not yet been overwritten; simplifying 14 this task has been the focus of several forensic tool developers [Farmer00b, Kruse02, Northcutt01]. Second, and far more difficult, she must then piece together this incomplete information and form hypotheses about the details of the intrusion; this is, at best, a black art.

Self-securing storage has the ability to brighten this dismal picture. It makes available a large amount of information that was previously very difficult or impossible to obtain. This information provides several new diagnosis opportunities by highlighting system log file tampering, exposing modifications made by the intruder, and potentially allowing the capture of the intruder's exploit tools.

5.11.1 Post-mortem Information

Self-securing storage maintains an audit log of all requests and keeps the old versions of files. Therefore, the administrator is no longer relegated to working with just the remaining fragments after an intrusion. The administrator now has the ability to view the sequence of storage events as well as the entire state of storage at any point in time during the intrusion. This history means that the pre-diagnosis forensics effort is no longer needed because the storage system retains this information automatically.

Because self-securing storage removes the need for the forensics effort, performing post-intrusion diagnosis is no longer an all-or-nothing proposition. The forensics effort that was previously required meant that only in extreme situations would an intrusion be investigated seriously. With all of the storage information immediately available, the administrator can spend an appropriate amount of effort interpreting post-mortem information, with near-zero invested in pre-diagnosis forensics.

Self-securing storage also takes intrusion diagnosis out of the critical path. Since the intrusion state is saved within the history data, diagnosis can be started after post-intrusion recovery. The administrator can return the system to operation quickly, and then utilize the history for actual diagnosis. This relieves some of the pressure that constrains the amount of time and effort that an administrator is able to put into diagnosis.

5.11.2 New Diagnosis Opportunities

Following an intrusion, the administrator is left with many questions. Her goal is to determine exactly what happened so that she can assess the damage and ensure that the intrusion is not repeated. Some of the many questions she seeks to answer are:

- How did the intruder get in?
- When did the intrusion start?
- What tools were used?
- What data was changed?
- What data was seen?
- What tainted information was propagated through the system?
- Why was the system attacked?

Self-securing storage assists the administrator in answering the above questions by providing previously unavailable insight into storage activity. The rest of this section provides examples of information that is now available.

Highlighting audit log tampering: It is common for intruders to tamper with log files in an attempt to cover their tracks. With self-securing storage, not only is it possible to tell that the tampering occurred [Schneier98, Schneier99], but the administrator can locate and retrieve the exact entries that were erased

or modified. This means that when an intruder attempts to conceal their actions by doctoring log files, they are, in fact, doing just the opposite.

Capturing exploit tools: It is also common for intruders to load and run exploit tools locally after they initially gain entry. This can be done for several reasons, such as subsequent phases of a multi-stage intrusion, Trojan programs to capture passwords, or exploit tools that can be used to attack other machines. Normally written to the file system prior to being executed, these tools are automatically captured and preserved by self-securing storage. The capture of such exploit tools and "root kits" makes it easier to find the intruder's point of entry into the system and the weakness(es) they exploited [Venema00].

Exposing modifications: In addition to capturing exploit tools, any changes to system files and executables are obvious based on the storage device's audit log. This allows the administrator to see the scope of damage to the OS and other sensitive system software. The storage device tracks and "exposes" even legitimate software updates made by the administrator unless an out of band method is used to coordinate those specific modifications (e.g., via the secured administrative interface). Additionally, the device applies its own timestamps to modifications, in addition to those supplied by client systems. This allows a clear picture of the sequence of events even when the intruder may have manipulated the creation, modification, or access times of files [Farmer00a].

Recording reads: Since the server's audit log records all READ and WRITE operations, the administrator can estimate the real damage that may have been done by the intruder. For example, the storage log allows one to bound the set of files read by a system and the likelihood that the intruder has read specific confidential files. Additionally, the log can assist the administrator in determining whether intruder-modified files were read by legitimate users. This allows her to gauge the potential spread of misinformation, planted by the intruder for sabotage purposes. It is important to note that this is only an approximation since the client system's cache can obscure some storage requests. It does, however, provide a way of gauging intrusion damage that previously could not be measured at all. This even provides information for inferring the intruder's motivation for attacking the system. With a more complete view of the intruder's actions, the administrator has a greater chance of determining whether the intent was espionage, sabotage, or merely "entertainment."

Self-securing storage provides a new window into the scope of damage and the intentions of a digital intruder. This information can be invaluable in determining the impact of the compromise, preventing future intrusions, and catching those responsible. Clearly, much future research and experience will be needed to create post-mortem diagnosis tools that exploit the new wealth of information provided by self-securing storage.

5.12 Recovery from Intrusions

Intrusion detection and post-intrusion diagnosis are parts of a good computer security strategy. An efficient and effective plan for recovery, however, is a necessity. Maintaining well-administered and up-to-date systems will minimize the occurrence of intrusions, but they will inevitably happen, so it is critical that recovery be efficient and thorough.

In conventional systems, intrusion recovery is difficult and time-consuming, in terms of both system down-time and administrator time. It requires a significant amount of the administrator's time because there are many, error-prone steps involved in returning a compromised computer system to a safe state. Self-securing storage, with its history information, facilitates this task.

5.12.1 Restoration of Pre-intrusion State

All storage in a conventional system is suspect after an intrusion has occurred. As a result, full recovery necessitates wiping all information via a reformat of the storage device, re-installing the operating system from its distribution media, and restoring users' data from the most recent pre-intrusion backup. Shortcutting these steps can result in tainted information remaining in the system, yet following these steps results in significant down-time and inconvenience to users.

Self-securing storage addresses this unfortunate situation in several ways. First, all pre-intrusion state is preserved on the device. Therefore, once diagnosis yields an approximate time for the intrusion, the restore described above can be a single step for the administrator (issuing the *copy forward* command to bring forward the system state from before the intrusion). Second, the granularity of the restoration is not limited to the most recent backup; any or all files can be restored from arbitrarily close to the time of the intrusion. Third, restoration in self-securing storage is non-destructive. The administrator can quickly return the system to a safe state so that users may utilize the computer system, while preserving all storage history (including any intrusion evidence therein) in the storage server. The administrator may then perform detailed diagnosis at her leisure.

5.12.2 Performance of Restoration

To investigate the time required to return a system to operation, we gathered traces of all NFS activity to our local NFS server and replayed them against our prototype system. We then measured the amount of time required to copy forward the entire state of the device from various points in the past.

The prototype self-securing NFS server used for this experiment was a dual 600 MHz Pentium III system, running RedHat Linux 6.1 using kernel version 2.2.20. The traces were collected from an NFS version 2 server, running Linux 2.2. The traced NFS server supports the research efforts of approximately 30 graduate students, faculty and staff. The server contained approximately 66.5 GB of capacity of which 33.5 GB was consumed by 272,521 files. The workload on the server was mainly generated by code development and word processing activities of the supported users.

Days	Files	MB	Time w/ rsync (s)	Time w/ audit log (s)
1	7	7.3	3208	26.4
2	349	34.4	3311	139.0
3	359	66.8	3324	167.5
4	361	76.2	3350	198.4
5	362	66.8	3329	208.3
6	362	66.8	3325	212.6

Table 5.7: Recovery statistics. This table summarizes the results of completely restoring the state of the NFS server as a function of the detection latency. The times shown are based on using either `rsync` or the data in the device's audit log to create the list of files that must be copied forward. The large difference is due to `rsync` executing a STAT on every file at both the current time and the recovery time—a total of approximately 545,000 calls.

To evaluate both the number of files and bytes of data (due to legitimate use) that the administrator may have to recover, an initial snapshot of the file system was copied onto the prototype system, followed by the first day's worth of activity. The state of the storage system was returned to the original condition by copying forward the snapshot state. During this process, the time to recover the initial state, the number of affected files, and the total size of the affected files were recorded. This experiment was conducted for each of one through six days of activity, each time beginning with the original snapshot.

Two different methods were tested for selecting the files to recover. The first method utilized the `rsync` [Burns96] program to synchronize the current system state with the pre-intrusion copy of data. The `rsync` application was configured to select files based on their attributes, causing it to retrieve the attributes of all files on the device at both the current time and at the recovery time, leading to a large overhead for just building the list of files to recover. Creating this initial list required 3184 seconds—and by only checking the attributes, this method is vulnerable to manipulation of the file modification times. The second method used to create this list of files to recover is by using the device's audit log. This second method is much faster, requiring only 71 seconds to determine the list of files for the complete six days

worth of changes. In addition to being faster, this second method is not susceptible to timestamp manipulation. The results are summarized in Table 5.7.

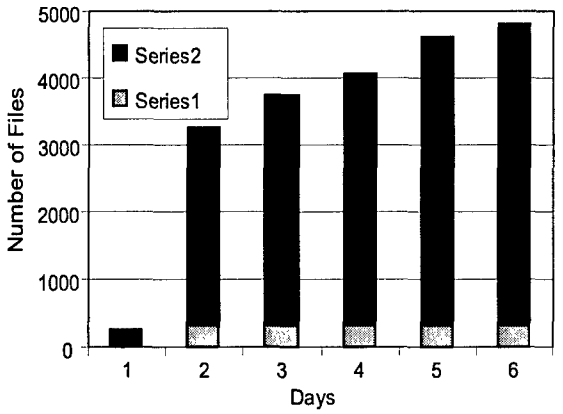


Figure 5.6: Files modified. Shows the total number of user files that must be restored to return to the pre-intrusion state as a function of the detection latency.

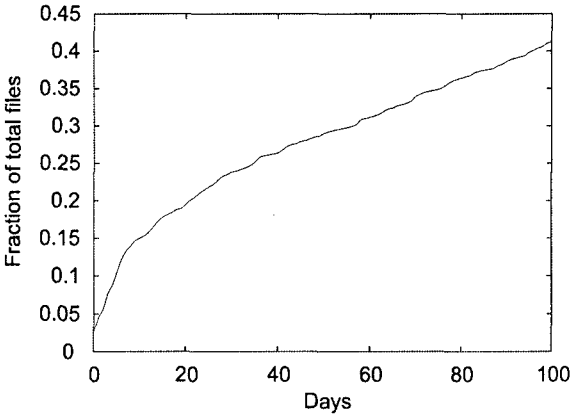


Figure 5.7: Fraction of files modified. Shows the fraction of files that were modified in the examined Microsoft systems, as a function of the number of days. The average number of files per file system was approximately 13,000.

Based on Figure 5.6, it is obvious that very few files need to be restored relative to the total number of files that were modified. This is because a large number of files are transient. Files that were completely created then deleted between the recovery time and the current time need not be copied forward.

To project the amount of data that would need to be recovered for longer detection latencies, we examined a snapshot of over 10,000 file systems from desktop computers at Microsoft Corporation [Douceur99]. The snapshot contains a listing of all files, their size, and modification times for each of the systems. Based on the last-modified time of the files, we can project the number of files that would need to be restored as a function of the detection latency. The results are shown in Figure 5.7 as a fraction of the total number of files stored.

Based on the playback data, we can estimate the amount of time that would be required to restore a "typical" one of these desktop systems. Looking at only the time required to restore the files, the copy forward executed at a rate of 2.55 files per second on our prototype system. This means that one week worth of changes could be copied forward in $\frac{0.134 \cdot 13000}{2.55} = 683$ seconds. Likewise, the system could be restored to its state as of one month ago in $\frac{0.239 \cdot 13000}{2.55} = 1219$ seconds. We believe that the "in-time" performance of out prototype can be improved, significantly increasing the rate of recovery beyond 2.55 files per second.

5.12.3 Preservation of User Data

In addition to removing the intruder and restoring the system to a safe state, an administrator is often under considerable pressure to retain a recent version of users' work—a version from after the system was compromised. In conventional systems, the mechanics of doing this involves finding separate media on which to temporarily store the users' data, then performing the normal reformat, reinstall, restore, and finally restoring this recent version of the users' data. There also exists the challenge of determining whether this data is safe to keep at all since the intruder could have modified it.

Self-securing storage obviates the mechanical steps by automatically retaining the users' recent work. While this provides the opportunity for the administrator to easily restore it, there is still some question about the authenticity of this data. The question of intruder tampering, while still very difficult to answer, is one step closer due to the availability of the storage system's version history. The version history can

show the sequence of modifications to the data as well as, in the case of a network-attached storage device, the client and user who made the modifications. This provides additional information for tampering investigations, though they will remain very difficult.

Additionally, in current systems, the administrator is left with only two choices when restoring a given file: use the current, potentially-tainted data or use the most recent backup, losing all intermediate work. Self-securing storage provides the opportunity to restore any version in between those two extremes as well. This can be beneficial in situations where the data was (or might have been) modified by the intruder; the administrator is able to restore the version from just prior to the tampering and need not lose all changes. Careful validation of user data is still too time-consuming to use on all data, but it can be performed on an as-needed basis for important data files.

5.12.4 Application-specific Recovery

With only the above information, it is possible to accomplish recovery of data in a reasonable fashion. However, without examining the contents of files, it is difficult to determine what application level change was made.

Given a tool that understands the contents of a file, it would be possible to, in some situations, untangle changes made by a legitimate user and an intruder within a single file. For instance, the intruder may have planted a macro virus within a word processing document. A tool that understands the format of such files could remove the virus while leaving the other data intact. Modern virus detectors are able to handle this specific case, but in general, this sounds like a daunting task—creating application-specific recovery tools for data files. However, a small number of applications create most files in a given system. Therefore, a few such utilities will cover most of the data.

An additional use for these application-specific recovery tools is for bootstrapping recovery and diagnosis of complex programs, such as databases. Consider a database application that maintains a log of operations, capable of undoing or redoing its operations. A recovery tool could potentially validate the contents of the database against the database's log, ensuring that any changes made to the database contents were made through the proper interface. The database's built-in validation and auditing could then be used to dig deeper into the changes, knowing that the database log information is consistent with the actual database contents. Ammann, et al. have already approached the problem of removing undesirable but committed transactions from databases [Ammann00], assuming that any malicious transactions have been inserted via the normal database interfaces (as opposed to accessing raw storage).

5.13 Discussion

This section discusses some remaining issues involved in using self-securing storage to detect, diagnose, and recover from intrusions.

Audit log accuracy: The information stored in the device's audit log is derived from the information contained in the storage protocol. As a result, information about the requester's identity is only as good as the guarantees provided by the storage protocol itself. For instance, if protocol requests can be forged, these forged requests will be added to the audit log as seen by the storage server. When present, this limitation is inherent in the choice of storage configuration. This is only relevant to network-attached self-securing storage devices. Locally-attached disks receive all of their commands from a single OS, which does not provide any user-specific information in storage requests.

Tracking tainted data: The audit log maintained on the device shows not only the files that were written after an intrusion, but also which clients or users subsequently read (potentially) intruder-tainted data. While it is possible to consider subsequent writes (of different files) by those clients and users as suspect as well, the possible set of tainted data is likely to grow very large. Also, the probability that a file is affected via a specific WRITE decreases with each iteration. Additionally, it is not possible to completely track tainted data, since it may have been transmitted to other files in the system via external methods (e.g., printouts or word-of-mouth among users).

Additional OS information: The addition of a small number of additional fields inside each storage request would greatly increase the utility of the audit log. For instance, if each request were tagged with the process ID and name of the process that is making the request, it is much easier to determine (during both detection and diagnosis) whether a given access was benign. Additional, useful information would be an indication of the purpose of a READ operation. In current file systems, a READ operation for data retrieval looks nearly identical to a read for the purpose of executing.

Client caches: File system caches on client systems obscure traffic that would otherwise be seen by the storage server. The caches effectively act as a filter causing the audit log to only have a partial view of the OS's storage activity. For example, read caches can obscure the propagation of intruder-tainted data since it is likely to be in the client's cache. This danger is smaller in some file systems, such as NFS version 2, that perform aggressive "freshness" checks prior to returning the cached contents of a file. This freshness check is visible to the storage system, and the window of vulnerability during which the freshness check is not necessary is small (a few seconds). Client-side write caches are also a problem since short-lived files that are written, read, and deleted quickly may never be transmitted to the storage device. In this case, the existence of the file may never be known to the storage system. It would not have any associated entries in the audit log or versions in the history pool. This presents a larger problem for diagnosis, since it means that temporary files may be lost.

5.14 Hiding from Self-Securing Storage: The Game Continues

Self-securing storage has the potential to expose intruders and their actions by explicitly watching at a new point in the system. We expect it to work very well initially, and less so as intruders learn about self-securing storage and its capabilities. That is, we expect clever intruders (and those that borrow their tools) to modify their behavior in an attempt to mitigate the benefits that self-securing storage provides. This section explores some potential actions that intruders might take in an attempt to avoid detection or thwart attempts at diagnosis and recovery. We note that those in the white hats gain ground when intruders must change tactics and avoid convenient actions. Thus, we also discuss some ramifications of these new behaviors on the intruders themselves.

Minimize log file evidence: In a system protected by self-securing storage, it is not possible for an intruder to tamper with log files once they are written. Using the self-securing storage device for system log files functions much like a remote logging server. If intruders wish to hide their presence and actions, this limits the types of attacks and exploits that they can use in their initial compromise of the system. Once they gain control of the system, however, they can manipulate the logging facilities to prevent information from entering the log. This can be made a non-trivial activity. If they wish to avoid detection, they must filter log entries that they generate (as a result of their malicious activities), but they must allow normal entries to continue to be logged. Filtering too much or too little will be detectable. Additionally, it may be possible to correlate system log entries and storage log entries, further complicating the filtering process.

Use RAM-based file systems: One way of preventing capture of exploit tools and utilities is to store them in a RAM disk instead of the usual file system that is kept on self-securing storage. The problem with this approach is that the data that is written is not persistent, and a simple reboot of the machine will wipe it out. While this will erase the evidence, it will also erase any backdoor or Trojan executables that were left behind.

Manipulate memory images: Since overwriting of system executables is easily detectable, one way to create a backdoor version of a (long running) program would be to directly modify its memory image. This would leave no traces on the storage system, but is likely to be difficult to do. Additionally, the modifications could be erased by restarting the application [Castro00].

Tamper slowly: If the intruder is able to avoid having his action(s) detected until after the detection window elapses, self-securing storage will be of little help in diagnosing and repairing the damage. An intruder that is willing to make small changes over a large amount of time can increase his chances of suc-

cess. The problem that this creates for an attacker is that it takes a large amount of time to tamper with a significant amount of data.

Redirect file system requests: Once the intruder is able to compromise the OS, he can manipulate the file system code in the kernel in such a way that requests for one file are redirected to another. For example, he can use this to redirect requests from a legitimate system executable to a Trojan version of the same program. While self-securing storage would not view this as a change to the system executable (and not issue an alert), the Trojan executable will be captured, and the true executable remains intact. Again, since this strategy does not modify any of the true executables, a reboot will solve the problem.

Encrypt tools: To prevent capture of exploit tools, the intruder can use tools that are written to disk in an encrypted form and decrypted just prior to execution (in a similar manner to some viruses). The decryption key can be held in memory of the affected system (at a well-known location). As long as the key is present and correct, the tools can be used, but by removing or changing the key, it would not be possible to recover the true contents of the exploit tools stored on the compromised system. While this prevents capture, the key is necessarily stored in a volatile location; hence a restart of the system will clear it, rendering the tools useless.

Use a network loader: The attacker could utilize a network loading utility that would read an executable directly into memory and execute it. This avoids the file system all together, but is, again, not persistent. These examples show that it is possible to dodge some aspects of self-securing storage. However, they also show that doing so requires a level of expertise and effort not necessary for conventional systems.

5.15 Additional Related Work

Much related work has been discussed within this chapter. For emphasis, we note that there have been many intrusion detection systems focused on host OS activity and network communication [Axelsson98]. Also, the most closely related tool, Tripwire [Kim94], was used as an initial template for our prototype's file modification detection ruleset.

Our work is part of a recent line of research exploiting physical [Ganger01, Zhang02] and virtual [Chen01] protection boundaries to detect intrusions into system software. Notably, Garfinkel et al. [Garfinkel03] explore the utility of an IDS embedded in a virtual machine monitor (VMM), which can inspect machine state while being compromise independent of most host software. Storage-based intrusion detection rules could be embedded in a VMM's storage module, rather than in a physical storage device, to identify suspicious storage activity.

Perhaps the most closely related work is the original proposal for self-securing storage [Strunk00], which argued for storage-embedded support for intrusion survival. Self-securing storage retains every version of all data and a log of all requests for a period of time called the *detection window*. For intrusions detected within this window, security administrators have a wealth of information for post-intrusion diagnosis and recovery.

Such versioning and auditing complements storage-based intrusion detection in several additional ways. First, when creating rules about storage activity for use in detection, administrators can use the latest audit log and version history to test new rules for false alarms. Second, the audit log could simplify implementation of rules looking for patterns of requests. Third, administrators can use the history to investigate alerts of suspicious behavior (i.e., to check for supporting evidence within the history). Fourth, since the history is retained, a storage IDS can delay checks until the device is idle, allowing the device to avoid performance penalties for expensive checks by accepting a potentially longer detection latency.

5.16 Conclusions and Future Work

A storage IDS watches system activity from a new viewpoint, which immediately exposes some common intruder actions. Running on separate hardware, this functionality remains in place even when client OSes or user accounts are compromised. Our prototype storage IDS demonstrates both feasibility and efficiency

within a file server. Analysis of real intrusion tools indicates that most would be immediately detected by a storage IDS. After adjusting for storage IDS presence, intrusion tools will have to choose between exposing themselves to detection and being removed whenever the system reboots.

In continuing work, we are developing a prototype storage IDS embedded in a device exporting a block-based interface (SCSI). To implement the same rules as our augmented NFS server, such a device must be able to parse and traverse the on-disk metadata structures of the file system it holds. For example, knowing whether `/usr/sbin/sshd` has changed on disk requires knowing not only whether the corresponding data blocks have changed, but also whether the inode still points to the same blocks and whether the name still translates to the same inode. We have developed this translation functionality for two popular file systems, Linux's ext2fs and FreeBSD's FFS. The additional complexity required is small (under 200 lines of C code for each), simple (under 3 days of programming effort each), and changes infrequently (about 5 years between incompatible changes to on-disk structures). The latter, in particular, indicates that device vendors can deploy firmware and expect useful lifetimes that match the hardware. Sivathanu et al. [Sivathanu03] have evaluated the costs and benefits of device-embedded FS knowledge more generally, finding that it is feasible and valuable.

Another continuing direction is exploration of less exact rules and their impact on detection and false positive rates. In particular, the potential of pattern matching rules and general anomaly detection for storage remains unknown.

6 BLOCKING INVASIVE SOFTWARE USING RATE LIMITING MECHANISMS

6.1 An Introduction to Mass-Mailed Worms

In recent years the Internet has experienced an increasing number of malicious programs propagating through electronic mail. High-profile worms such as *I love you* [NetworkAssoc00] and *Sircam* [NetworkAssoc01] are but a few historical examples. The more outbreak of *MyDoom* [NetworkAssoc04] substantially degraded network services and was directly responsible for a massive Distributed Denial of Service (DDoS) attack. An earlier outbreak, *SoBig* [NetworkAssoc03], utilized sophisticated hybrid propagation techniques to replicate over network shares and emails, resulting in a rapid infection rate that surpassed those achieved previously. Both SoBig and MyDoom disrupted normal network operations and caused unexpected downtime and an increase in associated IT expenses. In the next few sections, we focus on studying fundamental behavior and characteristics of two mass-mailing worms, SoBig and MyDoom, from real traffic traces, which may lead to new insights to automatically detect, suppress and stop their propagation.

The severity of these attacks can be attributed to a number of factors, including efficient mechanisms for targeting victims and insufficient defense mechanisms. Previous work on defending email-based attacks focuses primarily on filtering techniques implemented at the outgoing mail servers [Williamson03]. Unfortunately, newer instances of mass-mailing worms come equipped with their own SMTP engines [NetworkAssoc03, NetworkAssoc04], and therefore are capable of bypassing filters or detection mechanisms deployed at the outgoing mail server. An alternative approach would be examining incoming mails. Unfortunately, the prevalence of email messages with malicious content makes it difficult and costly to perform comprehensive filtering before the emails reach end users. The worm is then unlikely to be detected in the end user's mailbox because most users update their anti-virus software/signatures infrequently, if at all. As a result, we will continue to see outbreaks of email-based attacks, the frequency and sophistication of which will only increase with time.

An email worm is a program that propagates by sending copies of itself to recipients via electronic mail. Once a recipient opens the email attachment, the malicious program executes on the victim's machine and further propagates itself. Typically, the worm program chooses its targets by harvesting email addresses from the victim's address book, web cache, and hard disk. The worm then sends mails containing its own code to targets in an attempt to repeat the infection cycle.

6.1.1 Background

Email worms are different from scanning worms such as Code Red and Slammer [CERT03, Zou02]. The latter typically exploits a vulnerability (e.g., buffer overflow) in the network services running on a target machine, and they must perform IP scanning to find vulnerable targets. Email worms infect hosts by tricking users into inadvertently executing malicious code.

Unlike a scanning worm, which needs to aggressively search for new victims to compromise, an email worm has much higher hit rate because it obtains targets from victim machines. Unlike traditional email viruses, mass-mailing worms do not limit their targets strictly to those in a victim's address book. Worms like SoBig and MyDoom utilize aggressive spreading techniques such as harvesting legitimate domain names from victim machines (e.g., by scanning web caches and hard disks) then attempting to construct probable addresses. These worms also use social engineering techniques (e.g., disguise as a system error message) to lure users to open t he malicious attachment. Some mass-mailing worms even spread using avenues other than mail. For in stance, both SoBig and MyDoom replicated over file sharing clients (e.g., Kazza).

Both SoBig and MyDoom exhibit unique and atypical traffic patterns. For instance, clients infected with SoBig and MyDoom used their own SMTP engines in propagation attempts. Under normal circumstances, connections to outside SMTP servers are highly unusual for most enterprise clients.[12]

6.1.2 Trace Data

Our study of SoBig and MyDoom was conducted on real network traffic traces. These traces are collected from the edge router of CMU's Electrical and Computer Engineering (ECE) Department.

The ECE network consists a total of 1,128 hosts, of which 4 are mail servers, 3 are DNS servers and around 1,000 are normal end host clients. We recorded, in an anonymous form, all IP and common second layer headers of network traffic (e.g., TCP or UDP) entering or exiting the ECE network. The header information for TCP and UDP contain the source and destination addresses and port numbers. We also recorded DNS traffic payloads which were anonymized accordingly.

For this study, we examined traces from two specific time periods. The first is from August 5, 2003 to September 5, 2003. This period contains the outbreak of the "SoBig" worm [NetworkAssoc03], unleashed on August 16th, 2003. The second is from January 15, 2004 to February 5, 2004, which corresponds with the outbreak of "MyDoom" [NetworkAssoc04], unleashed on January 24th, 2004. Residual effects of both worms lingered on for months, but the effects of the infection were most prominent during the first two weeks.

6.2 Analysis

In this section we describe our study of the SoBig and MyDoom traces. We begin by examining traffic patterns before and during the worm outbreaks. More specifically, we investigated change in the rate of outgoing TCP, SMTP, and DNS-related traffic. We found that traffic anomalies were easily detectable from monitoring client machine traffic patterns.

For our analysis, it is advantageous to identify the infected hosts and contrast their behavior with that of normal hosts. We devised a simple heuristic to identify infected hosts.[13] The heuristic involves contrasting outgoing SMTP connections from a host before and during the outbreak. An uninfected host should make no or very few outgoing SMTP connections (other than to its designated mail server). If a host changes its behavior by creating a large number of SMTP connections during the outbreak, it is marked as a candidate for being infected. We further refined the heuristic by considering the sizes of the message payload—those with payload similar to that of the worms reported by Symantec are suspects [NetworkAssoc03, NetworkAssoc04]. Note that this is only a heuristic and we cannot identify infected hosts with 100% certainty. Using this heuristic, we identified 5 infected hosts during the SoBig outbreak and 6 infected hosts during the MyDoom outbreak. In the analysis that follows, those infected hosts refer to the sets identified by this heuristic.

In the following sections we analyze data before, during, and after the worm outbreaks. We study the number of outgoing TCP traffic flows, the number of distinct IPs contacted by TCP flows, and the relationship between DNS traffic and TCP flow data. In each section we present results by comparing average traffic of normal clients to the infected clients. We also present the effects of the mass-mailing worms on the outgoing traffic patterns of mail servers.

6.2.1 TCP Traffic Patterns

In this section we study the number of outgoing TCP traffic flows observed on our network.

TCP behavior of infected hosts: Figure 6.1 shows the average outgoing TCP flows (both successes and failures) for infected clients. As shown, before the worm outbreak (Day 0 - 12), the hosts made very few outgoing SMTP connections. Both the SoBig and MyDoom outbreaks (around day 12 - 15 in the graph)

[12] With the exception of clients using multiple mail client configurations to access addresses at different domains.

[13] Since we anonymize source and destination IP addresses, it is not straightforward to pinpoint infected hosts.

caused the volume of SMTP flows to jump to almost 50% of the total TCP flows. For SoBig, we also see a particularly pronounced increase in the volume of outgoing TCP flows outside of the increase in SMTP. This additional traffic can be attributed to the worm periodic ally contacting worm servers to download new malicious code. We also see a large number of SMTP failures for both worms. In SoBig's case SMTP failure s amounted to about 25% of total TCP flow traffic and approximately 40% in MyDoom's case. We speculate that MyDoom's higher failure rate is due to its attempts to "guess" the names of mail servers at particular domains, which may have resulted in resolving IP addresses protected by firewalls or otherwise unequipped to receive traffic on SMTP. In contrast, Figure 6.2 shows the average outgoing TCP flows of normal hosts during the same period. These graphs include all the outgoing TCP flows of the client and the subset of SMTP flows. The graphs indicate a cyclic weekly pattern of traffic, a higher amount of traffic during weekdays and dipping to a lower amount on the weekend. Furthermore, the uninfected client very rarely attempted to make connections to outside SMTP servers, which resulted in a minimal number of SMTP flows.

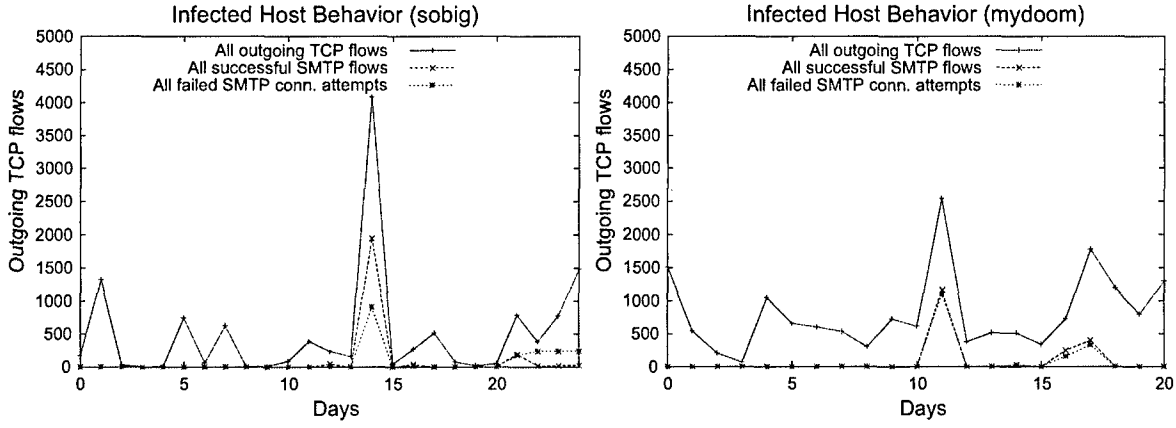


Figure 6.1: Average outgoing TCP flows for infected hosts

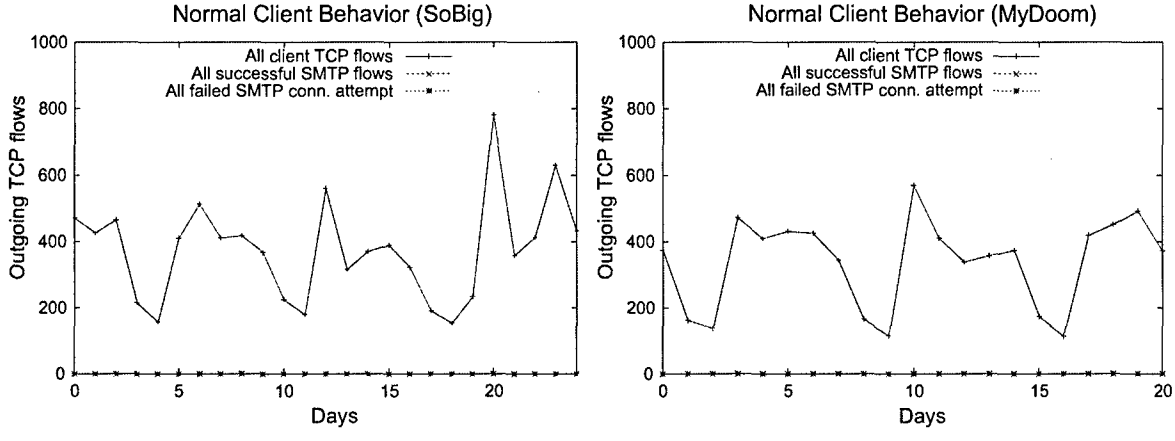


Figure 6.2: Average outgoing TCP flows for normal hosts

Behavior of mail servers: Figure 6.3 shows all the outgoing SMTP flows of the mail servers. Intuitively, since both worms carried their own SMTP engine, the outgoing traffic of mail servers should not increase. Yet, there is a pronounced spike during the SoBig outbreak whereas mail server traffic during MyDoom's outbreak remains relatively flat. We believe this is due to the 'spoofed mail' phenomenon, as described below.

A newly infected host scans the victim's hard drive for email addresses of prospective targets. The "From:" address is occasionally spoofed or hard coded in by the worm author (e.g. "admin@fake.com"). The "To:" address can be partially guessed (e.g. "John.Doe@victim.com") or it may be taken directly from a victim's address book. If the recipient does not exist on the destination mail server, that server will send an email back to the mail server listed in the "From:" address. We believe that the reason for the increase in the TCP flows is due to this phenomenon – both SoBig and My- Doom aggressively guess destination email addresses, which resulted in an increase in bounced email message flows. The data in Figure 6.3 suggests that SoBig was either more aggressive in guessing email addresses or that it more often provided return addresses in legitimate domains.[14]

Discussion: From the TCP flows statistics, we can see that SoBig employed a more aggressive propagation strategy than MyDoom. Although there were less clients infected by SoBig in our network, the total volume of flows generated by its infected clients was 50% greater than MyDoom's. Our data suggests that traffic analysis on the mail server would be ineffective against MyDoom, since mail server traffic during the MyDoom outbreak does not appear to be distinctively unusual. However, for Sobig, it is possible that one can detect its presence due to the large volume increase in the SMTP flows.

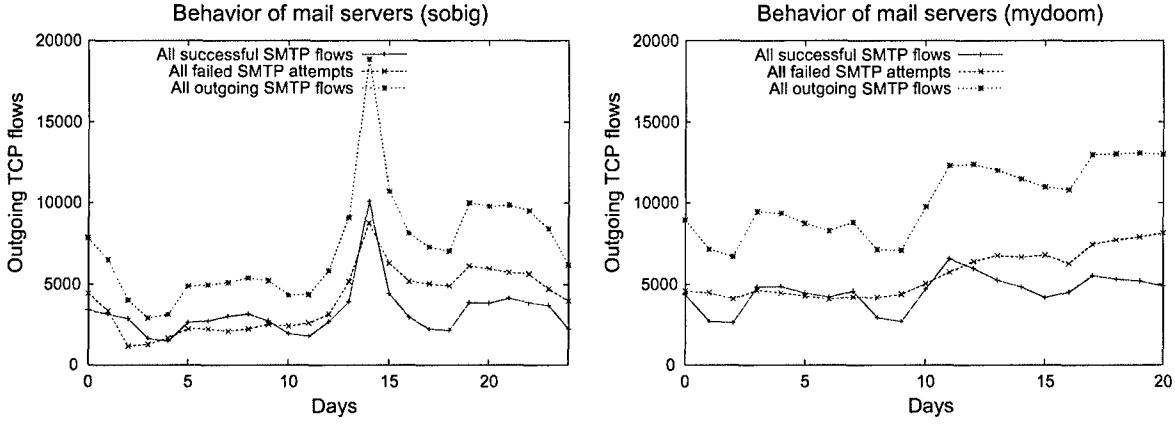


Figure 6.3: Number of outgoing SMTP flows for mail servers

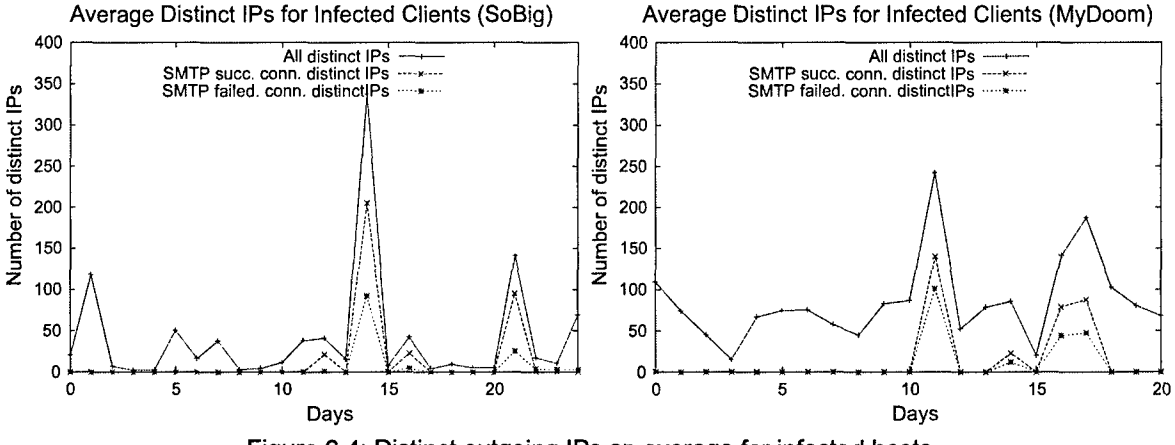


Figure 6.4: Distinct outgoing IPs on average for infected hosts

[14] MyDoom sometimes left the sender field blank or filled it with random characters.

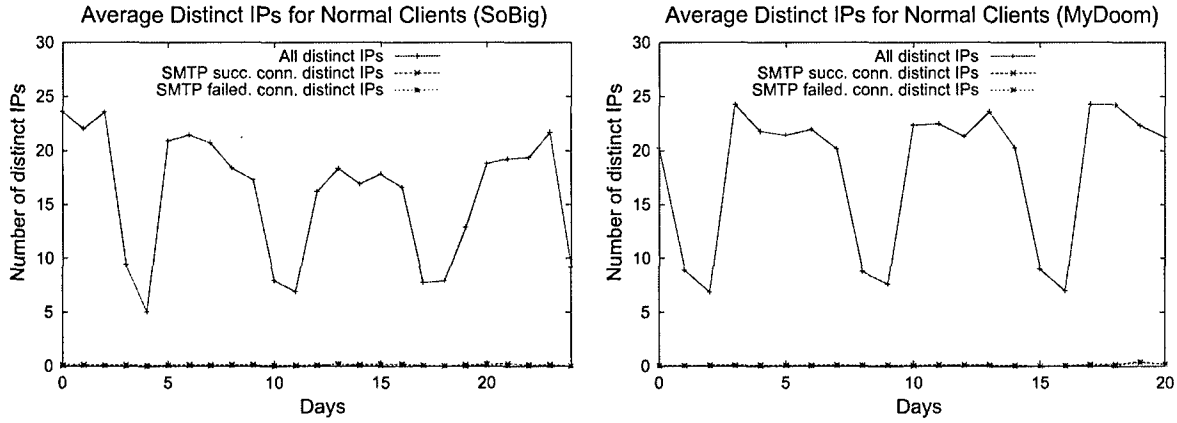


Figure 6.5: Distinct outgoing IPs on average for normal hosts

6.2.2 Distinct IPs

In this section we study the number of distinct destination IPs of outgoing TCP flows and relate this statistics to the TCP flow traffic analyzed in the previous section.

Behavior of infected hosts Figure 6.4 shows the average number of distinct destination IPs for the outgoing TCP flows of the infected hosts. Comparing the average number of successful TCP flows and the average number of distinct IPs from Figure 6.1 in the previous section, we can see that at SoBig's peak an infected client on average had 2000 successful SMTP flows corresponding to only 200 distinct IPs. At MyDoom's peak infected clients averaged 900 successful SMTP flows to 115 distinct IPs. These data suggest that multiple infection attempts were directed to the same target (perhaps different email addresses within the same domain). On average, the infected hosts sent about 10 infected emails to each server.

Plots in Figure 6.5 show that for a normal host, very few of the distinct IPs contacted involved SMTP. A typical uninfected client initiated about 400 successful TCP flows per day. Since these flows were to approximately 20 unique IP addresses, around 20 flows per IP address were observed.

Behavior of mail servers: For the mail server, the ratio of successful outgoing SMTP flows to distinct IPs contacted is smaller for mail servers compared to normal clients. At the peak of the SoBig outbreak, there were about 9500 successful outgoing SMTP flows corresponding to 1600 distinct IPs. The increase in distinct IPs contacted can be attributed to contact with additional mail servers resulting from the "spoofed mail" effect, discussed in Section 6.2.1.

Discussion: Contrary to our original belief that distinct IPs contacted should remain relatively stable during outbreaks of email worms (since domain names are scanned from web caches and files from victims machines), hosts contacted an noticeably large number of distinct mail servers. This is interesting because it means that rate limiting schemes such as Williamson's [Williamson02] that work by limiting the rate of distinct IPs contacted can still be effective. However, it should be noted that an email worm may send a large number of emails to the same mail server, hence straightforward rate limiting on distinct IPs would not be as effective as against random scanning worms.

6.2.3 DNS and Related Traffic

In this section we study DNS and related traffic. A worm attempts to infect targets by using a list of mail addresses. In order for the infection to reach its target, the worm's SMTP engine (on the infected host or on a designated mail server) needs to contact DNS servers to obtain the IP address of the destination mail server. Thus, we expect that a disproportionately large number of DNS queries will be made during the outbreak of an email worm. We conjecture that the patterns of DNS traffic during an outbreak may yield interesting insights into the behavior of an email worm. The detailed analysis can be found in the following subsections. The ECE network we study has three DNS servers. To simplify the analysis, we use the

concept of a "virtual DNS cache", an abstract entity intended to model the combined behavior of the name servers as comprising a single cache.[15]

There are a number of events in the cache that are of interest:

New cache entry. This event occurs when a previously unseen or expired IP address is returned in a DNS query. It is added to the cache along with its associated Time to Live (TTL).

Refreshed cache entry. This event occurs when an unexpired IP address is returned in a DNS query with a TTL that may or may not increase its lifetime in the cache.

Cache entry expiration. This event occurs whenever the TTL of an IP address expires.

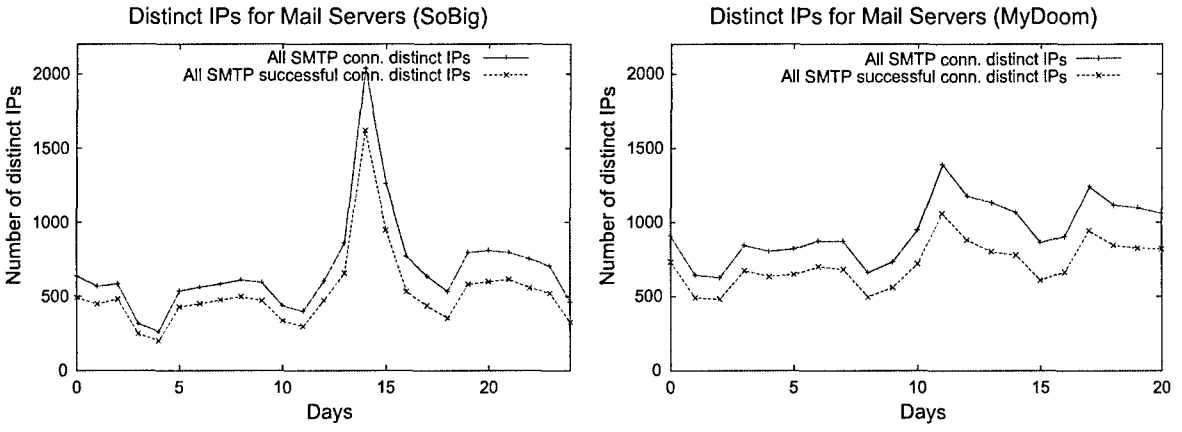


Figure 6.6: Distinct outgoing IPs for mail servers

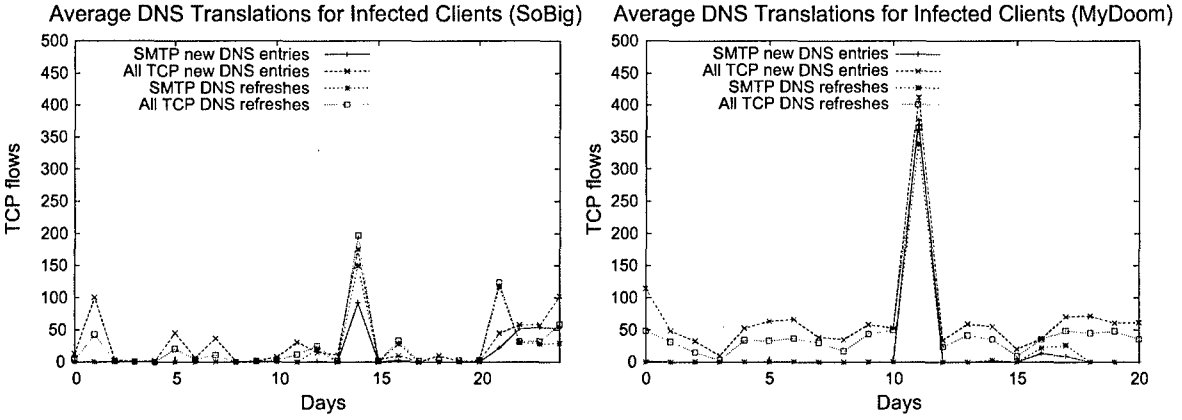


Figure 6.7: Average TCP-related and SMTP-related DNS translations for infected hosts

Modeling the cache in such a manner allows us to emulate the type of statistics and events that are already available to a nameserver internally. For the purposes of our discussion, if a host first contacts a freshly translated IP on a SMTP destination port, we refer to the translation as "SMTP-related", and if the host first contacts the IP using TCP, the translation is "TCP-related".

Behavior of infected hosts: Figure 6.7 shows the TCP-related DNS translations corresponding to new or refresh cache events for infected hosts. Figure 6.8 shows the same data for normal hosts. Before the outbreak, for both So- Big and MyDoom, SMTP-related translations were virtually non-existent. This behavior is consistent with the behavior of a normal client, as seen in Figure 6.8.

[15] In other words, all DNS traffic was combined into a single pool, regardless of the specific DNS server responsible for it.

During the outbreak of SoBig, SMTP-related translations increased dramatically to almost 50% of TCP-related translations for new entries and 80% for refreshes. We also observed that refreshes dominated new entries, which is the opposite of what was observed before the outbreak.

The effects of MyDoom are similar to those of SoBig. During the outbreak, the number of TCP-related translations resulting in new and refreshed entries amount to equal volumes of traffic, whilst before the outbreak DNS lookups resulting in new entries are significantly larger in volume than DNS refreshes. This is presumably due to the DNS lookups from different infected clients for the same mail server which resulted in DNS refreshes. Figure 6.8 shows the translation patterns for normal clients, which are consistent with the behavior of the infected clients prior to the worm outbreak.

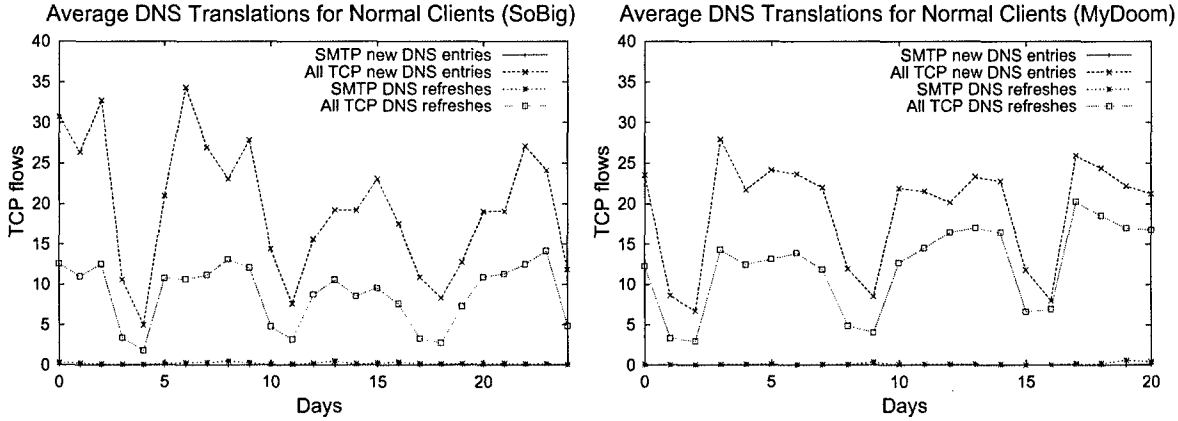


Figure 6.8: Average TCP-related and SMTP-related DNS translations for normal clients

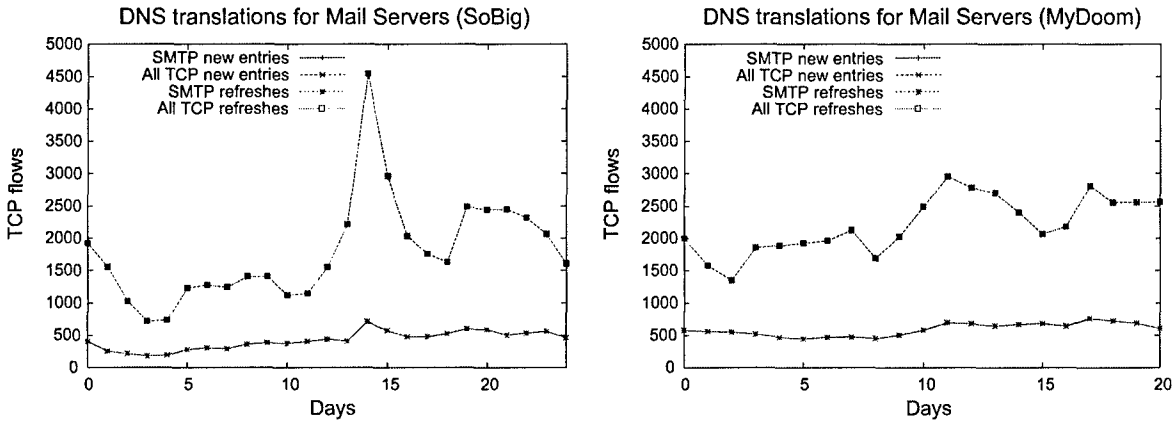


Figure 6.9: TCP-related and SMTP-related translations for mail servers

Behavior of Mail Servers: Figure 6.9 shows TCP-related and SMTP-related translations corresponding to new and refreshed DNS entries. The traffic patterns on the mail servers are quite different from what is observed on normal clients. On the mail servers, refreshes dominate the translations while the number of new entries remains consistently low. This is because there is virtually no other traffic on the mail server besides SMTP, whose destinations are mail servers with typically large TTLs. In contrast, as we will show in Section 6.2.4, client flows are dominantly HTTP.

As previously mentioned, our original belief is that there should be no increase in the outgoing TCP flows on the mail servers. Yet for the Sobig traces, there is a pronounced traffic spike occurring around day 13-15 in the graph, indicating an abnormally high volume of DNS translations. This corresponds to results seen in previous sections, where we speculate that the traffic is due to the mail server sending out an unusually high amount of bounced mails to addresses spoofed by the email worm, which resulted in the increase of DNS refreshes.

Discussion: Our study of the infected hosts shows that associating DNS translations with TCP flow data reveals a spike in DNS related traffic, and it hints at the "second-order" effects of the worm, such as additional DNS traffic generated and additional data loaded into the local name-servers' caches. We also observed that during times of infection client translations uncharacteristically resulted in more refreshes than new entries. This phenomenon could provide the basis for an interesting detection or defense strategy, such as rate-limiting DNS responses that only contain refreshed entries to clients.

What was not accounted for in our study was local DNS traffic, which may look similar to the figures in Section 6.2.1 if little address caching is performed on the end hosts. Also unaccounted for are intermediate translations that result from "walking the DNS hierarchy" to arrive at the DNS server delegated for a particular host name. If these translations are included, we could see a sizable increase in DNS traffic attributed to worm activity.

6.2.4 Overall Traffic

Given the increase in both TCP flows and DNS translations on the infected clients and mail servers, one would expect that similar trends exist in the aggregate traffic of the network. However, we found that HTTP traffic significantly dominates TCP network traffic and DNS translations. Figure 6.10 shows aggregate SMTP-related, HTTP-related, and TCP-related DNS translations in our network for the period of SoBig. Figure 6.11 shows the overall successful outgoing TCP flows in our network for the same period.[16] From inspecting new and refreshed DNS entries, we found that DNS events related to mail are rather insignificant in comparison to other traffic. As seen in Figure 6.10 there is a slight spike in SMTP-related traffic, but this comprises a very small portion of overall DNS traffic.

Moreover, we can see that HTTP makes up around 90% of outgoing TCP flows. During Sobig's infection peak, the number of successful flows attributed to SMTP was about 20,000, while HTTP amounted to about 200,000 flows. For MyDoom (graph not present) the number of SMTP flows averaged about 10,000 and HTTP amounted to on average about 300,000 flows.

By looking at overall traffic flows it is hard to meaningfully discern network behavior and the misbehavior of infected hosts. Selective traffic filtering must be employed to successfully detect the presence of a mail worm, or the noise of other traffic (e.g., HTTP) will make a worm extremely difficult to detect. In addition, to understand which hosts are making abnormal contributions to overall traffic, one must separate them into different groups (e.g., clients, servers, p2p clients, ...).

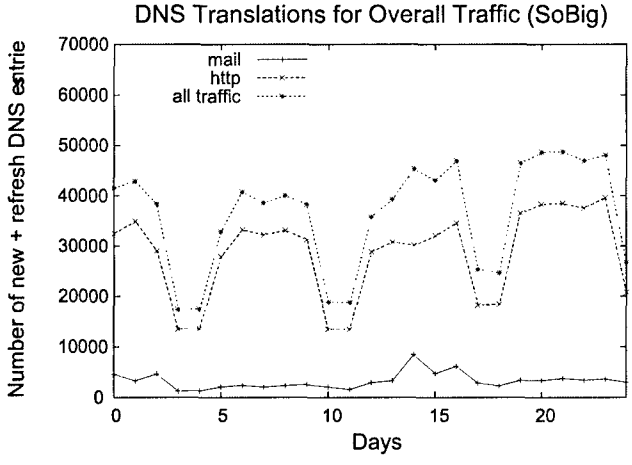


Figure 6.10: DNS effects on overall traffic.

6.3 Discussion

In the previous section we studied the effects of mass mailing worms from a variety of perspectives. We found that mass-mailing worms' activity on end host machines is apparent whether one monitors the number of outgoing TCP flows, the number of distinct IP addresses contacted by those TCP flows, or the DNS lookup pattern required to resolve the address. An advantage of monitoring DNS patterns to detect a worm is that one could implement a mail-worm watchdog at the DNS server, rather than analyzing all outgoing packets at the router. Containment possibilities at

[16] In both graphs, the data from the time of MyDoom is nearly identical to that of the time of SoBig. We only present one set due to space constraints.

the DNS server are also intriguing. In particular, the DNS server may be able to slow down the rate that mass-mailing worms propagate by delaying answers to translation requests. Another possibility is that a DNS server could attempt to "quarantine" suspicious emails by replacing the address of the requested mail server with that of a mail "holding bay". These suspicious mails could then be inspected by a more heavy weight filtering mechanism.

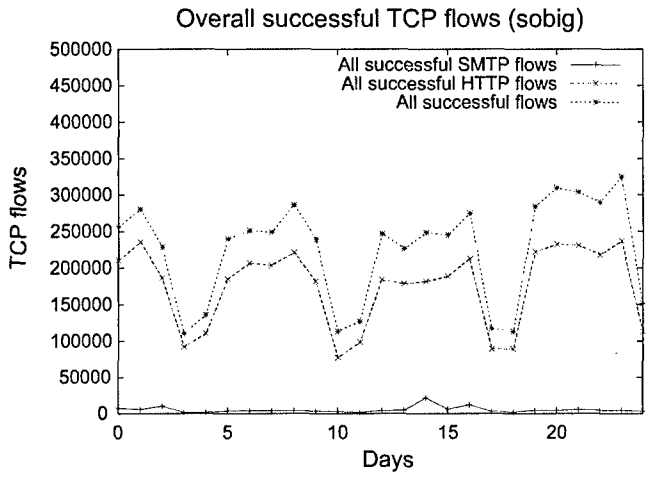


Figure 6.11: TCP flows of overall traffic.

The analysis in Section 6.2.1 shows that mail worms that create messages with a legitimate return address will place additional load on mail servers by causing noticeable "bounced" mail messages to the domains. In our analysis, we discovered that SoBig caused more "backscatter" mail traffic than MyDoom, presumably because SoBig was more aggressive in attempting to guess addresses or because MyDoom often did not provide a properly formed return address. Any assessment of the load placed on a network by a worm should factor in this "spoofed mail" effect, as well as the increase of DNS traffic attributable to it (see Figure 6.9). In order to limit the load on the mail server, one could employ a strategy limiting the number of bounced mail messages (for example, those in response to mail with attachments).

Additionally, we found that most DNS traffic generated by the worms was to refresh unexpired entries in the DNS cache rather than to input new entries. This is intuitive, because addresses associated with mail servers often have longer TTLs. This suggests that the additional DNS traffic generated by worms may be superfluous, and that loads on DNS servers may be minimized by honoring the TTLs of their current entries rather than refreshing them prematurely.

Finally, the total number of outgoing flows of overall traffic shows that DNS traffic attributed to TCP is dominated by HTTP, not SMTP. Therefore, in order to detect the presence of a mass mailing worm, it is necessary to apply additional filtering techniques to overall traffic. By examining traffic on the SMTP port or looking at sets of particular hosts or servers, we were able to distinguish the worms' presence in an academic network setting. Future work will focus on a more in depth investigation of the DNS traffic, with an eye towards separating IPs corresponding to MX (mail exchange) records from others. If this proves fruitful, it may be helpful to add logic to the DNS server that flags internal hosts requesting an abnormal number of mail server address translations as part of a wider defense or detection mechanism.

Our study suggests existing defenses specifically designed for monitoring outgoing mail on SMTP servers do not work for newer types of mass mailing worms, because almost all of these worms have their own SMTP engine. Defenses should instead be deployed where most network traffic can be seen – at the edge router or at individual hosts. In addition, selective filtering should be employed, or noise from other TCP traffic (e.g. HTTP) will obscure the traffic patterns of email worms.

Williamson's [Williamson02] and Ganger's [Ganger02] schemes will be less effective against email worms than random scanning worms. Ganger's detection scheme is based on using a secure network interface to detect and rate-limit connections to IP addresses that were not introduced to the host by a DNS translation. Since email worms require DNS lookups (for MX records) to send mail to their targets, this scheme would be ineffective.

Although the data did suggest a spike in the distinct IP addresses contacted by the infected hosts, Williamson's scheme [Williamson02] that restricts connections to different IP addresses could be easily circumvented: the worms we observed sent an average of 10 infected emails to each IP address it contacted,

and this ratio could be further increased by sending more infection attempts to a single friendly (or compromised) external SMTP server. Williamson also proposed a different scheme that would rate limit mail to distinct email addresses [Williamson03]. Other work has shown such rate limiting schemes are ineffective unless implemented on almost all host machines [Wong04a]. Another defense mechanism is for a router to filter out all outgoing SMTP traffic from non-mail servers. This solution would be possible and quite effective in an enterprise network setting where clients are mostly homogeneous. However, it may prove to be too restrictive for academic computing environments.

Future worms may attempt to hide under the radar by sending emails at a slower rate, or by intelligently delegating targeted addresses across other infected intranet hosts. We believe that an effective traffic monitoring scheme would use different filtering mechanisms and correlate their results. For instance, traffic monitoring could be done on both per-client and aggregate basis, and traffic could be monitored using a variety of metrics, including the number of total flows and unique destination IPs per each destination port. The more specifically we can examine traffic, and the more vantage points we employ to examine it, the easier it will be to mitigate false positives and detect abnormal traffic.

6.4 Quarantining Internet Worms using Rate Limiting Mechanisms

As mentioned in the previous section, one class of techniques that seems promising to defend against worm outbreaks is rate control—schemes that aim to limit the contact rate of worm traffic [Ganger02, Williamson02]. Since worms typically spread at a rapid speed from host to host, restricting the contact rate of a worm constrains how fast the infection can spread in the network. Proposals of rate control consider deploying such mechanisms primarily at the individual host level. Here, we investigate rate control at individual end hosts and at the *edge* and *backbone* routers, for both random propagation and local-preferential connection worms. Our analysis shows that both host and edge-router based rate control result in a slowdown (in the spreading rate of the worm) that is linear to the number of hosts (routers) implementing the rate limiting filter. In particular, host-based rate control has very little benefit unless rate limiting filters are universally deployed. Rate control at the backbone routers, however, is substantially more effective. Our results hold true for both random propagation worms (e.g., Code Red I) and worms that spread via a preferential connection algorithm such as those that target local hosts within a subnet.

Results are similar when dynamic immunization is taken into account. As the worm spreads and the knowledge of the worm disseminates, an increasing number of hosts (both infected and susceptible) will be patched, immunized and consequently removed from the susceptible population. In an effort to study realistic worm attacks, the models discussed here incorporate dynamically changing the immunization rates. This is in contrast to the traditional models for which the rate of immunization remains constant throughout the infection outbreak [Kephart93, Wang03a, ChenZ03, Kephart91, Wang03b].

To provide context for the models, we examine traffic traces obtained from a sizable campus computing network. We observe that limiting the rate of unique IP addresses contacted (as in [Williamson02]) from the edge of the departmental network to no more than 16 (total contacts) per five-second period would almost never affect legitimate traffic. Individual host rates can be kept to under four per five-second period. Limiting only non-DNS-translated IP address contacts [Ganger02] can reduce the contact rate by another factor of 2 – 4. Our traces also captured the behavior of machines infected by two worms: Welchia and Blaster. The results confirm that infected machines exhibit much higher contact rates and could be dramatically slowed by rate limiting.

Combining practical rate limits with our models allows us to estimate how well such approaches might work in practice. For instance, to secure an enterprise network from worms that propagate using a local-preferential connection algorithm, our study shows that unless rate limiting filters are deployed at both the edge routers and a certain percentage of the end hosts, little benefit will be gained.

6.4.1 Background—Epidemiological Models

One class of epidemiological models, homogeneous models, is widely used in the studies of human infections. A homogeneous model assumes homogeneous mixing among the individuals in the population [Bailey75, McKendrick26]; that is, every individual has equal contact to every one else in the population. This assumption is similar to the ways in which random propagation worms spread in computer networks. This model is described in more detail in [Bailey75]. A homogeneous model assumes a connected network with N nodes. It also assumes an average infection rate β across all links. If we represent total number of infected nodes at time t as It, a deterministic time evolution of I (infected hosts) can be obtained as below,

$$\frac{dI_t}{dt} = \beta I_t (N - I_t / N) \tag{6.1}$$

The solution to Equation (6.1) is $I/N = e^{\beta t} / c+e^{\beta t}$, where c is a constant. c is determined by the initial infection level. $c \rightarrow N - 1$ when the initial infection level is low, since the fraction of infected hosts will be small. From this we can see that the infection grows exponentially initially and reaches saturation after a certain point. The time takes to reach a certain infection level α is

$$t = l\eta\alpha / \beta \tag{6.2}$$

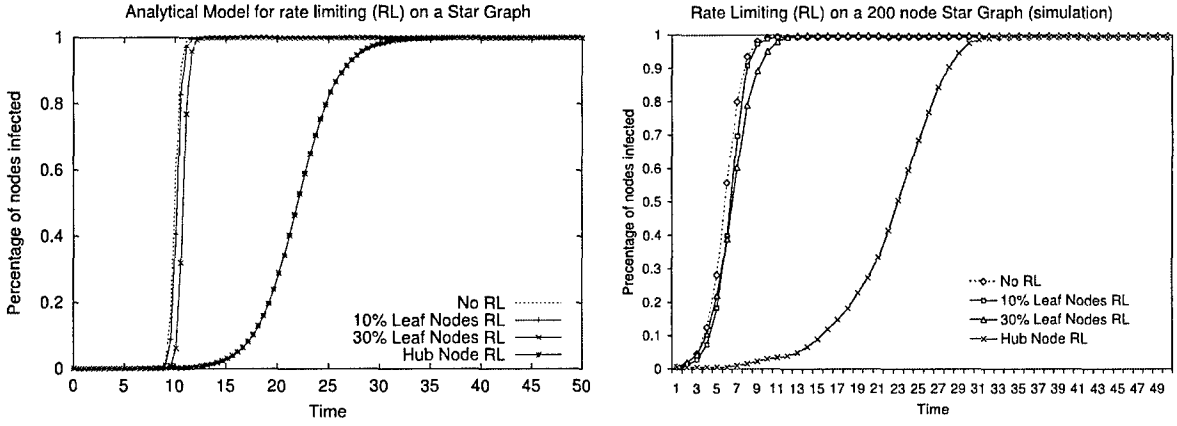
The analytical models described in the later parts of this chapter are derived from the basic homogeneous model and share the same assumptions.

6.5 Rate Limiting

In this section, we present a study on rate limiting mechanisms as a defense to combat the propagation of Internet worms. Rate limiting is a mechanism by which an element in the network can restrict the rate of communication with other elements. Since worms spread rapidly via fast connections to uninfected machines, rate limiting can help suppress the propagation of the worm. A number of rate limiting schemes have been proposed in the literature, including Williamson's virus throttle [Williamson02] and Ganger's DNS based scheme [Ganger02]. However, it is not known precisely how and where rate control mechanisms should be deployed in a network. Clearly, instrumenting rate control on every individual node in a network is expensive administratively and hence not feasible. The question then becomes: are there alternative deployment strategies that can yield a more desirable effect than others?

We believe that the answer to this question is yes. In this section we illustrate the effect of different deployment strategies using a star graph topology. Consider a star graph where a central hub node is connected to all the leaf nodes. We analyze two deployment scenarios: a) rate control at a certain percentage of the leaf nodes, and b) rate control at the center hub node only. Note that a star topology is very different from the Internet's topology and the study of a star topology is mainly for demonstration of the difference from deployment at leaf and hub nodes.

Deployment at leaf nodes: Assume we deploy rate limiting filters at q percent of the leaf nodes. Let $x_1 = I(1-q)$ be the number of infected nodes that are not confined by the filtering mechanism, $x_2 = Iq$ the number of infected nodes with the filter mechanism, β_1 the contact rate of the infected host without the filter, and β_2 the contact rate allowed by the filter, with $\beta_1 >> \beta_2$.



a) analytical model of infection evolution

b) simulated infection evolution

Figure 6.12. Plots showing the differences between various rate-limiting deployment mechanisms on a 200-node star topology.

We obtain the time evolution of the infection as below,

$$\frac{dI}{dt} = x_1\beta_1(N-I)/N + x_2\beta_2(N-I)/N \tag{6.3}$$

Solving Equation (6.3) gives us $I/N = e^{\lambda t} / c+e^{\lambda t}$, where $\lambda = q\beta_2 + (1 - q)\beta_1$. When $\beta_1 >> \beta_2$ and $e\lambda t$ is small, $\lambda = \beta_1 (1 - q)$. From this, we can derive that the time t to reach a certain infection level α is $t = ln\alpha/(\beta_1 (1 - q))$. That is, the rate of infection is proportional to $1-q$, the percentage of nodes that do not have rate limiting filters. Comparing to Equation (6.2), we can see that deploying rate limiting filters at the leaf nodes yields a linear slowdown that is proportional to the number of nodes that have rate control.

Deployment at hub: When deploying rate control at the center hub node, we need to consider both node-level and link-level rate limiting. Assume we deploy rate limiting at the hub node with rate β and link rate limiting with rate γ. When $\beta \geq \gamma I$ — the contact rate at the hub node is greater than the combined contact rate of infected leaf nodes — then γI is the primary limiting factor for infection propagation. Otherwise, the propagation is limited by the hub node contact rate, β.

For link-level rate limiting (this is when the contact rate at the hub node is higher than the combined contact rates of all the infected leaf nodes), we have

$$\frac{dI}{dt} = \gamma I(N-I)/N, \text{when } \gamma I \leq \beta \tag{6.4}$$

Solving Equation (6.4) gives us

$$I/N = \frac{e^{\gamma t}}{c+e^{\gamma t}}, \text{when } \gamma I \leq \beta$$

For node rate limiting (this is when the combined contact rates of the infected leaf nodes exceeds the hub node contact rate), we have

$$\frac{dI}{dt} = \beta(N-I)/N, \text{when } \gamma I > \beta \tag{6.5}$$

Solving Equation (6.5) gives us

$I/N = 1 - ce^{-\beta t/N}$, when $\gamma I > \beta$

From the solution to Equation (6.4), we can derive that the time t to reach an infection level α is $t = N(ln(\alpha))/\beta$. Compared to rate control at the leaf nodes, this suggests a slowdown that is comparable to installing rate control filters at all of the leaf nodes — in which case $t = ln(\alpha)/\beta_2$. Indeed, the graph in Figure

6.12(*a*), which plots both leaf-node and hub-node rate control on a 200-node star topology, indicates exactly that. Figure 6.12(*b*) shows simulated propagations on the same topology.

In our simulation, we limited the links to 10 packets per second with the hub rate limit $\beta = 0.01$. The simulation results are an average of ten simulation runs. For leaf-node rate control, we simulated rate limiting at 10% and 30% of the leaf nodes. As shown in Figure 6.12(*b*), rate limiting at 10% of the leaf nodes has neligible impact. Rate limiting at 30% of the leaf nodes results in a slight slowdown of the infection rate. Rate control at the hub node is significantly more effective. For instance, reaching a level of 60% infection with rate limiting at 30% of the leaf nodes is approximately three times quicker than rate limiting at the hub. These results confirm our analytical model.

This simple but illustrative example shows that deployment strategies have a significant impact on the effectiveness of rate control schemes. On the Internet, we can deploy rate control at end hosts, edge routers, and backbone routers. In the next section we investigate each of these deployment cases.

6.6 Deploying Rate Control on the Internet

In this section we investigate three different ways of deploying rate limiting schemes on the Internet: on individual hosts, edge-routers, and at backbone routers. We develop a mathematical model to reason about each deployment strategy's effectiveness, and conduct simulation experiments to confirm the model's predictions.

6.6.1 Host-based Rate Limiting

Deploying rate limiting filters at individual hosts is similar to rate limiting at the leaf nodes of a star topology as described in Section 6.5. Again, let q be the percentage of nodes that install the filter mechanism. $x_1 = I(1-q)$ is the number of infected nodes that are not confined by the filter mechanism, and $x_2 = Iq$ is the number of infected nodes with the filter mechanism. β_1 is the contact rate of the infected host without the filter, β_2 is the contact rate allowed by the filter, and $\beta_1 >> \beta_2$.

Similarly, we can use Equation (6.3) to model the time evolution of infection. The solution to Equation (6.3) gives us

$I/N = e\lambda t\, C + e\lambda t$, where $\lambda = q\beta_2 + (1 - q)\beta_1$

When $\beta_1 >> \beta_2$, $\lambda = \beta_1 (1 - q)$. The analysis in Section 6.5 on rate limiting on leaf nodes also holds here. Figure 6.13 shows the time evolution of I with $\beta_1 = 0.8$ and $\beta_2 = 0.01$. As we see in Figure 2, the deployment of host-based confinement mechanisms yields a linear slowdown in the infection rate of the worm. Note the difference between 80% deployment and 100% deployment of rate limiting, this shows that rate limiting has very little benefit unless all end hosts implement rate limiting.

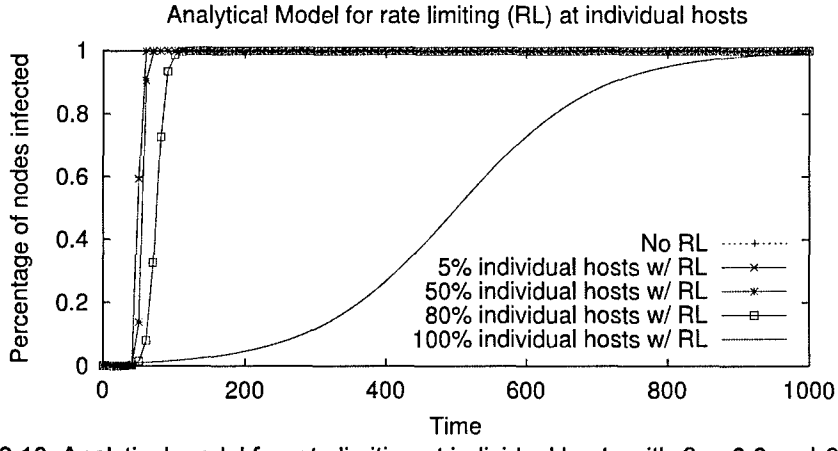


Figure 6.13. Analytical model for rate limiting at individual hosts with $\beta_1 = 0.8$ and $\beta_2 = 0.01$

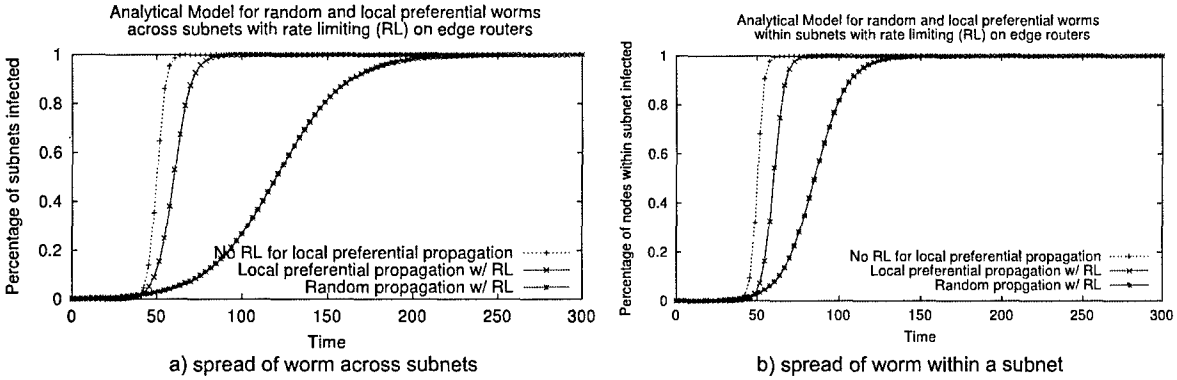
6.6.2 Rate Limiting at Edge Routers

Edge-router based deployment is similar to the host-based rate limiting scheme. From the set of networks that install the filter, we can calculate the effective q (percentage of nodes that install the filter mechanism) and the rest of the calculation is the same.

When filters are installed at edge routers, worms propagate much faster within the subnet than across the Internet. We denote the contact rate within the subnet as β_1 and the contact rate across the Internet as β_2. Clearly, $\beta_1 \geq \beta_2$. For a random propagation worm, the infection growth within the subnet has the form $x = e^{\beta_1 t}/C_1 + e^{\beta_1 t}$, where x is the number of infected nodes within a particular subnet. The number of subnets infected has a similar growth form $y = e^{\beta_2 t}/C_2 + e^{\beta_2 t}$, where y is the number of infected subnets.

For worms that use a preferential targeting algorithm (i.e., those that target nodes within the same subnet), the growth formula stays the same except for that the infection rate within the subnet, $\beta 1$, could be substantially larger than that of a random propagating worm. Consequently, the effectiveness of rate control at edge routers diminishes when a worm employs an intelligent targeting algorithm such as subnet preferential selection.

Figure 6.14 depicts the analytical models for both local preferential connection and random propagation worms with rate limiting filters at the edge routers. It shows the time evolution of the percentage of hosts infected with $\beta_1 = 0.8$ and $\beta_2 = 0.01$. In the base case with no rate limiting, the infection grows exponentially before it reaches its maximum limit. With rate control there is a slight slowdown in the rate of infection. As shown in Figure 6.14(a), our model indicates that edge router rate limiting is more effective for the random propagation model. To verify this, we created simulations to compare edge router rate limiting for both local preferential and random propagation models. The results of the simulations are shown in Section 6.6.4.



a) spread of worm across subnets

b) spread of worm within a subnet

Figure 6.14. Analytical models for random and local preferential worms

6.6.3 Rate Limiting at Backbone Routers

In this section we investigate rate limiting at the backbone routers of the Internet. In order for a worm to propagate from one network to another, the worm packets need to go through backbone routers on the Internet. Therefore, deploying rate limiting mechanisms at the backbone routers can help throttling worm propagation. We perform an approximate analysis of rate limiting at backbone routers below.

If we deploy the rate limiting mechanism on the core routers that cover α percent of the total IP-to-IP paths, then

$$\frac{dI}{dt} = I\beta(1-\alpha)(N-1)/N + \delta(N-I)/N, \qquad (6.6)$$

where β is the contact rate of one infected host, $\delta = \min(I\beta\alpha, rN/2^{32})$, and r is the average overall allowable rate of the routers with the rate limiting control. When r is relatively small, the right hand side of

Equation (6.6) can be approximated by only the first term. We can thus obtain $I/N = e^{\lambda t} / c+e^{\lambda t}$, where $\lambda = \beta(1 - \alpha)$ and c is a constant.

6.6.4 Simulation Results

The simulator used to conduct our experiments is built on top of Network Simulator (ns-2) [Fall02]. All experiments in this section are conducted using a 1,000 node power-law graph generated by BRITE [Medina01]. The graph shares similar characteristics to an AS topology such as the Oregon router views. Unless specified otherwise, each simulation is averaged over 10 individual runs. In addition, the time units in all our simulations are simulation ticks as defined by ns-2.

We begin each simulation with a random set of initial infections. At each time unit each infected node will attempt to infect everyone else with infection probability β. The infection packet is routed using a shortest path algorithm through the network. Links that have the rate limiting mechanism will only route packets at a rate of γ.

In order to experiment with the different deployment cases, we designate the top 5% and 10% of nodes with the most number of connections as backbone and edge routers respectively. The remaining nodes are end hosts. Rate limiting is implemented by restricting the maximal number of packets each link can route at each time tick and queuing the remaining packets. In order to ensure that normal traffic gets routed, we assign each rate-controlled link a base communication rate of 10 packets per second. We then compute a link weight that is proportional to the number of routing table entries the link occupies. We multiply this weight to the base rate to obtain the actual link rate simulated for each link. We believe that this simulated routing will allow most normal traffic to be routed through since the most utilized links will have a higher throughput.

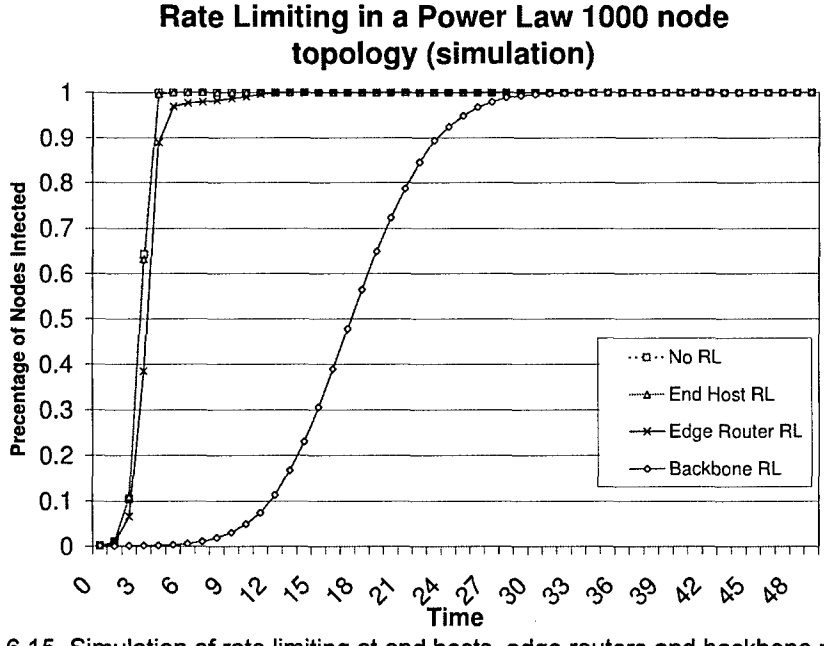


Figure 6.15. Simulation of rate limiting at end hosts, edge routers and backbone routers.

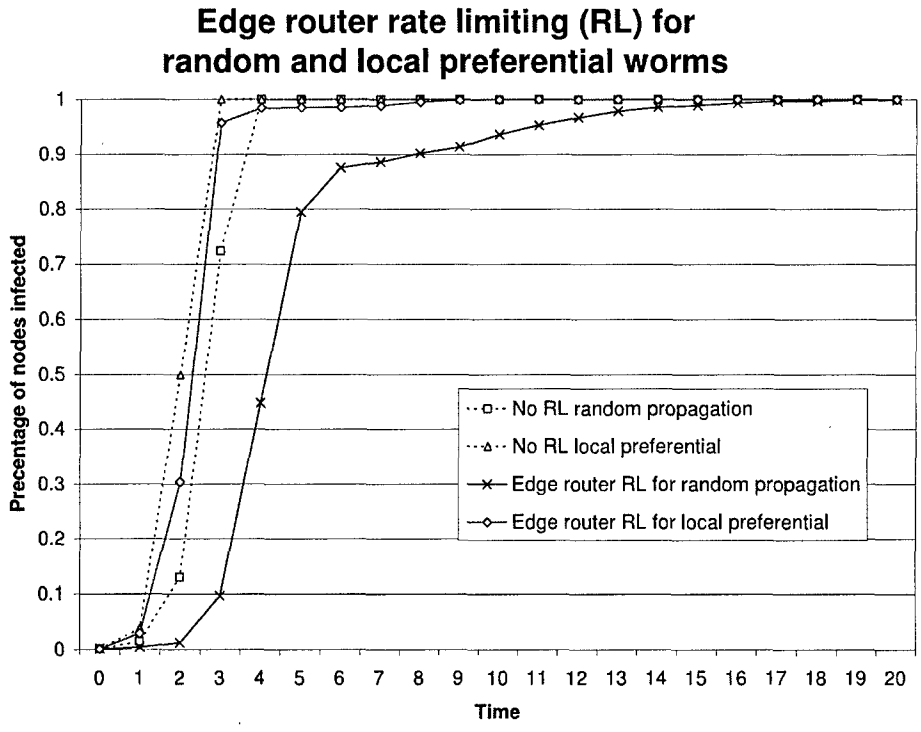


Figure 6.16. Simulation and comparison of rate limiting within subnets at the edge router for local preferential and random propagation worms.

Figure 6.15 shows the simulation results for random propagation worms, for the cases of no rate limiting, rate limiting at 5% of the end hosts, edge routers and backbone routers. As shown, the simulation results confirm our analytical models in Sections 6.6.1 and 6.6.2. More specifically, there is negligible difference between no rate limiting and rate limiting at 5% of end hosts. While rate limiting at the edge routers shows a slight improvement, rate limiting at the backbone routers renders a substantial improvement. Compared to the case of end host and edge router based rate limiting, it takes approximately five times as long for the worm to spread to 50% of all susceptible hosts if rate limiting is implemented at the backbone routers.

Figure 6.16 shows the simulated propagations for rate limiting at the edge router for both local preferential and random propagation worms within subnets. The dotted lines are the base cases (with no rate limiting) for local preferential and random worms respectively. As our simulations show, there is very little perceivable benefits for implementing rate limiting at the edge routers if worms propagate using a local preferential algorithm. For random propagation worms, however, rate control at the edge routers still yields a 50% slowdown. Clearly, edge router based rate limiting is more effective in suppressing random propagation worms as opposed to worms that propagate via local preferential connections. These results also confirm our analytical model described in Section 6.6.2. Figure 6.17 shows the simulated propagation for local preferential worms for both host and backbone-router-based rate limiting across subnets. As shown, even with a 30% deployment of rate limiting mechanisms at the end hosts there is negligible difference when compared to no rate limiting. Deploying rate limiting filters on the backbone routers, as shown in Figure 6.17, is substantially more effective.

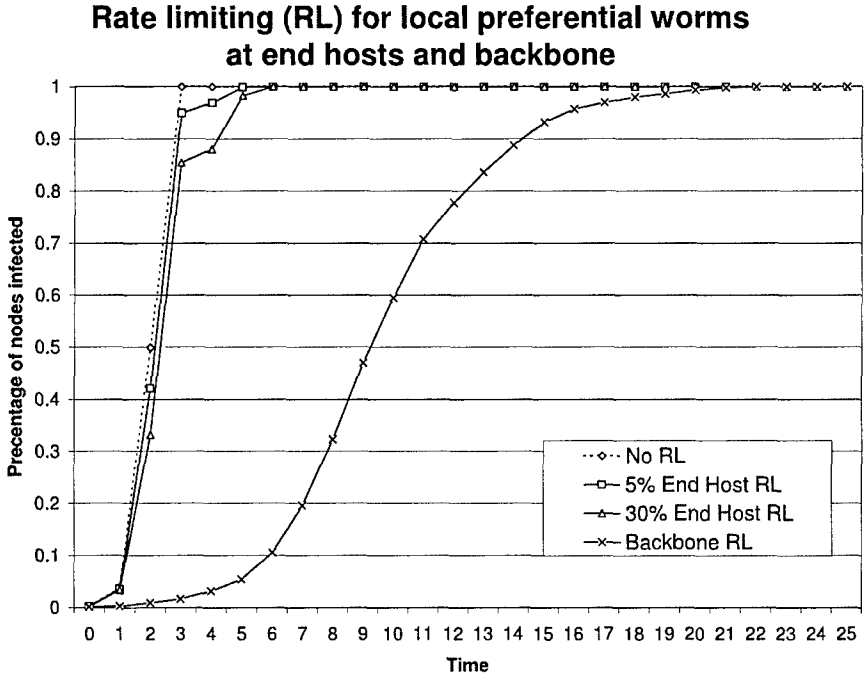


Figure 6.17. Simulation of rate limiting across subnets for local preferential worms at the end hosts and backbone routers.

6.7 The Effect of Dynamic Immunization

Thus far in our analysis we have ignored the effect of patching and dynamic immunization. Clearly, the infection progress will be hampered when the exploited vulnerabilities are patched (immunized) dynamically. In this section, we examine dynamic immunization and its effect on rate limiting.

6.7.1 Delayed Immunization

The immunization model we consider here assumes that the immunization process starts at time d, after a certain percentage of hosts are infected. Thereafter, in each time interval, each susceptible host will be patched with probability μ. The following differential equations model the dynamics of the worm propagation in the presence of immunization:

$$\frac{dI}{dt} = I\beta N - IN, \text{when } t \le d$$

$$\frac{dI}{dt} = I\beta N - IN - I\mu, \text{when } t > d$$

$$\frac{dN}{dt} = -\mu N, \text{when } t > d$$

Solving the above equations gives us

$$\frac{I}{N_0} = \frac{e^{\beta t}}{c + e^{\beta t}}, \text{when } t \le d$$

$$\frac{I}{N_0} = \frac{e^{(\beta-\mu)(t-d)}}{c_0 + e^{\beta(t-d)}}, \text{when } t > d$$

where N_0 denotes the initial number of susceptible hosts.

Figure 6.18(a) shows a plot of the equations. We also conducted simulations of delayed immunization on a synthetic 1000-node power-law graph with d = 20%, 50%, and 80% of the nodes infected with $\beta = 0.8$ and $\mu = 0.1$. The results are in Figure 6.19(a). Clearly, the earlier immunization takes place, the more effective it is. In Figure 6.19(a), immunization starting at 20% infection yielded a total infected population of 80% of the nodes, as opposed to 90% infected when immunizing at 50% and 98% infected at 80%.

We note that the assumption of a constant probability of immunization, denoted by μ, is not completely realistic. In reality, the probability of immunization may increase as the worm spreads and as the vulnerability it exploits becomes widely publicized. Similarly, the probability of immunization may decrease as the infection becomes a rarer occurrence, i.e., on its way to extinction. We believe that the rate of immunization observes a bell curve. However, the exact shape of such a curve is not easily obtainable and we lack data to confirm the rate of immunization. Therefore, we use the simple assumption of immunization at a constant rate.

6.7.2 Rate Control with Dynamic Immunization

In this section we examine the effect of delayed immunization with rate limiting. We focus on the case of rate limiting at backbone routers (since that is the most effective strategy according to the analysis in Section 6.6). Assuming the first instance of immunization occurs at time d, the growth of the infection with delayed immunization and rate control at the backbone routers is as follows:

$$\frac{dI}{dt} = I\beta(1-\alpha)(N-I)/N + \delta(N-I)/N, \text{ when } t \le d$$

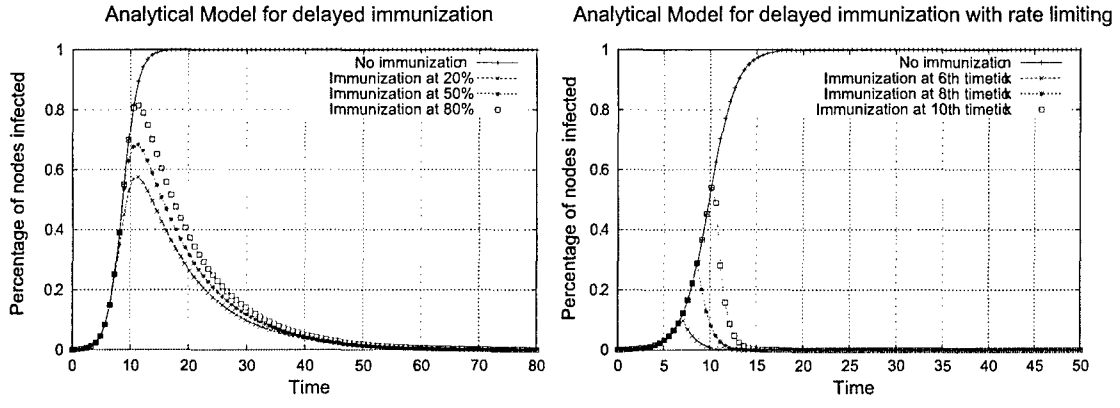
$$\frac{dI}{dt} = I\beta(1-\alpha)(N-I)/N + \delta(N-I)/N - \mu I, \text{ when } t > d$$

$$\frac{dI}{dt} = -\mu N, \text{ when } t > d$$

where β is the contact rate of one infected host, $\delta = \min(I\beta\alpha, rN/2^{32})$, r is the average overall allowable rate of the routers with rate limiting control and α is the percentage of paths that have rate limited links. When r is relatively small, the solution can be approximated as

$$\frac{I}{N_0} = \frac{e^{\gamma t}}{c + e^{\gamma t}}, \text{when } t \le d$$

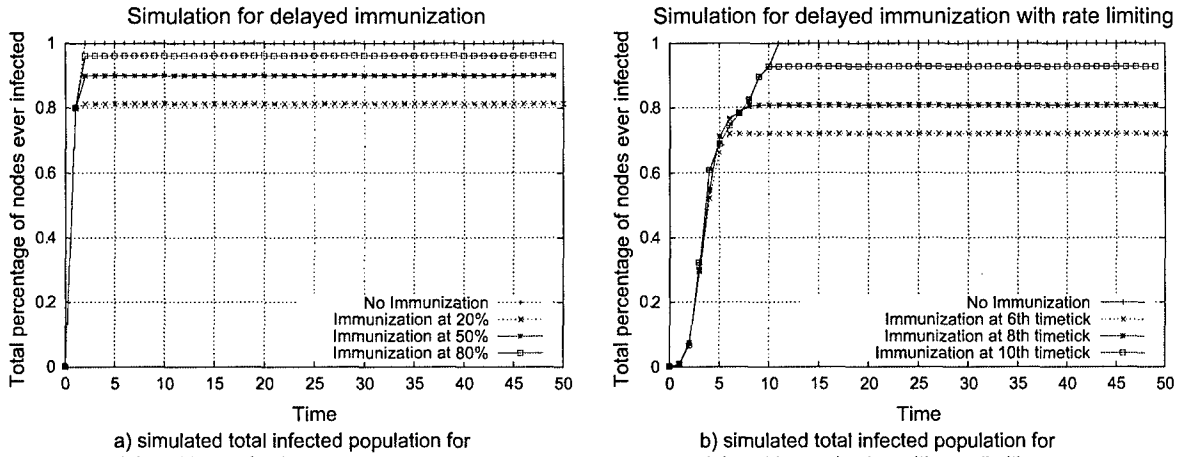
$$\frac{I}{N_0} = \frac{e^{(\gamma-\mu)(t-d)}}{c_0 + e^{\gamma(t-d)}}, \text{when } t \le d, \text{where } \gamma = \beta(1\text{-}\alpha)$$



a) analytical model of infection evolution for delayed immunization

b) analytical model of infection evolution for delayed immuniation with rate limiting

Figure 6.18. Analytical Models (with and without rate limiting) for delayed immunization on a 1000-node power-law graph



a) simulated total infected population for delayed immunization

b) simulated total infected population for delayed immunization with rate limiting

Figure 6.19. Simulations of delayed immunization (with and without rate limiting) on a 1000-node power-law graph

We also conducted simulations of delayed immunization with rate limiting on a synthetic 1000-node power-law graph with $\beta = 0.8$ and $\mu = 0.1$. Since the goal here is to identify the benefits of rate limiting, the timeticks chosen in both analytical and simulation are the timeticks at which immunization started in our analytical model for delayed immunization without rate limiting (e.g. for immunization starting at 20%, our analytical model shows that it should happen around the 6th timetick). Figure 6.18(b) plots the analytical model and Figure 6.19(b) plots the simulated results of delayed immunization with rate limiting. The plots show exactly how immunization delays in combination with rate limiting at the backbone affects the infection propagation. Recall that in Figure 6.19(a), immunization at 20% with no rate limiting results in a total infected population of 80%. Figure 6.19(b) shows a similar experiment with the same simulation parameters but with rate limiting, which results in a total infected population of 72%, a 10% drop from the case where no rate limiting was implemented. The same results hold for other values of delay period d. In summary, rate limiting helps to slow down the spread and as a result buys time for system administrators to patch their systems and ultimately minimize the damage of worm outbreaks.

6.8 Rate Limiting in Practice

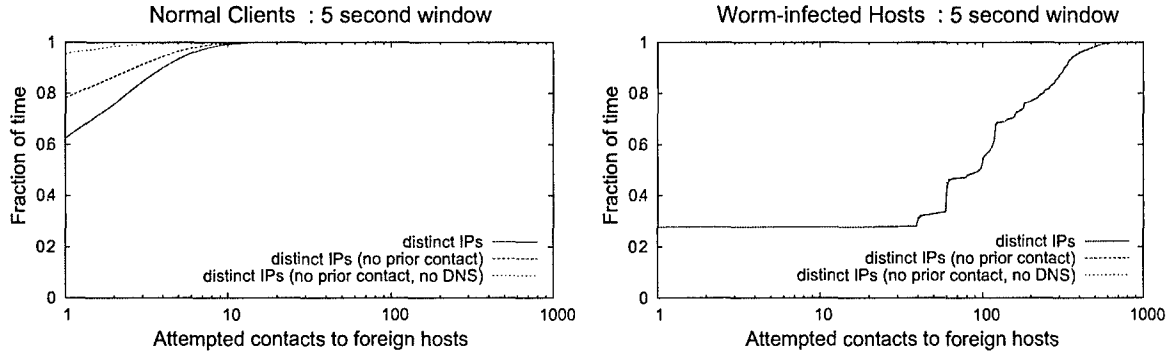
This section presents an analysis of real network traces, with a goal of identifying rates at which connections can be throttled in practice. Specifically, we wish to identify rate limits that an enterprise network can realistically implement that will significantly slow worms while having minimal impact on legitimate communications. Using these rates in our models produces a corresponding propagation prediction that might be viable in practice.

We focus on two recently proposed techniques for rate limiting. The first, proposed by Williamson [Williamson02], restricts the number of unique IP addresses with which a host communicates in a given period of time; the default discussed in that paper was five per second (per host). The second, proposed by Ganger et al. [Ganger02], restricts the number of unknown IP addresses (those without valid DNS cache entries and that did not initiate contact) to which a host can initiate connections in a given period of time; the default discussed was six per minute (per host). The second technique focuses on the common approach used by self-propagating worms to identify target hosts: picking pseudo-random 32- bit values to use as an IP address (thus performing no DNS translation).

We evaluated these techniques using a 23-day trace from the edge router for CMU's Electrical and Computer Engineering (ECE) Department. The traces recorded in an anonymized form all IP and common second layer headers of traffic (e.g., TCP or UDP) entering or exiting the ECE network from August 15th until September 7th, 2003. The contents of all DNS packets were recorded and anonymized. In addition to

the regular activity of the department, this period includes two major worm outbreaks: Blaster [CERT03] and Welchia.

Through examining the traces, we were able to partition the ECE subnet (1128 hosts total) into four types of hosts: normal "desktop" clients, servers, clients running peer-topeer applications, and systems infected by worms. Each type of hosts exhibited significantly different connectivity characteristics. The 999 "Normal Clients" exhibited traffic patterns driven by client-server communication, such as HTTP, AFS, and FTP traffic. 17 "Servers" provide network services, such as SMTP, DNS, or IMAP / POP. The 33 clients running peer-to-peer applications (in these traces Kazaa, Gnutella, Bittorrent, and edonkey) were placed in their own category because they exhibit greater connectivity than normal hosts. This can be attributed to the nature of peer-to-peer systems; packets must be exchanged periodically in order to establish which hosts are on the network and the content they serve. Finally, 79 systems were observed to have been infected by the Blaster and/or Welchia worms. Both these worms exploited the Windows DCOM RPC vulnerability. Blaster scanned subnets for other vulnerable hosts by attempting to send itself to TCP destination port 135. Welchia was a "patching" worm which first scanned subnets for vulnerable hosts using ICMP ping requests. If a host replied, Welchia attempted to infect the system, make further attempts to propagate, patch the vulnerability, and reboot the host. We were able to differentiate between the two worms by looking for a large amount of ICMP echo requests intermixed with TCP SYNs to port 135. We found that although Welchia's intention was benign, its peak scanning rate was an order of magnitude greater than Blaster's.[17]



a) CDF of aggregate contact rates for 999 "normal desktop" clients. Note how the contact classification refinements result in lower values.

b) CDF of aggregate contact rates for 79 clients infected by the Blaster and/or Welchia worm.

Figure 6.20. CDF of Contact rates in a five second interval for normal and infected clients

Figure 6.20 shows the observed aggregate contact rates for (a) normal clients and (b) worm-infected clients. As shown, they are very different. In addition to the solid lines, which indicate the number of distinct IP addresses contacted in a 5-second period, two other lines are given to indicate the effect of possible refinements on rate limiting. The dashed line shows the number of distinct IP addresses contacted from within the network. The dotted line shows the number of distinct IP addresses contacted from within the network and not counting those for which valid DNS translations are obtained. Clearly, these refinements may be useful in limiting contact rates to lower numbers while having less impact on legitimate communications. For instance, to avoid having impact 99.9% of the time, inside-to-outside contact rate could be limited to 16 per five seconds for all contacts at the edge router, 14 per five seconds for contacts to hosts that did not initiate contact first, or 9 per five seconds for contacts to hosts for which a valid DNS translation did not exist or did not initiate contact first. The tightness of the three lines in the worm-infected graph support this statement, showing that worms traffic spike all three metrics.

[17] We discovered an instance of Welchia that scanned 7068 hosts in a minute. By contrast, Blaster's peak scanning rate was only 671 hosts in a minute. Blaster, however, was much more persistent in its propagation attempts.

The P2P and server systems are less well-behaved than normal systems and less ill-behaved (in terms of contact rate) than worm-infected systems. But, the contact rate limits would have to be greatly increased in order to avoid impacting regular traffic. Specifically for P2P clients, the network could be limited to 89 per five seconds for all contacts, 61 per five seconds for contacts to hosts that did not initiate contact first, or 26 per five seconds for contacts to hosts for which a valid DNS translation did not exist or did not initiate contact first. Alternately, an administrator could categorize systems as we have done, and give them distinct rate limits. This would tightly restrict most systems (those not pre-determined to be special), while allowing special others to contact at higher rates. Of course, performance penalties will be faced by new P2P users, until they convince the security administrator to deem them "special". Many administrators would prefer this model to the unconstrained load spikes that they currently face, and have to diagnose, as new P2P applications are introduced to their environment.

Rather than aggregate limits at the edge routers, as discussed above, another way to limit contact rates is per individual host (e.g., in host network stacks [Williamson02] on smart network cards or switches [Ganger02]). Our analysis of the traces indicates that the resulting restrictions can safely limit a single "normal desktop" system initiating contact to, for example, four unique IP addresses per five seconds or one unique non-DNS-translated IP addresses per five seconds. Although these numbers are lower, however, the 1128 machines in the network could conceivably each use their full slot when a worm infects them, meaning that the aggregate contact rate from the intra-net would be much higher than the rate limits discussed for the edge router case. This suggests that per-host rate limits are a poor way to protect the external Internet from internal worm traffic.

Per-host limits, however, are a much better at (in fact, the only way) to protecting the internal network [Ganger02] once worms get past the outer firewall.

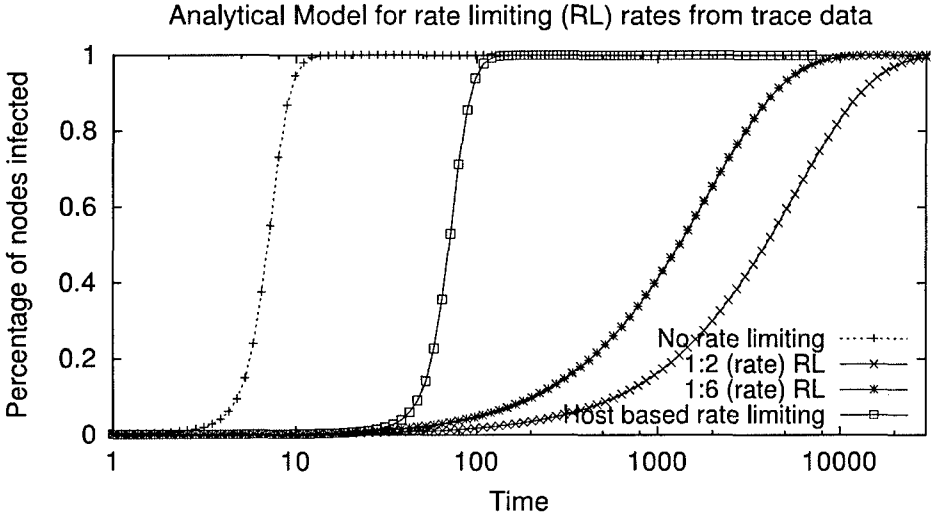


Figure 6.21. Effect of rate limiting given the rates proposed by our trace study.

A final observation from the traces relates to the choice of a rate limit window size. We observed that longer windows accommodate lower long-term rate limits, because heavy-contact rates tend to be bursty. For example, for aggregate non-DNS rates, 99.9% of the values are five for one second, twelve for five seconds, and fifty for sixty seconds. The downside to a long window, however, is that one could face a lengthy delay after filling it, before the next connection is allowed. Visible disruptions of this sort may make long windows untenable in practice. One option worth exploring is hybrid windows with, for example, one short window to prevent long delays and one longer window to provide better rate-limiting. Figure 6.21 illustrates the effect of different rate limits on worm propagation. We approximate Williamson's IP throttling scheme and Ganger's DNS-based scheme using Equations (6.4) and (6.5) in Section 6.5. Although Equations (6.4) and (6.5) model deployment-at-hub, they can be used to approximate edge router rate limiting in the case of a single subnet. Recall β is the aggregated node contact rate while γ is the con-

tact rate per link. As the traces indicated a lower aggregated rate for the DNS-based scheme, we choose the ratio of γ to β as 1:2 to represent the DNS-based scheme and the ratio of 1:6 for the IP throttling scheme. As shown, the rate limiting method based on DNS queries gives better results than the rate limiting method based purely on IP addresses visited. The plots also indicate unmistakably that aggregated rate limiting at the edge router performs better than per-host limits.

6.9 An Empirical Analysis of Three Rate Limiting Mechanisms

In the following sections, we undertake an empirical analysis of existing rate limiting mechanisms, with the goal of understanding the relative performance of the various schemes. Our study is based on real traffic traces collected from the border of a network with 1200 hosts. The trace data includes real attack traffic of Blaster and Welchia, which infected over 100 hosts. We implement each scheme against the trace data and analyze their performance in terms of false positive and false negative rates. In the case of worm defense, it is particularly important that false positives are kept at a minimum without greatly impacting false negatives.

We analyze the efficacy of the various schemes on both worm traces and normal traffic. The inclusion of real worm data allows us to draw insights without having to consider the limitations of simulated attacks. We study three rate limiting schemes, Williamson's IP throttling [Williamson02], Chen's failed-connection-based scheme [ChenS04] and Schechter's credit-based rate limiting [Schecter04]. Williamson's throttling scheme limits the rate of distinct IP connections from an end host [Williamson02]. Chen et al. [ChenS04] and Schechter et al. [Schecter04] both apply rate limiting to hosts that exhibit an abnormally high number of failed connections. In addition, we study an alternative rate limiting strategy based on DNS statistics—namely limiting outgoing connections without prior DNS translations, thereby restricting the contact rate of scanning worms. Ganger et al. made the first observation that DNS-based statistics can be used to detect and contain malicious worms [Ganger02]. Recently Whyte et al. showed that DNS-based worm detection can be extended to a network setting [Whyte05]. The DNS-based rate limiting mechanism we study is a modified version of [Ganger02]. One goal of this study is to investigate using DNS behavior as a basis for rate limiting and its relative performance with respect to other schemes.

In addition to studying DNS-based rate limiting, the other components of our analysis seek to understand the fundamentals of rate limiting technology. For instance, we evaluate the impact of dynamic vs. static rates. We study the effect of host vs. edge-based deployment. Some of these issues were not explored adequately in the studies of the individual schemes. Our analysis is the first that we are aware of that offers evaluation of the different rate limiting schemes on an equal footing—running against the same traffic traces. The trace data we use in this study is from an open network without strict traffic policies. Since most of the rate limiting mechanisms target enterprise networks with stricter traffic settings, we believe that our analysis provides reasonable insights into how well these schemes might perform in practice.

6.9.1 Trace Data

This study was conducted using traffic traces collected from the border of an academic department. The network has 1200 externally routable hosts and serves approximately 1500 users. Hosts are used for research, administration, and general computing (web browsing, mail, etc). There is a diverse mix of operating systems on the network. Since May 2003 we recorded in an anonymized form all IP and common second layer headers of packets (e.g., TCP or UDP) leaving and entering the network. We also recorded DNS traffic payloads for use in the experiment in Section 6.13.

During the course of tracing, we recorded two worm attacks: *Blaster* and *Welchia* [Symantec03a, Symantec03b]. Both are scanning worms that exploited the Windows DCOM RPC vulnerability. For each attack recorded, we conducted post-mortem analysis to identify the set of infected hosts within the network. We further identified outbound worm traffic as those from infected hosts with a particular destination port (e.g., port 135 for Blaster). Whenever possible, a payload size identical or similar to those publicized in Symantec's worm advisories is used as additional evidence to identify worm traffic. It is important to note that infected hosts in our network were exclusively Windows clients that, under normal circumstances,

rarely (if ever) made any outbound port 135 connections to external addresses. Once infected, these hosts initiated tens of thousands of outbound connections to port 135. As such, the task of identifying worm traffic is made relatively easy.

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

Figure 6.22(a) shows the daily volume of outgoing traffic as seen by the edge router for the trace period. Figure 1(b) shows the number of distinct IP addresses daily. As shown, the aggregate outgoing traffic experienced a large spike as Blaster hits the network on day 6. At its peak, the edge router saw 11 million outbound flows in a day. This is in contrast to the normal 500,000 flows/day. The increase in traffic is predominantly due to worm activities.

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

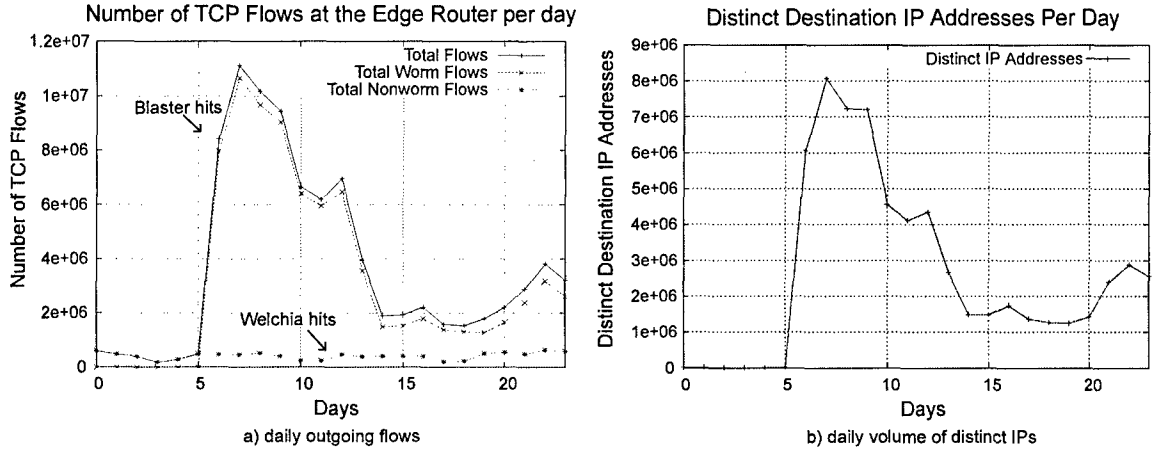


Figure 6.22: Traffic Statistics for the Blaster/Welchia Trace

6.9.2 Analysis Methodology

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

For each scheme analyzed, there exists a set of parameters that impact the performance of the mechanism. We identify these parameters and evaluate the sensitivity of the error rates with respect to each parameter.

In some cases, the impact of the parameters has not been studied previously. A contribution of our study is to understand precisely how these parameters might be implemented in practice.

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

6.10 Williamson's IP Throttling

Williamson's IP throttling scheme operates on the assumption that normal applications typically exhibit a stable contact rate to a limited number of external hosts (e.g., web servers, file servers) [Williamson02]. Restricting host-level contact rates to unique IPs can limit rapid connections to random addresses (e.g., worm traffic). Williamson accomplishes this by keeping a *working set* of addresses for each host, which models the normal contact behavior of the host. The throttling mechanism permits outgoing connections for addresses in the working set, but delays other packets by placing them in a delay queue. If the delay queue is full, further packets are simply dropped. The packets in the delay queue are dequeued and processed at a constant rate (one per second, as suggested by [Williamson02]). At the same rate, the least recently used address in the working set is evicted to make room for the new connection. As a result, connections to frequently contacted addresses are allowed through with a high probability while connections to random addresses (as those initiated by scanning worms) are likely delayed and possibly dropped.

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

End Host Throttling: To analyze Williamson's end host IP throttling, we reconstructed end-host traffic from our trace and simulated Williamson's rate limiting scheme using these traces. Figure 6.23(a) shows the daily false positive rate for infected hosts with the size of the working set ranging from 4 to 10. Again, false positive rates are calculated as the percentage of benign traffic subjected to rate limiting. The data points in Figure 6.23(a) show daily false positive statistics as averages across *infected* hosts while the host stayed infected. For comparison reasons, we tested Williamson's scheme on normal hosts, the result of which are shown in Figure 6.23(b).

A few high-level insights are important here: First, Figure 6.23 suggests that false positives are low during normal operation (about 15%). Once infection occurs, however, Williamson's scheme yields false positive rates nearly 90%. This is undesirable as during the worm outbreak, essentially all benign traffic is subjected to delay incurred by the throttling scheme. Figure 6.23(d) shows the average queue length for infected hosts. As shown, when infection hit on day 6, the average queue length quickly reached the maximum (100 in this case) and remained in the neighborhood of 90%. This means that during infection, delay for each fresh IP connection was approximately 90 seconds or greater if the queue was filled with distinct hosts, which is likely to be the case due to the random scanning nature of the worms.

We note, however, the way we define false positives is slightly unfair; we label every delayed non-worm SYN packet a false positive. In reality, many applications can tolerate a slight delay. Table 6.1 shows the delay statistics for a normal host during a 3-hour period. As shown, all delays were less than 10 seconds, which may be entirely acceptable for certain applications. In contrast, Table 6.2 shows the worst case delay statistics for an infected host for the same time period. As shown, once a host is infected, the delay queue becomes saturated with worm packets and legitimate applications on the host are subjected to excessive delays and blockage.

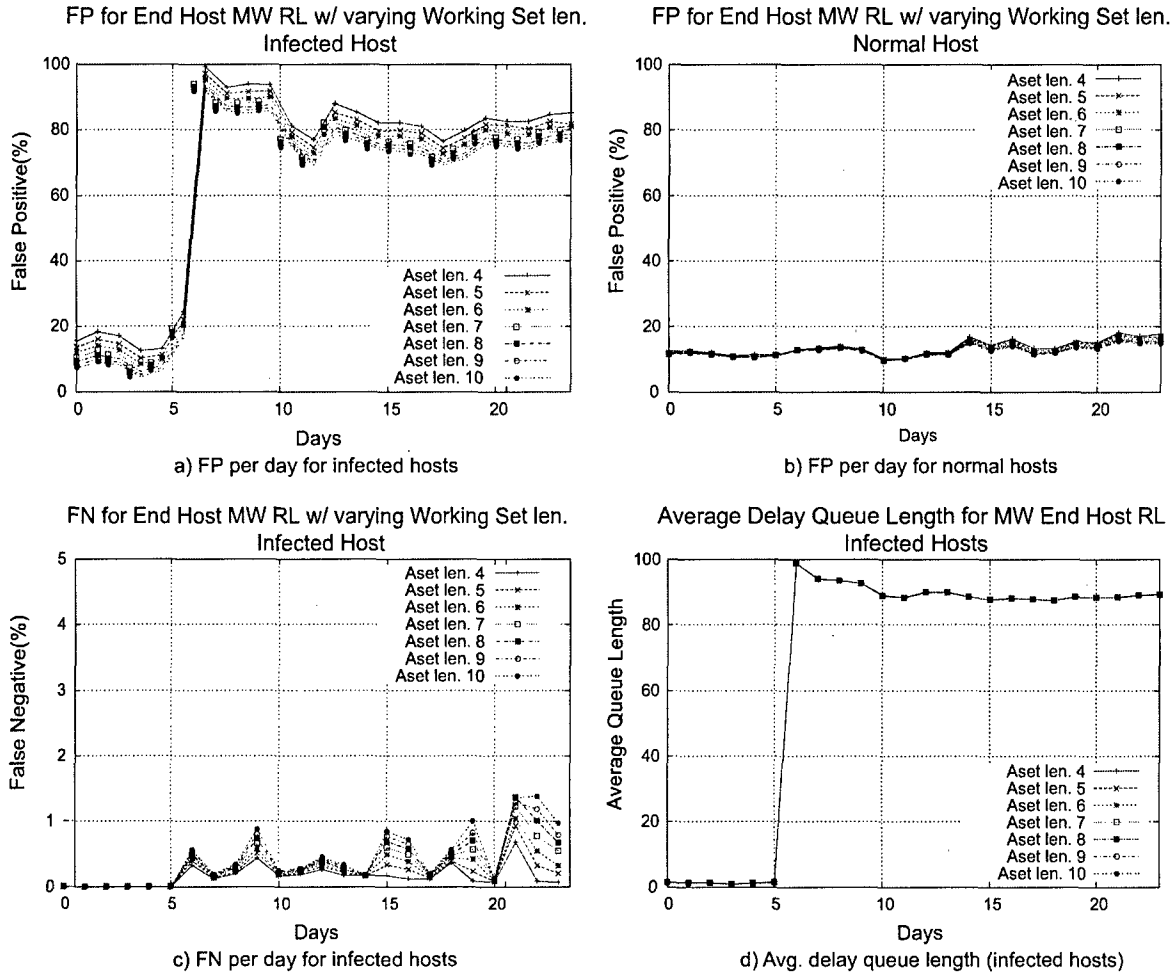


Figure 6.23: Results for Williamson's End Host RL mechanism

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

Figure 6.23(c) shows the false negative rates, which are predominantly below 1%. This means that Williamson's scheme is effective against worm spread, though it also incurs large delays for legitimate applications running on the same host. The strength of Williamson's scheme lies in its logical simplicity and ease of management. One can imagine a more complex data structure than a simple queue to deal with delayed connections. Alternatively, one can employ a dynamic rate scheme that changes the dequeuing rate accordingly with the length of the delay queue. Schemes such as these can potentially reduce the false positive rates, but at the price of increased complexity.

Throttling at the Edge Router: Previous studies [Moore03, Wong04a] showed that end-host rate limiting is ineffective unless deployment is universal. As part of this study, we investigate the effect of applying Williamson's throttling to the aggregate traffic at the edge of the network. Aggregate, edge-based

throttling is attractive because it requires the instrumentation of only the ingress/egress point of the subnet. Furthermore, aggregate throttling dose not require per-host state to be kept. We note that the logic of aggregate throttling can be extended to the border point of a network cell within an enterprise, as shown in [Staniford04], which can provide a finer protection granularity.

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

Table 6.1: Delay statistics for a normal host during a 3-hour period

In a previous traffic study, we identified a candidate rate of 16 addresses per five seconds for edge throttling for a similar network [Wong04a]. In the analysis that follows, we present results obtained with five aggregate rate limits: 10, 16, 20, 25 and 50 IPs per every five-second window.

Figure 6.24(a) shows the false positive rates for edge-router rate limiting using various rate limits. The corresponding false negative rates are shown in Figure 6.24(b). Compared with the end-host case, edge-based rate limiting exhibits significantly higher false positive rates during normal operation. This is primarily due to the fact that aggregate throttling penalizes hosts with atypical traffic patterns, thereby contributing to a higher false positive rate. We can increase the working set size at the edge to reduce the false positives, but false positives will increase accordingly. As such, Williamson's throttling is best suited for end-host rate limiting where behavior of a host is somewhat predictable.

6.11 Failed Connection Rate Limiting (FC)

Chen et al. proposed another rate limiting scheme based on the assumption that a host infected by a scanning worm will generate a large number of failed TCP requests [ChenS04]. Their scheme attempts to rate limit hosts that exhibit such behavior. In the discussions that follow, we refer to this scheme as FC (for Failed Connection).

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

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

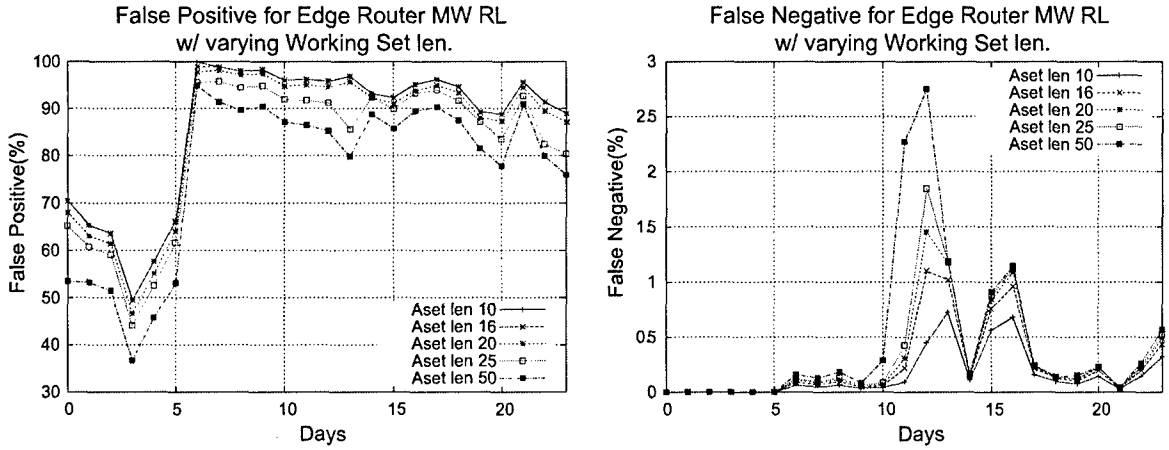
Temporal FC attempts to limit both the short term failure rate λ and a longer term rate Ω. Chen suggested Ω be a daily rate and a λ per second rate. The value of Ω is intended to be significantly smaller than λ * (seconds in a day). Hosts in a hash table entry are subjected to rate limiting if the failure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

To evaluate these two algorithms we conducted experiments with the border trace, with varying values of λ and Ω. Figure 6.25(a) and (b) show the error rates for basic and temporal FC, with λ equaling 1 and Ω

equaling 300, as recommended by Chen. Figure 6.25(a) shows an increase in the false positive rates during the first week of infection. This increase is due to the fact that a worm generates rapid failed connections and quickly depletes the available tokens. Until more tokens become available, legitimate traffic is stopped altogether, as seen in the third and forth row of Table 6.3.

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

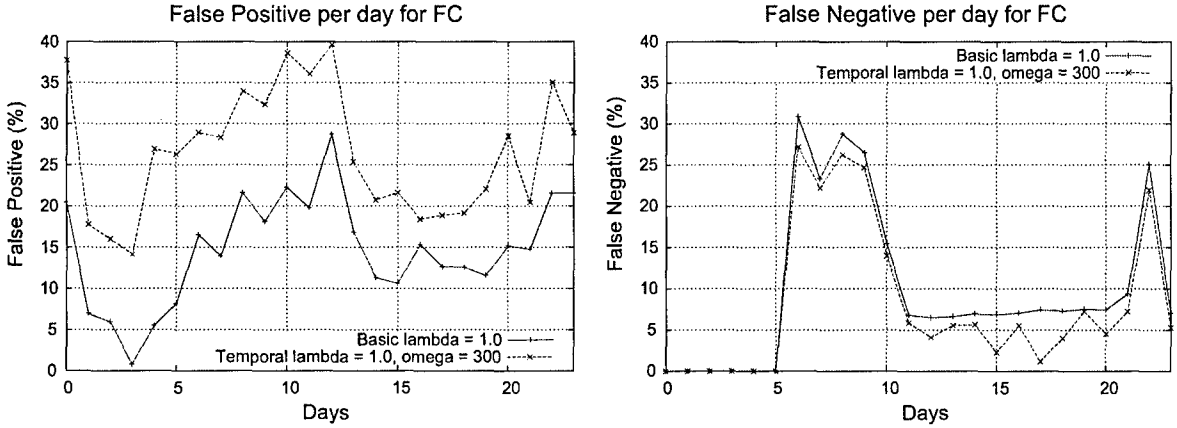
Table 6.2: Delay Statistics for an infected host during a 3-hour period



a) FP per day at edge router

b) FN per day at edge router

Figure 6.24: Results for Williamson's RL mechanism at Edge Router



a) FP per day

b) FN per day

Figure 6.25: Error rates per day for Basic and Temporal FC with $\lambda = 1.0$ & $\Omega = 300$.

In Figure 6.25(b) there is a pronounced initial jump in the false negative rates as Blaster hits on day 6, and in a few days the false negatives decrease significantly. The bulk of false negatives can be attributed to the fact that Chen's scheme uses only TCP RST as an indication of a failed connection. Since many firewalls simply drop packets instead of responding with TCP RSTs, using TCP RSTs exclusively underestimates the number of failed connections. Figure 6.26(b) shows the error rates including TCP TIMEOUTs. As shown, false negative rates of FC are reduced significantly when Timeouts are considered. The drop in false negative rate on day 10 in Figure 6.25(b) is correlated with the onset of the Welchia outbreak. Blaster scanning generates a substantial number of TCP TIMEOUTs while Welchia tends to generate TCP RSTs (Welchia scans via ICMP ECHO). As more and more Blaster hosts are patched and Welchia makes up a greater portion of the worm traffic, the false negatives are reduced.

Figure 6.26 plots the false positive rates against the false negative rates with varying values for λ and Ω. The data points in this graph are averaged daily statistics over the entire trace period. In temporal FC, when failures reach $\Omega/2$, the rate limiting algorithm proceeds to rate limit hosts in a much more aggressive fashion than the basic scheme. This strategy results in a significant amount of non-worm traffic from "infected" hosts being dropped. In the third row of Table 6.3, temporal FC dropped approximately 2.5 times more benign traffic compared to basic FC. Since a typical worm outbreak will quickly reach $\Omega/2$ failures, temporal FC is more restrictive and thus renders higher false positives.

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

Table 6.3: False Positives and Cause for Day 6 λ = 1.0 and Ω = 300

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

6.12 Credit-based Rate Limiting (CB)

Another rate limiting scheme based on failed connection statistics is the credit-based scheme by Schechter et. al. [Schecter04]. We refer to it as CB (for Credit Based). CB differs from Chen's in two significant ways. First, it performs rate limiting exclusively on *first contact* connections—outgoing connections for destination IPs that have not been visited previously. The underlying rationale is that scanning worms produce a large volume of failed connections, but more specifically they produce failed first-contact connections, therefore anomalous first-contact statistics are indicative of scanning behavior. The notion of first contact is fundamental to CB and as we show later is instrumental to its success. Second, CB considers both failed and successful connection statistics. Simply described, CB allocates a certain number of connection credits per host; each failed first-contact connection depletes one credit while a successful one adds a credit. A host is only allowed to make first-contact connections if its credit balance is positive.

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

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

Figure 6.27(a) shows CB's daily false positive and false negative rates with a 64-address PCH. The data points in this graph are averages across all hosts. As shown, the average false positive and false negative rates are between 5% and 15% during the infection period. The false positive results significantly outperform both FC and Williamson's. CB's false negative results are comparable to those of Williamson's. These results speak strongly of CB's insight of rate limiting first contacts rather than distinct IPs or straightforward failed connections. Since worm scanning consists primarily of first-contact connections, CB's strategy gives rise to a more precise means of rate limiting.

Table 6.4 shows the false positive data for the top two false-positive-generating hosts. Both clients that incurred high false positives are P2P clients. The data show that the worst case false positive rate is rather high—nearly 40% for the host in row one. For comparison reasons, here we also include the HTTP client discussed previously (row 3 from Table 6.3). As shown, CB is able to accommodate this bursty web client while FC dropped a significant portion of the client's traffic.

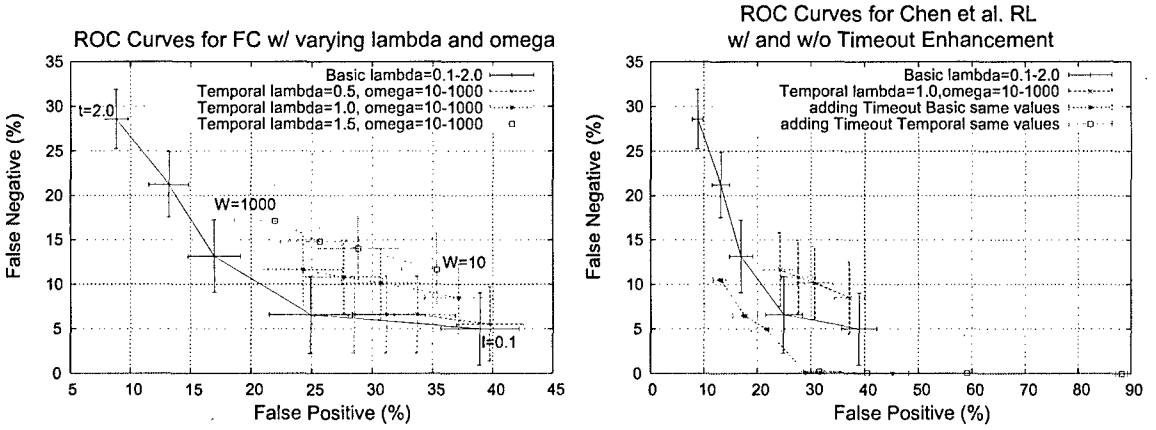


Figure 6.26: ROC for different λ and Ω values for Basic and Temporal RL algorithms

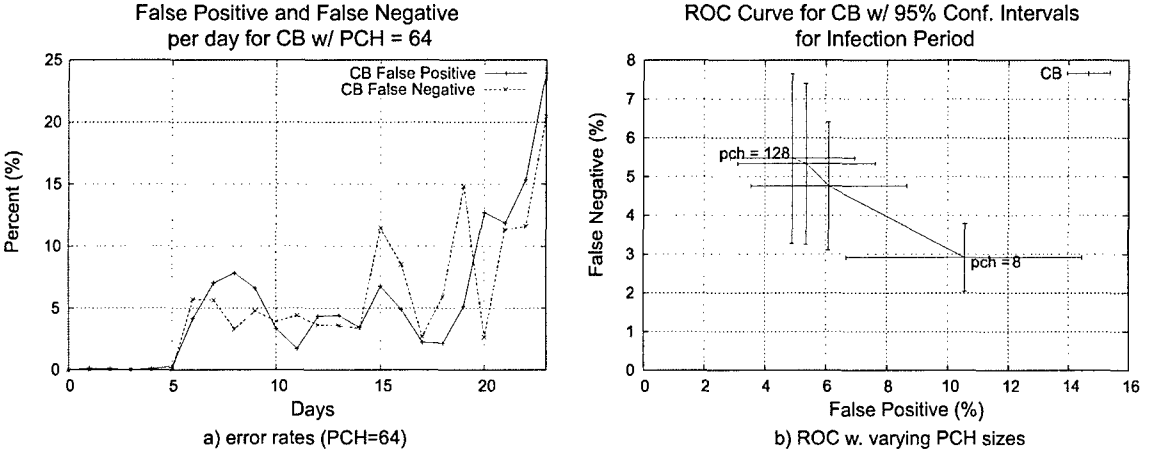


Figure 6.27: Results of Error Rates for CB RL

Figure 6.27(b) plots the average false positive rates against the corresponding false negative rates for PCH of 8, 16, 32, 64, and 128. The data points in this graph are obtained by averaging per-host statistics over the entire 24-day trace period (sans the pre-infection days). As shown, CB's error rates are not particularly sensitive to the length of the PCH's. A 6% increase in the false positive value is observed when PCH is reduced from 128 entries to 8. As the PCH size increased so did the false negative rate, which is a peculiar phenomenon. We are unable to find a satisfactory explanation for this. We conjecture that a possible error in the Blaster mutex code allowed multiple instances of Blaster to execute on the same machine, thereby generating repeated scanning to the same addresses.

Note that CB is essentially a host-based scheme since states are kept for each host. Aggregating and correlating connection statistics across the network can reduce the amount of state kept. For example, if host A makes a successful first-contact connection to an external address, further connections for that address could be permitted through regardless of the identity of the originating host. This optimizes for the scenario that legitimate applications (e.g., web browsing) on different hosts may visit identical external addresses (e.g., cnn.com). A more detailed investigation of aggregate CB can be found later in Section 6.14.

6.13 DNS-based Rate Limiting

In this section we analyze a rate limiting scheme based on DNS statistics. The underlying principle is that worm programs induce visibly different DNS statistics from those of legitimate applications [Wong04b, Whyte05, Ganger02]. For instance, the non-existence of DNS lookups is a telltale sign for scanning activity. This observation was first made by Ganger et al. [Ganger02]. The scheme we analyze here is a modification of Ganger's NIC-based DNS detection scheme.

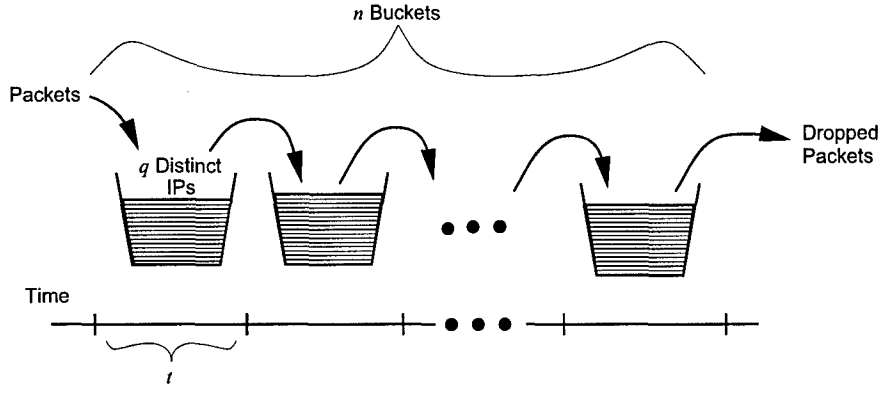


Figure 6.28: Cascading Bucket RL Scheme

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

Table 6.4: Per Host False Positives and Cause for Day 6 for PCH = 64

The high-level strategy of the DNS rate limiting scheme (hereafter refer to as DNS RL) is simple: for every outgoing TCP SYN, the rate limiting scheme permits it through if there exists a prior DNS translation for the destination IP, otherwise the SYN packet is rate limited. The algorithm uses a *cascading bucket* scheme to contain untranslated IP connections. A graphical illustration of the algorithm is shown in Figure 6.28. In this scheme, there exists a set of n buckets, each capable of holding q distinct IPs. The buckets are placed contiguously along the time axis and each spans a time interval t.

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

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

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

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

6.13.1 Analysis

The critical parameter for the cascading-bucket scheme is the rate limit, which manifests in the values of q (the size of each bucket), t (the time interval), and n (number of buckets). To simplify our analysis, we varied the value of q and kept n and t constant.[18] Additionally, the value of $n*t$ was set to 120 seconds to model the TCP timeout period. This scheme allows a certain number of untranslated IP connections to exit the network, which intends to accommodate legitimate direct-IP connections. In our data set, we observed some direct server-server communication and direct-IP connections due to peer-to-peer, streaming audio and passive FTP traffic. These were the main cause of false positives observed. One can attempt to maintain a white list to allow legitimate direct-IP connections and thus further reduce false positives. However, as observed in [Whyte05], a comprehensive white list for an open network may not be feasible.

We first analyze the host-level DNS throttling scheme. For this, we maintain a set of cascading buckets for each host. Figure 6.29(a) and (b) show the false positive and false negative rates for infected hosts. The data in these graphs are daily error rates averaged over all infected hosts. Figure 6.29(c) plots the analogous false positive rates for normal hosts. In addition, Table 6.5 presents the delay statistics for a normal host and Table 6.6 shows the worst case delay statistics for an infected host.

These results yield a number of observations: First, host-level DNS throttling significantly outperforms the other mechanisms analyzed previously. As seen in Figure 6.29, the average false positive rates fall in the range of 0.1% to 1.7% with corresponding false negative rates between 0.1% to 3.2%, both significantly lower than the error statistics of the others. We also observed that applications that do experience false positives here tend to be those that fall outside of the security policies of an enterprise network (e.g., peer-to-peer applications)— disruption of such applications are generally considered not critical to the network operation.

Table 6.5 shows the delay statistics for a normal host. As shown, DNS RL delayed 8 total flows for this host, as opposed to the 385 flows using Williamson's (Table 6.2). Also note that all the delays in Table 6.5 are less than 10 seconds, which are not significant. Table 6.6 shows the worst case delay statistics for

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

an infected host during the peak of its infection period. The statistics show that DNS RL dropped approximately 17% of the host's benign traffic, compared to over 90% when using Williamson's. In addition, DNS RL delays less flows for normal hosts than Williamson's. Also note in Table 6.6, nearly delayed malicious flows are subjected to the maximum allowed delay and over 95% of the malicious flows are dropped.

During the outbreak period, the false positives for infected hosts included both dropped and delayed traffic flows. A q value of 5 would drop approximately 0.075% and delay 0.375% of the legitimate traffic. Figure 6.29(d) shows summarized statistics from our analysis for a liberal value of q = 10. During Blaster's outbreak, on average 97% of the worm traffic was rate limited—approximately 82% dropped and the other 18% delayed with an average delay of one minute each.

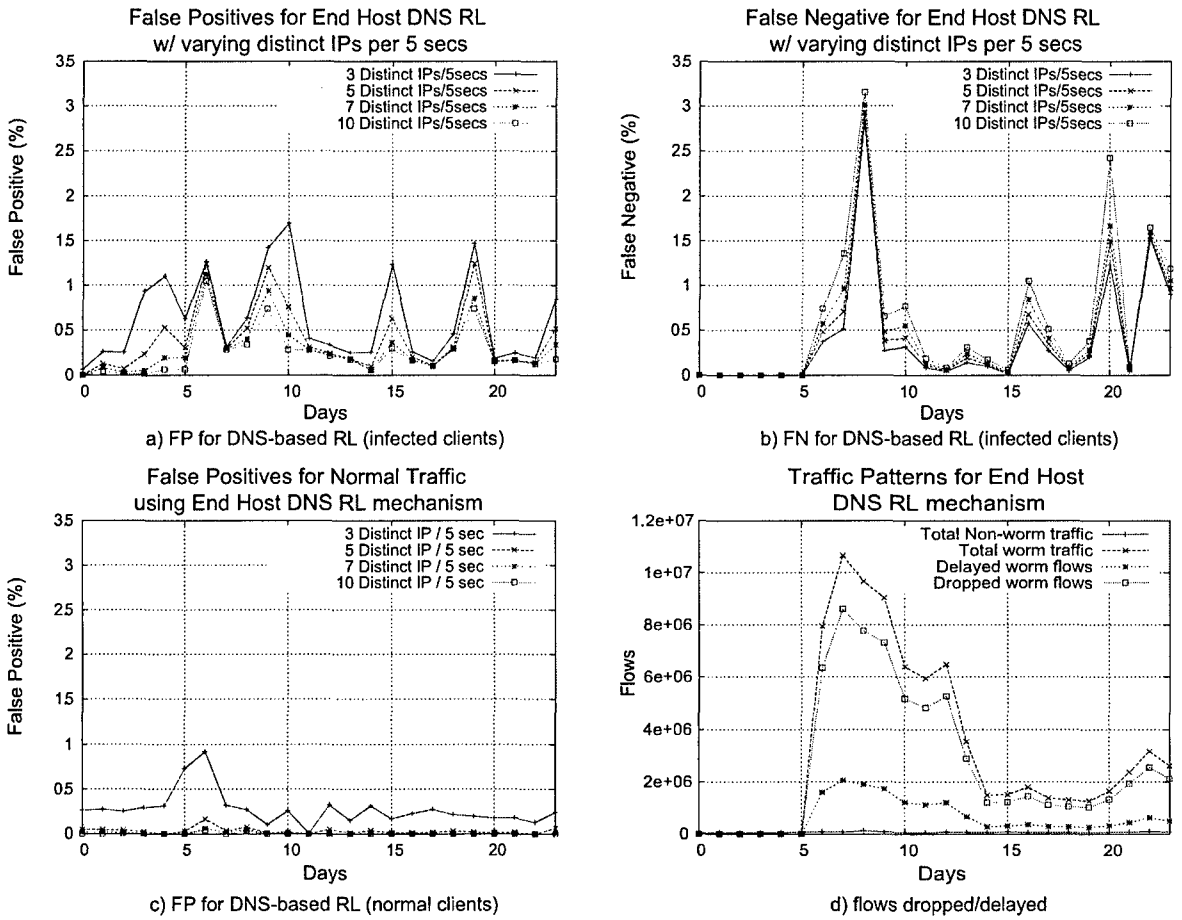


Figure 6.29: Results for DNS-based End Host RL

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

Table 6.5: DNS RL delay statistics for a normal host (3-hour period).

Our results also show that DNS rate limiting is capable of containing slow spreading worms. As a comparison, Weaver's Approximate TRW containment mechanism can block worms that scan faster than 1 scan per second [Weaver04]. Using the DNS scheme, with value of $q = 3$ and $t = 5$ for instance (3 direct-IP connections in 5- second window), we can contain worms that scan at the rate of 0.6 scans per second (or more) with 99% accuracy.

To test the effect of aggregate throttling, we implemented a single set of cascading buckets for the entire network. For this set of experiments, the value of q was set to 20, 50, and 100 IPs per five second window. Figure 6.30 shows the error rates for the aggregate implementation. As shown, a q value of 20 or 50 IPs yielded few false negatives and a false positive rate of approximately three to five percent. Note that when q is set to 20 or 50, the false negative rates of edge-based rate limiting are lower than the host-level scheme. This is because the aggregate traffic limit is more restrictive overall than the collective limit in the host-based case. Although the false positive rates for the aggregate case are slightly higher than the host-level case, overall the error rates are fairly low—5% false positive and < 1% false positive.

Delay Amount	Benign	Malicious
No delay	806	1
1 - 30 sec.	4	34
31 - 60 sec.	2	35
61 - 100 sec.	12	40
> 100 sec.	11	4903
Dropped	172	112862
Total	1007	117875

Table 6.6: DNS RL Delay Statistics for an infected host during a 3-hour period

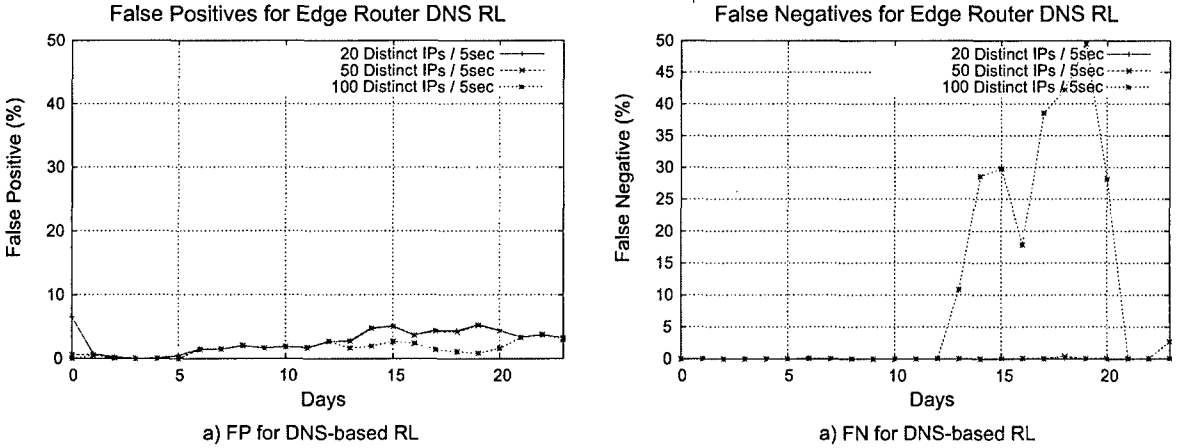


Figure 6.30: Results for DNS-based RL at the Edge Router

6.14 Discussion

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

DNS-based RL vs. others: A summary comparison of the DNS-based scheme with the others is in Figure 6.31. The parameters here are consistent with the values used in the previous sections. As shown, DNS-based rate limiting has the best performing false positive and false negative rates. Host-based DNS

throttling renders an average false positive and false negative rate below 1%. These results present a strong case for DNS-based rate limiting.

Recall that the q value in DNS throttling allows for q untranslated IP connections per host to exit the network every t seconds. To put things in perspective, for the first day of infection, the network had a total of 468,300 outbound legitimate flows. When q = 7a total of 463 legitimate flows are dropped, which yields a false positive rate of 0.099%. This is less than 1 dropped flow per host per day. As a comparison, CB dropped 3767 legitimate flows for the same day, a false positive rate of 7.8%.

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

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

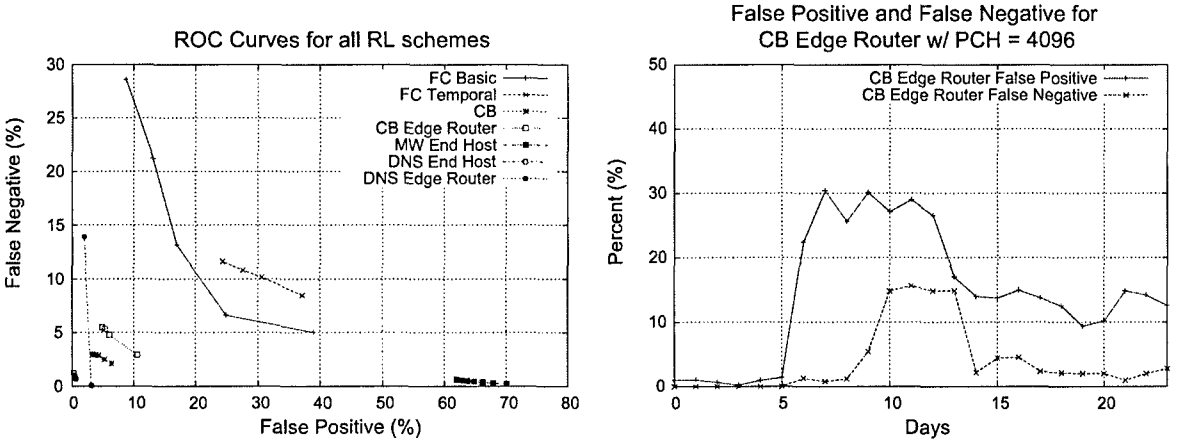


Figure 6.31: Avg. error rates for all RL schemes and Edge CB Results

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

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

First, a worm could use reverse DNS-lookups (PTR lookups) to "pretend" that it has received a DNS translation for a destination IP. Jung et al. [Jung01] characterizes that PTR lookups are primarily for incoming TCP connections or lookups related to reverse blacklist services. These types of lookups can be easily filtered and not considered as valid entries in the DNS cache. In addition, a PTR lookup prior to an infection attempt will significantly reduce the infection speed.

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

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

Dynamic vs. static rates: Rate limiting schemes impact the rate of both legitimate and malicious connections. Williamson's imposes a strictly static rate, e.g., five distinct IPs per second, irrespective of the traffic demand. FC is predominately static while CB allows for a dynamic traffic rate by rewarding successful connection and penalizing failed connections. Results in Figure 6.31 show that CB outperforms FC. This is partially due to CB's dynamic rates which render a more graceful filtering scheme that permits both bursty application behavior and temporarily abnormal-but-benign traffic patterns. As we briefly discussed in Section 6.10, mechanisms that impose a static rate can benefit from incorporating dynamic rate limits. Dynamic rate limiting is an interesting topic worth further study.

Host vs. aggregate: An issue of significance is host versus aggregate rate limiting. The general wisdom is that host-level throttling is more precise but is at the same time more costly because per host state must be maintained. Indeed, Williamson's IP throttling, when applied at the edge, rendered visibly higher false positives than its host-based counterpart. This is because IP contact behavior at the host-level is more fine-grained and thus more likely to be stable. In contrast, aggregate traffic at the edge includes hosts, whose behavior may vary significantly from each other, thereby contributing to a higher error rate. A similar case was observed with CB when applied to the aggregate traffic, the results of which are shown in Figure 6.31(b). As shown, the false positive rates reach approximately 30%, compared to the 10% with the host-based deployment. Edge-based DNS throttling, however, appears to be an exception. Figure 6.31(a) shows that a carefully chosen rate limit, e.g., 50 IPs per five seconds, yields excellent accuracy for edge-based DNS throttling. It has lower false positive and false negative rates than other host-based schemes. The fundamental reason behind this is that DNS statistics, in particular the presence (or the lack) of IP translations, remain largely invariant from host to the aggregate level.

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

We note that edge-based throttling in itself does not defend against internal infection. One way to protect against internal infection (and not pay the cost of host-level throttling) is to divide an enterprise network into various cells (as suggested by Staniford [Staniford04]) and apply the aggregate throttling at the border of each cell. We leave the analysis of more fine-grained, intra-network protection as future work.

6.15 Conclusions

A number of rate limiting schemes have been proposed recently to mitigate scanning worms and recent work in rate limiting schemes such as traffic throttling [Williamson02] and secure NICs [Ganger02] show potential in mitigating widespread worm attacks. Our contributions in this work are several: First, we showed through modeling and simulation experiments that deploying rate limiting filters at the backbone routers is extremely effective. Rate control at the edge routers is helpful for randomly propagating worms, but does very little to suppress local preferential spreading worms. Individual host based rate control results in a slight linear slowdown of the worm spread, regardless of the spreading algorithm. A direct consequence of this analysis is that in order to secure an enterprise network, one must install rate limiting filters at the edge routers as well as some portion of the internal hosts.

Second, through a study of real network traces from a campus computing network, we discovered that there exist reasonable rate limits for an enterprise network that would severely restrict the spread of a

worm but would have negligible impact on almost all legitimate traffic. This is especially encouraging since rate limiting filters can be easily installed and configured at various strategic points throughout a network. The result of the trace study confirmed that per-host rate limiting by itself is not sufficient to secure the enterprise network—aggregated rate limiting at the edge router must be employed at the same time to minimize the spread of worm attacks. This is the first study in this area of which we are aware of that has studied rate limiting with real traffic traces and has identified realistic rate limits in practice.

Third, in our empirical analysis of the different rate limiting schemes, using real traffic and attack traces from an open network environment, we found that the scheme that performs well in an open network and will perform equally well (if not better) in an environment with strict traffic policies (e.g., enterprise network). Our analysis reveals these insights: First, the subject of rate limiting is by far the most significant "parameter"—failed-connection behavior alone is too restrictive as evidenced by FC; rate limiting first-contacts renders better results and DNS behavior-based rate limiting is by far the most accurate strategy. Second, it is feasible to delineate worm behavior from normal traffic even at an aggregate level, as indicated by the DNS analysis. This is an interesting result because aggregate rate limiting alleviates the universal participation requirement thought necessary for worm containment [Moore03, Wong04a]. This result also suggests that it may be possible to apply rate limiting deeper into the core of the network, a subject that is of great interest to many. Third, preliminary investigation suggests that incorporating dynamic rates results in increased accuracy. As most of rate limiting schemes to-date focus on static rates, an immediate follow-up research is dynamic rate limiting and how that can be implemented in practice.

7 SURVIVABLE STORAGE BASED ON SELF-SECURING STORAGE DEVICES

7.1 Introduction

Survivable storage systems spread data redundantly across a set of distributed storage-nodes in an effort to ensure its availability despite the failure or compromise of storage-nodes. Some systems support only replication, while others also use more space-efficient (and network-efficient) erasure-coding approaches in which a *fragment*, which is smaller than a full copy, is stored at each storage-node. Client read and write operations interact with multiple storage-nodes, according to some consistency protocol, to implement consistency in the face of client and storage-node faults and concurrent operations.

This chapter describes and evaluates a new consistency protocol that operates in an asynchronous environment and tolerates Byzantine failures of clients and storage-nodes. It then goes on to introduce *lazy verification* and describe implementation techniques for exploiting its potential. Lazy verification enables the system to amortize verification effort over multiple operations, to perform verification during otherwise idle time, and to have only a sub-set of storage-nodes perform verification.

7.1.1 The Consistency Protocol

The consistency protocol supports a hybrid failure model in which up to t storage-nodes may fail: $b \leq t$ of these failures can be Byzantine and the remainder can be crash. The protocol also supports the use of m-of-n erasure codes (i.e., m-of-n fragments are needed to reconstruct the data), which usually require less network bandwidth (and storage space) than full replication.

Briefly, the protocol works as follows. To perform a write, a client erasure codes a data-item into a set of fragments, determines the current logical time and then writes the time-stamped fragments to at least a threshold set of storage-nodes. Storage-nodes keep all versions of the fragments they are sent (in practice, until garbage collection frees them). To perform a read, a client fetches the latest fragment versions from a subset of storage-nodes and determines whether they comprise a completed write; usually, they do. If they do not, additional and historical fragments are fetched, and repair may be performed, until a completed write is observed. The fragments are then decoded and the data-item is returned.

The protocol gains efficiency from five features. First, it supports the use of space-efficient m-of-n erasure codes that can substantially reduce communication overheads. Second, most read operations complete in a single round trip: reads that observe write concurrency or failures (of storage-nodes or a client write) may incur additional work. Most studies of distributed storage systems (e.g., [Baker91, Noble94]) indicate that concurrency is uncommon (i.e., writer-writer and writer-reader sharing occurs in well under 1% of operations). Failures, although tolerated, ought to be rare. Third, incomplete writes are replaced by subsequent writes or reads (that perform repair), thus preventing future reads from incurring any additional cost; when subsequent writes do the fixing, additional overheads are never incurred. Fourth, most protocol processing is performed by clients, increasing scalability via the well-known principle of shifting work from servers to clients [Howard88]. Fifth, the protocol only requires the use of cryptographic hashes, rather than more expensive digital signatures.

Following sections describe the protocol in detail and develop bounds for thresholds in terms of the number of failures tolerated (i.e., the protocol requires at least $2t+2b+1$ storage-nodes). Also described and evaluated is its use in a prototype storage system called PASIS [Wylie00]. To demonstrate that our protocol is efficient in practice, we compare its performance to BFT [Castro01, Castro02], the Byzantine fault-tolerant replicated state machine implementation that Castro and Rodrigues have made available [Castro]. Experiments show that PASIS scales better than BFT in terms of server network utilization and in terms of work performed by the server. Experiments also show that response times of BFT (using multicast) and PASIS are comparable.

This protocol is timely because many research storage systems are investigating practical means of achieving high fault tolerance and scalability of which some are considering the support of erasure-coded

data (e.g., [Frolund03, Ganger03b, Kubiatowicz00, Morris02]). Our protocol for Byzantine-tolerant erasure-coded storage can provide an efficient, scalable, highly fault-tolerant foundation for such storage systems.

7.1.2 Lazy Verification

Sections 7.6 through 7.9 of this chapter describe a read/write consistency protocol that uses versioning storage to avoid proactive write-time verification. Rather than verification occurring during every write operation, it is performed by clients during read operations. Read-time verification eliminates the work for writes that become obsolete before being read, such as occurs when the data is deleted or overwritten. Such data obsolescence is quite common in storage systems with large client caches, as most reads are satisfied by the cache but writes must be sent to storage-nodes to survive failures. One major downside to read-time verification, however, is the potentially unbounded cost: a client may have to sift through many ill-formed or incomplete write values. A Byzantine client could degrade service by submitting large numbers of bad values. A second practical issue is that our particular approach to verification for erasure-coding, called validating timestamps, requires a computation cost equivalent to fragment generation on every read.

Lazy verification operates by having the storage-nodes perform verification in the background. It avoids verification for data that has a short lifetime and is never read. In addition, it allows verification to occur during otherwise idle periods, which can eliminate all verification overhead in the common cases. Clients will only wait for verification if they read a data-item before verification of the most recent write operation to it completes. Our lazy verification implementation limits the number of unverified write operations in the system and, thus, the number of retrievals that a reader must perform to complete a read. When the limit on unverified writes is reached, each write operation must be verified until the storage-nodes restore some slack. Limits are tracked on a per-client basis, a per-data-item basis, and on a storage-node basis (locally).

Combined, the different limits can mitigate the impact of Byzantine clients (individually and collectively) while minimizing the impact on correct clients. In the worst case, a collection of Byzantine clients can force the system to perform verification on every write operation, which is the normal-case operation for most other protocols.

This section describes and evaluates the design and implementation of lazy verification in the PASIS storage system [Goodson04a]. Several techniques are introduced for reducing the impact of verification on client reads and writes, as well as bounding the read-time delay that faulty clients can insert into the system. For example, a subset of updated storage-nodes can verify a write operation and notify the others, reducing the communication complexity from $O(N^2)$ to $O(bN)$ in a system of N storage-nodes tolerating b Byzantine failures; this reduces the number of messages by 33% even in the minimal system configuration. Appropriate scheduling allows verification to complete in the background. Indeed, in workloads with idle periods, the cost of verification is hidden from both the client read and write operations. Appropriate selection of updates to verify can maximize the number of verifications avoided (for writes that become obsolete). The overall effect is that, even in workloads without idle periods, the lazy verification techniques implemented provide over a factor of four higher write throughput when compared to a conventional approach that proactively performs verification at write-time.

7.2 Efficient Byzantine-tolerant Erasure-coded Storage: Background

In a fault-tolerant, or survivable, distributed storage system, clients write and (usually) read data from multiple storage-nodes. This scheme provides access to data-items even when subsets of the storage-nodes have failed. One difficulty created by this architecture is the need for a consistent view, across storage-nodes, of the most recent update. Without such consistency, data loss is possible. To provide reasonable semantics, storage systems must guarantee that readers see consistent data-item values. Specifically, the linearizability [Herlihy90] of operations is desirable for a shared storage system.

A common data distribution scheme used in such systems is replication. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. Alternately, more space-efficient erasure coding schemes can be used to reduce network load and storage consumption. With erasure coding schemes, reads require fragments from multiple servers. Moreover, to decode the data-item, the set of fragments read must correspond to the same write operation.

Most prior systems implementing Byzantine fault tolerant services adopt the replicated state machine approach [Schneider90], whereby all operations are processed by server replicas in the same order (*atomic broadcast*). While this approach supports a linearizable, Byzantine fault-tolerant implementation of any deterministic object, such an approach cannot be wait-free [Fischer85, Herlihy91, Jayanti98]. Instead, such systems achieve liveness only under stronger timing assumptions. An alternative to state machine replication is a Byzantine quorum system [Malkhi97], from which our protocol inherits techniques (i.e., our protocol can be considered a Byzantine quorum system that uses the threshold quorum construction). Protocols for supporting a linearizable implementation of any deterministic object using Byzantine quorums have been developed (e.g., [Malkhi01]), but also necessarily forsake wait-freedom to do so. Additionally, most protocols accessing Byzantine quorum systems utilize computationally expensive digital signatures.

Byzantine fault-tolerant protocols for implementing read-write objects using quorums are described in [Herlihy87, Malkhi97, Martin02]. In our protocol, a reader may retrieve fragments for several versions of the data-item in the course of identifying the return value of a read. Similarly, readers in [Martin02] "listen" for updates (versions) from storage-nodes until a complete write is observed. Conceptually, our approach differs in that clients read past versions, versus listening for future versions broadcast by servers. In our fault model, especially in consideration of faulty clients, our protocol has several advantages. First, our protocol works for erasure-coded data, whereas extending [Martin02] to erasure coded data appears nontrivial. Second, ours provides better message efficiency. Third, ours requires less computation, in that we do not require the use of expensive digital signatures. Advantages of [Martin02] are that it tolerates a higher fraction of faulty servers than our protocol, and does not require servers to store a potentially unbounded number of data-item versions. Our prior analysis of versioning storage, however, suggests that the latter is a non-issue in practice [Strunk00], and even under attack this can be managed using the garbage collection mechanism we describe in Section 7.5.1.1.

7.3 System Model

We describe the system infrastructure in terms of *clients* and *storage-nodes*. There are N storage-nodes and an arbitrary number of clients in the system. A client or storage-node is *correct* in an execution if it satisfies its specification throughout the execution. A client or storage-node that deviates from its specification *fails*. We assume a hybrid failure model for storage-nodes. Up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults [Lamport82]; the remainder are assumed to crash. We make no assumptions about the behavior of Byzantine storage-nodes and Byzantine clients. A client or storage-node that does not exhibit a Byzantine failure (it is either correct or crashes) is *benign*.

Our protocol tolerates Byzantine faults in any number of clients and a limited number of storage nodes while implementing linearizable and wait-free read-write objects. Linearizability is adapted appropriately for Byzantine clients (we discuss the necessary adaptations in [Goodson04b]). Wait-freedom functions as in the model of Jayanti et al. [Jayanti98] and assumes no storage exhaustion.

The protocol tolerates crash and Byzantine clients. As in any practical storage system, an authorized Byzantine client can write arbitrary values to storage, which affects the value of the data, but not its consistency. We assume that Byzantine clients and storage-nodes are computationally bounded so that we can benefit from cryptographic primitives.

We assume an asynchronous model of time (i.e., we make no assumptions about message transmission delays or the execution rates of clients and storage-nodes, except that it is non-zero). We assume that communication between a client and a storage-node is point-to-point, reliable, and authenticated: a correct

storage-node (client) receives a message from a correct client (storage-node) if and only if that client (storage-node) sent it.

There are two types of operations in the protocol — *read operations* and *write operations* — both of which operate on data-items. Clients perform read/write operations that issue multiple read/write requests to storage-nodes. A read/write request operates on a *data-fragment*. A data-item is *encoded* into data-fragments. Clients may encode data items in an erasure-tolerant manner; thus the distinction between data-items and data-fragments. Requests are *executed* by storage-nodes; a correct storage-node that executes a write request *hosts* that write operation.

Storage-nodes provide fine-grained versioning; correct storage-nodes host a version of the data-fragment for each write request they execute. There is a well known zero time, **0**, and null value, $\perp$, which storage-nodes can return in response to read requests. Implicitly, all stored data is initialized to $\perp$ at time **0**.

7.4 Byzantine Fault-tolerant Consistency Protocol

This section describes our Byzantine fault-tolerant consistency protocol, which efficiently supports erasure-coded data-items by taking advantage of versioning storage-nodes. It presents the mechanisms employed to encode and decode data, and to protect data integrity from Byzantine storage nodes and clients. It then describes, in detail, the protocol in pseudo-code form. Finally, it develops constraints on protocol parameters to ensure linearizability and wait-freedom.

7.4.1 Overview

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations and to identify data-fragments pertaining to the same write operation across the set of storage-nodes. Data-fragments are generated by erasure-coding data-items. For each write, a logical timestamp is constructed by the client that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp). This is accomplished by querying storage-nodes for the greatest timestamp they host, and then incrementing the greatest response. In order to verify the integrity of the data, a hash that can verify data-fragment correctness is appended to the logical timestamp.

To perform a read operation, clients issue read requests to a subset of storage-nodes. Once at least a read threshold of storage-nodes reply, the client identifies the *candidate*—the response with the greatest logical timestamp. The set of read responses that share the timestamp of the candidate comprise the *candidate set*. The read operation classifies the candidate as *complete*, *repairable*, or *incomplete*. If the candidate is classified as complete, the data-fragments, timestamp, and return value are validated. If validation is successful, the candidate's value is decoded and returned; the read operation is complete. Otherwise, the candidate is reclassified as incomplete. If the candidate is classified as repairable, it is repaired by writing data-fragments back to the original set of storage-nodes. Prior to performing repair, data-fragments are validated in the same manner as for a complete candidate. If the candidate is classified as incomplete, the candidate is discarded, previous data-fragment versions are requested, and classification begins anew. All candidates fall into one of the three classifications, even those corresponding to concurrent or failed write operations.

7.4.2 Mechanisms

Several mechanisms are used in our protocol to achieve linearizability in the presence of Byzantine faults.

7.4.2.1 Erasure Codes

We use threshold erasure coding schemes, in which N data-fragments are generated during a write (one for each storage-node), and any m of those data-fragments can be used to decode the original data-item. Moreover, any m of the data-fragments can deterministically generate the other $N - m$ data-fragments.

7.4.2.2 Data-fragment Integrity

Byzantine storage-nodes can corrupt their data-fragments. As such, it must be possible to detect and mask up to b storage-node integrity faults.

Cross checksums: Cross checksums [Gong89] are used to detect corrupt data-fragments. A cryptographic hash of each data-fragment is computed. The set of N hashes are concatenated to form the cross checksum of the data-item. It is then stored with each data-fragment (i.e., it is replicated N times). Cross checksums enable read operations to detect data-fragments that have been modified by storage-nodes.

7.4.2.3 Write Operation Integrity

Mechanisms are required to prevent Byzantine clients from performing a write operation that lacks integrity. If a Byzantine client generates random data-fragments (rather than erasure coding a data-item correctly), then each of the $\binom{N}{m}$ permutations of data-fragments could "recover" a distinct data-item. These attacks are similar to *poisonous writes* for replication as described by Martin et al. [Martin02]. To protect against Byzantine clients, the protocol must ensure that read operations only return values that are written correctly (i.e., that are *single-valued*). To achieve this, the cross checksum mechanism is extended in three ways: validating timestamps, storage-node verification, and validated cross checksums.

Validating timestamps: To ensure that only a single set of data-fragments can be written at any logical time, the hash of the cross checksum is placed in the low order bits of the logical timestamp. Note, the hash is used for space-efficiency; instead, the entire cross checksum could be placed in the low bits of the timestamp.

Storage-node verification: On a write, each storage-node verifies its data-fragment against its hash in the cross checksum. The storage-node also verifies the cross checksum against the hash in the timestamp. A correct storage-node only executes write requests for which both checks pass. Via this approach, a Byzantine client cannot make a correct storage-node appear Byzantine. It follows that only Byzantine storage-nodes can return data-fragments that do not verify against the cross checksum.

Validated cross checksums: To ensure that the client that performed a write operation acted correctly, the reader must validate the cross checksum. To validate the cross checksum, all N data-fragments are required. Given any m data-fragments, the full set of N data-fragments a correct client should have written can be generated. The "correct" cross checksum can then be computed from the regenerated set of data-fragments. If the generated cross checksum does not match the verified cross checksum, then a Byzantine client performed the write operation. Only a single-valued write operation can generate a cross checksum that verifies against the validating timestamp.

7.4.2.4 Authentication

Clients and storage-nodes must be able to validate the authenticity of messages. We use an authentication scheme based on pair-wise shared secrets (e.g., between clients and storage-nodes), in which RPC arguments and replies are accompanied by an HMAC [Bellare96] (using the shared secret as the key). We assume an infrastructure is in place to distribute shared secrets. Our implementation uses an existing Kerberos [Steiner88] infrastructure.

7.4.3 Pseudo-code

The pseudo-code for the protocol is shown in Figures 7.1 and 7.2. The symbol LT denotes logical time and $LT_{\text{candidate}}$ denotes the logical time of the candidate. The set $\{D_1, \ldots, D_N\}$ denotes the N data-fragments; likewise, $\{S_1, \ldots, S_N\}$ denotes the set of N storage-nodes. In the pseudo-code, the binary operator '|' denotes string concatenation. Simplicity and clarity in the presentation of the pseudo-code was chosen over obvious optimizations that are in the actual implementation.

7.4.3.1 Storage-node Interface

Storage-nodes offer interfaces to: write a data-fragment with a specific logical time (WRITE); query the greatest logical time of a hosted data fragment (TIME_REQUEST); read the hosted data-fragment with the greatest logical time (READ_LATEST); and read the hosted data-fragment with the greatest logical time at or before some logical time (READ_PREV). Due to its simplicity, storage-node pseudo-code is omitted.

7.4.3.2 Write Operation

The WRITE operation consists of determining the greatest logical timestamp, constructing write requests, and issuing the requests to the storage-nodes. First, a timestamp greater than, or equal to, that of the latest complete write must be determined. Collecting $N - t$ responses, on line 7 of READ_TIMESTAMP, ensures that the response set intersects a complete write at a correct storage-node. In practice, the timestamp of the latest complete write can be observed with fewer responses.

Next, the ENCODE function, on line 2 of WRITE, encodes the data-item into N data-fragments. The data-fragments are used to construct a cross checksum from the concatenation of the hash of each data-fragment (line 3). The function MAKE_TIMESTAMP, called on line 4, generates a logical timestamp to be used for the current write operation. This is done by incrementing the high order bits of the greatest observed logical timestamp from the *ResponseSet* (i.e., *LT:TIME*) and appending the *Verifier*. The *Verifier* is just the hash of the cross checksum.

```
WRITE(Data) :
1: Time:= READ_TIMESTAMP()
2: {D1, . . . ,DN}:= ENCODE(Data)
3: CC := MAKE_CROSS_CHECKSUM({D1, . . . ,DN})
4: LT := MAKE_TIMESTAMP(Time, CC)
5: DO_WRITE({D1, . . . ,DN}, LT, CC)

READ_TIMESTAMP() :
1: for all Si ∈ {S1, . . . ,SN} do
2:     SEND(Si , TIME_REQUEST)
3: end for
4: ResponseSet := Ø
5: repeat
6:     ResponseSet := ResponseSet ∪ RECEIVE(S, TIME_RESPONSE)
7: until (UNIQUE_SERVERS(ResponseSet) = N - t)
8: Time := MAX[ResponseSet.LT.Time]
9: RETURN(Time)

MAKE_CROSS_CHECKSUM({D1, . . . ,DN}) :
1: for all Di ∈ {D1, . . . ,DN} do
2:     Hi := HASH(Di )
3: end for
4: CC := H1| . . . |HN
5: RETURN(CC)

MAKE_TIMESTAMP(LTmax, CC) :
1: LT:Time := LTmax:Time + 1
2: LT:Verifier := HASH(CC)
3: RETURN(LT)

DO_WRITE({D1, . . . ,DN}, LT, CC) :
1: for all Si ∈ {S1, . . . ,SN} do
2:     SEND(Si, WRITE_REQUEST, LT, Di, CC)
3: end for
4: ResponseSet := Ø
5: repeat
6:     ResponseSet := ResponseSet ∪ RECEIVE(S, WRITE_RESPONSE)
7: until (UNIQUE_SERVERS(ResponseSet) = N - t)
```

Figure 7.1: Write operation pseudo-code.

Finally, write requests are issued to all storage-nodes. Each storage-node is sent a specific data-fragment, the logical timestamp, and the cross checksum. A storage-node validates the cross checksum with the verifier and validates the data-fragment with the cross checksum before executing a write request (i.e., storage-nodes call VALIDATE listed in the read operation pseudo-code). The write operation returns to the issuing client once WRITE_RESPONSE messages are received from $N - t$ storage-nodes (line 7 of DO_WRITE). Since the environment is asynchronous, a client can wait for no more than $N - t$ responses. The function UNIQUE_SERVERS determines how many unique servers are present in the candidate set; it ensures that only a single response from each Byzantine storage-node is counted.

7.4.3.3 Read Operation

The read operation iteratively identifies and classifies candidates, until a repairable or complete candidate is found. Once a repairable or complete candidate is found, the read operation validates its correctness and returns the data.

The read operation begins by issuing READ_LATEST requests to all storage-nodes (via the DO_READ function). Each storage-node responds with the data-fragment, logical timestamp, and cross checksum corresponding to the greatest timestamp it has executed. The integrity of each response is individually validated through the VALIDATE function, called on line 7 of DO_READ. This function checks the cross

checksum against the *Verifier* found in the logical timestamp and the data-fragment against the appropriate hash in the cross checksum. Although not shown in the pseudocode, the client only considers responses from storage nodes to READ_PREV requests that have timestamps strictly less than that given in the request.

Since, in an asynchronous system, slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N - t$ read responses can be collected (line 10 of DO_READ). Since correct storage-nodes perform the same validation before executing write requests, the only responses that can fail the client's validation are those from Byzantine storage-nodes. For every discarded Byzantine storage-node response, an additional response can be awaited.

After sufficient responses have been received, a candidate for classification is chosen. The function CHOOSE_CANDIDATE, called on line 3 of READ, determines the candidate timestamp, denoted $LT_{candidate}$, which is the greatest timestamp found in the response set. All data-fragments that share $LT_{candidate}$ are identified and returned as the *CandidateSet*. The candidate set contains a set of validated data-fragments that share a common cross checksum and logical timestamp.

Once a candidate has been chosen, it is classified as complete, repairable, or incomplete based on the size of the candidate set. The definitions of INCOMPLETE and COMPLETE are given in the following subsection. If the candidate is classified as incomplete, a READ_PREV request is sent to each storage-node with its timestamp. Candidate classification begins again with the new response set. If the candidate is classified as either complete or repairable, the candidate set contains sufficient data fragments written by the client to decode the original data-item. To validate the observed write's integrity, the candidate set is used to generate a new set of data-fragments (line 6 of READ). A validated cross checksum, CC_{valid}, is computed from the generated data-fragments. The validated cross checksum is compared to the cross checksum of the candidate set (line 8 of READ). If the check fails, the candidate was written by a Byzantine client; the candidate is then reclassified as incomplete and the read operation continues.

If the check succeeds, the candidate was written by a correct client and the read enters its final phase. Note that for a candidate set classified as complete this check either succeeds or fails for all correct clients regardless of which storage-nodes are represented within the candidate set.

If necessary, repair is performed: write requests are issued with the generated data-fragments, the validated cross checksum, and the logical timestamp (line 10 of READ). Storage-nodes not hosting the write execute the write at the given logical time; those already hosting the write are safe to ignore it. Finally, the function DECODE, on line 14 of READ, decodes m data-fragments, returning the data-item.

```
READ() :
1: ResponseSet := DO_READ(READ_LATEST_REQUEST, ?)
2: loop
3:     ⟨CandidateSet, LT_candidate⟩ :=
                    CHOOSE_CANDIDATE(ResponseSet)
4:     if (|CandidateSet| ≥ INCOMPLETE then
5:         /* Complete or repairable write found */
6:         {D_1,...,D_N} := GENERATE_FRAGMENTS(CandidateSet)
7:         CC_valid := MAKE_CROSS_CHECKSUM({D_1, . . . ,D_N})
8:         if (CC_valid = CandidateSet.CC) then
9:             /* Cross checksum is validated */
10:            if (|CandidateSet| < COMPLETE) then
11:                /* Repair is necessary */
12:                DO_WRITE({D_1, . . . ,D_N}, LT_candidate, CC_valid)
13:            end if
14:            Data := DECODE({D_1, . . . ,D_N})
15:            RETURN(Data)
16:        end if
17:    end if
18:    /* Incomplete or cross checksum did not validate, loop again */
19:    ResponseSet := DO_READ(READ_PREV_REQUEST, LT_candidate)
20: end loop

DO_READ(READ_COMMAND, LT) :
1: for all Si ∈ {S_1, . . . ,S_N} do
2:     SEND(Si, READ_COMMAND, LT)
3: end for
4: ResponseSet := Ø
5: repeat
6:     Resp := RECEIVE(S, READ_RESPONSE)
7:     if (VALIDATE(Resp.D, Resp.CC, Resp.LT, S) = TRUE) then
8:         ResponseSet := ResponseSet ∪ Resp
9:     end if
10: until (UNIQUE_SERVERS(ResponseSet) = N - t)
11: RETURN(ResponseSet)

VALIDATE(D, CC, LT, S) :
1: if ((HASH(CC) ≠ LT.Verifier) OR (HASH(D) ≠ CC[S])) then
2:     RETURN(FALSE)
3: end if
4: RETURN(TRUE)
```

Figure 7.2: Read operation pseudo-code

7.4.4 Protocol Constraints

The symbol Q_C denotes a complete write operation: the threshold number of benign storage-nodes that must execute write requests for a write operation to be complete. To ensure that linearizability and wait-

freedom are achieved, Q_C and N must be constrained with regard to b, t, and each other. As well, the parameter m, used in DECODE, must be constrained. We present safety and liveness proofs for the protocol and discuss the concept of linearizability in the presence of Byzantine clients in [Goodson04b].

Write completion: To ensure that a correct client can complete a write operation,

$$Q_C \leq N - t - b. \quad (7.1)$$

Since slow storage-nodes cannot be differentiated from crashed storage-nodes, only $N - t$ responses can be awaited. As well, up to b responses received may be from Byzantine storage-nodes.

Read classification: To classify a candidate as complete, a candidate set of at least Q_C benign storage-nodes must be observed. In the worst case, at most b members of the candidate set may be Byzantine, thus,

$$|CandidateSet| - b - Q_C \text{, so } \texttt{COMPLETE} = Q_C + b. \quad (7.2)$$

To classify a candidate as incomplete a client must determine that a complete write does not exist in the system (i.e., fewer than Q_C benign storage-nodes host the write). For this to be the case, the client must have queried all possible storage-nodes (N_t), and must assume that nodes not queried host the candidate in consideration. So,

$$|CandidateSet| + t < Q_C \text{, so } \texttt{INCOMPLETE} = Q_C - \text{t}. \quad (7.3)$$

Real repairable candidates: To ensure that Byzantine storage-nodes cannot fabricate a repairable candidate, a candidate set of size b must be classifiable as incomplete. Substituting b into (3),

$$b + t < Q_C.. \quad (7.4)$$

Decodable repairable candidates: Any repairable candidate must be decodable. The lower bound on candidate sets that are repairable follows from (3) (since the upper bound on classifying a candidate as incomplete coincides with the lower bound on repairable):

$$1 \leq m \leq Q_C - t. \quad (7.5)$$

Constraint summary:

$$t + b + 1 \leq Q_C \leq N - t - b;$$

$$2t + 2b + 1 \leq N;$$

$$1 \leq m \leq Q_C - t.$$

7.5 Evaluation

This section evaluates the performance and scalability of the consistency protocol in the context of a prototype storage system called PASIS [Wylie00]. We compare the PASIS implementation of our protocol with the BFT library implementation [Castro] of Byzantine fault-tolerant replicated state machines [Castro02], since it is generally regarded as efficient.

7.5.1 PASIS Implementation

PASIS consists of clients and storage-nodes. Storage nodes store data-fragments and their versions. Clients execute the protocol to read and write data-items.

7.5.1.1 Storage-node Implementation

PASIS storage nodes use the Comprehensive Versioning File System (CVFS) [Soules03] to retain data-fragments and their versions. CVFS uses a log-structured data organization to reduce the cost of data versioning. Experience indicates that retaining every version and performing local garbage collection comes with minimal performance cost (a few percent) and that it is feasible to retain complete version histories for several days [Soules03, Strunk00].

We extended CVFS to provide an interface for retrieving the logical timestamp of a data-fragment. Each write request contains a data-fragment, a logical timestamp, and a cross checksum. To improve performance, read responses contain a limited version history containing logical timestamps of previously executed write requests. The version history allows clients to identify and classify additional candidates without issuing extra read requests. Storage-nodes can also return read responses that contain no data other than version histories, which makes candidate classification more network-efficient.

Pruning old versions, or garbage collection, is necessary to prevent capacity exhaustion of storage-nodes. A storage node in isolation, by the nature of the protocol, cannot determine which local data-fragment versions are safe to remove. An individual storage-node can garbage collect a data-fragment version if there exists a later complete write for the corresponding data-item. Storage-nodes are able to classify writes by executing the read consistency protocol in the same manner as the client. We discuss garbage collection more fully in [Goodson03].

7.5.1.2 Client Implementation

The client module provides a block-level interface to higher level software, and uses a simple RPC interface to communicate with storage-nodes. The RPC mechanism uses TCP/IP. The client module is responsible for the execution of the consistency protocol.

Initially, read requests are issued to $Q_C + b$ storage nodes. PASIS utilizes read witnesses to make read operations more network efficient; only m of the initial requests request the data-fragment, while all request version histories. If the read responses do not yield a candidate that is classified as complete, read requests are issued to the remaining storage-nodes (and a total of up to $N - t$ responses are awaited). If the initial candidate is classified as incomplete, subsequent rounds of read requests fetch only version histories until a candidate is classified as either repairable or complete. If necessary, after classification, extra data fragments are fetched according to the candidate timestamp. Once the data-item is successfully validated and decoded, it is returned to the client.

7.5.1.3 Mechanism Implementation

We measure the space-efficiency of an erasure code in terms of *blowup*—the total amount of data stored over the size of the data-item. We use an information dispersal algorithm [Rabin89], which has a blowup of $\frac{N}{m}$. Our information dispersal implementation stripes the data-item across the first m data-fragments (i.e., each data-fragment is $\frac{1}{m}$ of the original data-item's size). These *stripe-fragments* are used to generate the *code-fragments* via polynomial interpolation within a Galois Field. Our implementation of polynomial interpolation was originally based on publicly available code [Dai-a] for information dispersal [Rabin89]. We modified the source to make use of stripe-fragments and added an implementation of Galois Fields of size 2^8 that use lookup tables for multiplication. Our implementation of cross checksums closely follows [Gong89]. We use MD5 for all hashes; thus, each cross checksum is $N \times 16$ bytes long. Note, that if very small blocks are used with a large N, then the overhead due to size of the cross checksum could be substantial.

7.5.2 Experimental Setup

We use a cluster of 20 machines to perform our experiments. Each machine is a dual 1 GHz Pentium III machine with 384 MB of memory. Storage-nodes use a 9 GB Quantum Atlas 10K as the storage device. The machines are connected through a 100 Mb switch. All machines run the Linux 2.4.20 SMP kernel. In all experiments, clients keep a single read or write operation for a random 16 KB block outstanding. Once an operation completes, a new operation is issued (there is no think time). For all experiments, the working set fits into memory and all caches are warmed up beforehand.

7.5.2.1 PASIS Configuration

Each storage-node is configured with 128 MB of data cache, and no caching is done on the clients. All experiments show results using write-back caching at the storage nodes, mimicking availability of 16 MB of non-volatile RAM. This allows us to focus experiments on the overheads introduced by the protocol and not those introduced by the disk subsystem. All messages are authenticated using HMACs; pair-wise symmetric keys are distributed prior to each experiment.

7.5.2.2 BFT configuration

Operations in BFT [Castro02] require agreement among the replicas (storage-nodes in PASIS). BFT requires $N = 3b + 1$ replicas to achieve agreement. Agreement is performed in four steps: (i) the client broadcasts requests to all replicas; (ii) the *primary* broadcasts pre-prepare messages to all replicas; (iii) all replicas broadcast prepare messages to all replicas; and, (iv) all replicas send replies back to the client and then broadcast commit messages to all other replicas. Commit messages are piggybacked on the next pre-prepare or prepare message to reduce the number of messages on the network. *Authenticators*, lists of MACs, are used to ensure that broadcast messages from clients and replicas cannot be modified by a Byzantine replica. All clients and replicas have public and private keys that enable them to exchange symmetric cryptographic keys used to create MACs. Logs of commit messages are checkpointed (garbage collected) periodically.

An optimistic fast path for read-only operations is implemented in BFT. The client broadcasts its request to all replicas. Each replica replies once all previous requests have committed. Only one replica sends the full reply (i.e., the data and digest), and the remainder just send digests that can verify the correctness of the data returned. If the replies from replicas do not agree, the client re-issues the read operation. Re-issued read operations perform agreement using the base BFT algorithm.

The BFT configuration does not store data to disk; instead it stores all data in memory and accesses it via memory offsets. For all experiments, BFT view changes are suppressed. BFT uses UDP rather than TCP. The BFT implementation defaults to using IP multicast. In our environment, like many, IP multicast broadcasts to the entire subnet, thus making it unsuitable for shared environments. We found that the BFT implementation code is fairly fragile when using IP multicast in our environment, making it necessary to disable IP multicast in some cases (where stated explicitly). The BFT implementation authenticates broadcast messages via authenticators, and point-to-point messages with MACs.

7.5.3 Mechanism Costs

Client and storage-node computation costs for operations on a 16 KB block in PASIS are listed in Table 7.1. For every read and write operation, clients perform erasure coding (i.e., they compute $N - m$ data-fragments given m data-fragments), generate a cross checksum, and generate a verifier. Recall that writes generate the first m data fragments by striping the data-item into m fragments. Similarly, reads must generate $N - m$ fragments, from the m they have, in order to verify the cross checksum. Storage-nodes validate each write request they receive. This validation requires a comparison of the data-fragment's hash to the hash within the cross checksum, and a comparison of the cross checksum's hash to the verifier within the timestamp.

All requests and responses are authenticated via HMACS. The cost of authenticating write requests, listed in the table, is very small. The cost of authenticating read requests and timestamp requests are similar.

	b=1	b=2	b=3	b=4
Erasure coding	1250	1500	1730	1990
Cross checksum	360	440	480	510
Validate	82	58	48	40
Verifier	1.6	2.3	3.6	4.3
Authenticate	1.5	1.5	2.1	2.1

Table 7.1: Computation costs in PASIS in µs.

7.5.4 Performance and Scalability

7.5.4.1 Response Time

Figure 7.3 shows the mean response time of a single request from a single client as a function of the tolerated number of storage-node failures. Due to the fragility of the BFT implementation with $b > 1$, IP multicast was disabled for BFT during this experiment. The focus in this plot is the slopes of the response time lines: the flatter the line the more scalable the protocol is with regard to the number of faults tolerated. In our environment, a key contributor to response time is network cost, which is dictated by the space-efficiency of the protocol.

Figure 7.4 breaks down the mean response times of read and write operations, from Figure 7.3, into the costs at the client, on the network, and at the storage-node for $b = 1$ and $b = 4$. Since measurements are taken at the user-level, kernel-level timings for host network protocol processing (including network system calls) are attributed to the "network" cost of the breakdowns. To understand the response time measurements and scalability of these protocols, it is important to understand these breakdowns.

PASIS has better response times than BFT for write operations due to the space-efficiency of erasure codes and the nominal amount of work storage-nodes perform to execute write requests. For $b = 4$, BFT has a blowup of 13× on the network (due to replication), whereas our protocol has a blowup of $\frac{17}{5} = 3.4$ on the network. With IP multicast the response time of the BFT write operation would improve significantly, since the client would not need to serialize 13 replicas over its link. However, IP multicast does not reduce the aggregate server network utilization of BFT—for $b = 4$, 13 replicas must be delivered.

PASIS has longer response times than BFT for read operations. This can be attributed to two main factors: First, the PASIS storage-nodes store data in a real file system; since the BFT-based block store keeps all data in memory and accesses blocks via memory offsets, it incurs almost no server storage costs. We expect that a BFT implementation with actual data storage would incur server storage costs similar to PASIS (e.g., around 0.7 ms for a write and 0.4 ms for a read operation, as is shown for PASIS with $b = 1$ in Figure 7.4). Indeed, the difference in read response time between PASIS and BFT at $b = 1$ is mostly accounted for by server storage costs. Second, for our protocol, the client computation cost grows as t increases because the cost of generating data-fragments grows as N increases. In addition to the $b = t$ case,

Figure 7.3 shows one instance of PASIS assuming a hybrid fault model with $b = 1$. For space-efficiency, we set $m = t + 1$. Consequently, $Q_C = 2t + 1$ and $N = 3t + 2$. At $t = 1$, this configuration is identical to the Byzantine-only configuration. As t increases, this configuration is more space-efficient than the Byzantine-only configuration, since it requires $t - 1$ fewer storage-nodes. As such, the response times of read and write operations scale better.

7.5.4.2 Throughput

Figure 3.5 shows the throughput in 16 KB requests per second as a function of the number of clients (one request per client) for $b = 1$. In this experiment, BFT uses multicast, which greatly improves its network efficiency (BFT with multicast is stable for $b = 1$). PASIS was run in two configurations, one with the thresholds set to that of the minimum system with $m = 2$, $N = 5$ (write blowup of 2.5×), and one, more space-efficient, with $m = 3$, $N = 6$ (write blowup of 2×). Results show that throughput is limited by the server network bandwidth.

At high load, PASIS has greater write throughput than BFT. BFT's write throughput flattens out at 456 requests per second. We observed BFT's write throughput drop off as client load increased; likewise, we observed a large increase in request retransmissions. We believe that this is due to the use of UDP and a coarse grained retransmit policy in BFT's implementation. The write throughput of PASIS flattens out at 733 requests per second, significantly outperforming BFT. This is because of the network-efficiency of PASIS. Even with multicast enabled, each BFT server link sees a full 16 KB replica, whereas each PASIS server link sees $\frac{16}{m}$ KB. Similarly, due to network space-efficiency, the PASIS configuration using $m = 3$

outperforms the minimal PASIS configuration (954 requests per second). Both PASIS and BFT have roughly the same network utilization per read operation (16 KB per operation). To be network-efficient, PASIS uses read witnesses and BFT uses "fast path" read operations. However, PASIS makes use of more storage-nodes than BFT does servers. As such, the aggregate bandwidth available for reads is greater for PASIS than for BFT, and consequently PASIS has a greater read throughput than BFT. Although BFT could add servers to increase its read throughput, doing so would not increase its write throughput (indeed, write throughput would likely drop due to the extra inter-server communication).

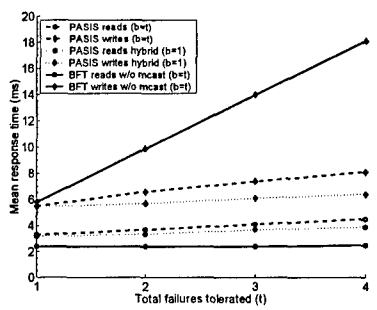


Figure 7.3: Mean Response time.

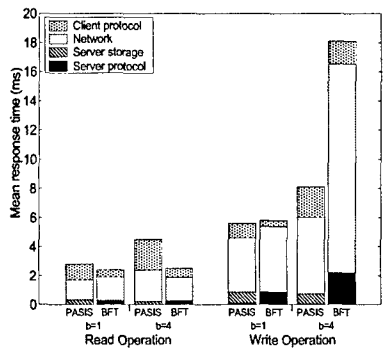


Figure 7.4: Response breakdown.

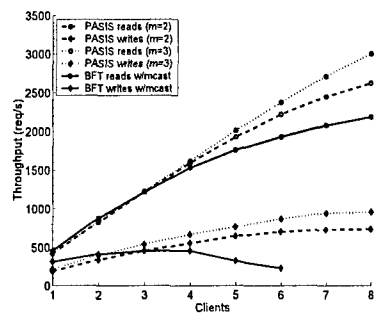


Figure 7.5: Throughput (b = 1).

7.5.4.3 Scalability Summary

For PASIS and BFT, scalability is limited by either the server network utilization or server CPU utilization. Figure 7.4 shows that PASIS scales better than BFT in both. Consider write operations. Each BFT server receives an entire replica of the data, whereas each PASIS storage-node receives a data-fragment $\frac{1}{m}$ the size of a replica. The work performed by BFT servers for each write request grows with b. In PASIS, the server protocol cost decreases from 90 µs for b = 1 to 57 µs for b = 4, whereas in BFT it increases from 0.80 ms to 2.1 ms. The cost in PASIS decreases because m increases as b increases, reducing the size of the data-fragment that is validated. We believe that the server cost for BFT increases because the number of messages that must be sent increases.

7.5.5 Concurrency

To measure the effect of concurrency on the system, we measure multi-client throughput of PASIS when accessing overlapping block sets. The experiment makes use of four clients, each with four operations outstanding. Each client accesses a range of eight data blocks, with no outstanding requests from the same client going to the same block. At the highest concurrency level (all eight blocks in contention by all clients), we observed neither significant drops in bandwidth nor significant increases in mean response time. Even at this high concurrency level, the initial candidate was classified as complete 89% of the time, otherwise classification required the traversal of history information. Of these history traversals, repair was only necessary a quarter of the time (i.e., 3% of all reads required repair). Since repair occurs so seldom, the effect on response time and throughput is minimal.

7.6 Lazy Verification in Byzantine Fault-Tolerant Distributed Storage Systems: Background

This section outlines the system model on which we focus, the protocol in which we develop lazy verification, and related work.

7.6.1 System Model and Failure Types

Survivable distributed storage systems tolerate client and storage-node faults by spreading data redundantly across multiple storage-nodes. We focus on distributed storage systems that can use erasure-coding schemes, as well as replication, to tolerate Byzantine failures [Lamport82] of both clients and storage-nodes. An m-of-N erasure-coding scheme (e.g., information dispersal [Rabin89]) encodes a data-item into N fragments such that any m allows reconstruction of the original. Generally speaking, primary goals for Byzantine fault-tolerant storage systems include data integrity, system availability, and efficiency (e.g., [Cachin05b, Castro02, Goodson04a]).

Data integrity can be disrupted by faulty storage-nodes and faulty clients. First, a faulty storage-node can corrupt the fragments/replicas that it stores, which requires that clients double-check integrity during reads. Doing so is straightforward for replication, since the client can just compare the contents (or checksums) returned from multiple storage-nodes. With erasure-coding, this is insufficient, but providing all storage-nodes with the checksums of all fragments (i.e., a cross checksum [Gong89]) allows a similar approach. Second, a faulty client can corrupt data-items with *poisonous writes*. A poisonous write operation [Martin02] gives incompatible values to some of the storage-nodes; for replication, this means non-identical values, and, for erasure-coding, this means fragments not correctly generated from the original data-item (i.e., such that different subsets of m fragments will reconstruct to different values). The result of a poisonous write is that different clients may observe different values depending on which subset of storage-nodes they interact with. Verifying that a write is not poisonous is difficult with erasure-coding, because one cannot simply compare fragment contents or cross checksums—one must verify that fragments sent to storage-nodes were correctly generated from the same data-item value.

Faulty clients can also affect availability and performance by *stuttering*. A stuttering client repeatedly sends to storage-nodes a number of fragments (e.g., $m-1$ of them) that is insufficient to form a complete write operation. Such behavior can induce significant overheads because it complicates verification and may create long sequences of work that must be performed before successfully completing a read.

Our work on lazy verification occurs in the context of a protocol that operates in an asynchronous timing model. But there is no correctness connection between the timing model and verification model. Asynchrony does increase the number of storage-nodes involved in storing each data-item, which in turn increases the work involved in verification and, thus, the performance benefits of lazy verification.

7.6.2 Read/write Protocol and Delayed Verification

This section begins a description of the concept of lazy verification in the context of the PASIS read/write protocol [Goodson04a, Wylie04]. This protocol uses versioning to avoid the need to verify completeness and integrity of a write operation during its execution. Instead, such verification is performed during read operations. Lazy verification shifts the work to the background, removing it from the critical path of both read and write operations in common cases.

The PASIS read/write protocol provides linearizable [Herlihy90] read and write operations on data blocks in a distributed storage system [Goodson04b]. It tolerates crash and Byzantine failures of clients, and operates in an asynchronous timing model. Point-to-point, reliable, authenticated channels are assumed. The protocol supports a hybrid failure model for storage-nodes: up to t storage-nodes may fail, $b \leq t$ of which may be Byzantine faults; the remainder are assumed to crash. For clarity of presentation, we focus exclusively on the fully Byzantine failure model here (i.e., $b = t$), which requires $N \geq 4b + 1$ storage-nodes. The minimal system configuration ($N = 4b + 1$) can be supported only when the reconstruction threshold m satisfies $m \leq b + 1$; in our experiments we will consider $m = b + 1$ only.

An overview of the PASIS read/write protocol, including the mechanisms for erasure codes, data-fragment integrity and write-operation integrity (timestamp validation, storage-node verification, etc.) may be found in Sections 7.4.1 and 7.4.2. Under this protocol, the majority of verification work is performed by the client during the read operation. The lazy verification approach developed in the remainder of this chapter addresses this issue by having the storage-nodes communicate, prior to the next read op-

eration if possible, to complete verification for the latest complete write. Each storage-node that observes the result of this verification can inform a client performing a read operation. If at least $b + 1$ confirm verification for the candidate, read-time verification becomes unnecessary. Section 7.7 details how lazy verification works and techniques for minimizing its performance impact.

7.6.3 Related Work

The notion of verifiability that we study is named after a similar property studied in the context of *m*-of-*N* *secret sharing*, i.e., that reconstruction from any *m* shares will yield the same value (e.g., [Feldman87, Pederson91]). However, to address the additional requirement of secrecy, these works employ more expensive cryptographic constructions that are efficient only for small secrets that, in particular, would be very costly if applied to blocks in a storage system. The protocols we consider here do not require secrecy, and so permit more efficient constructions. Most previous protocols perform verification proactively during write operations. When not tolerating Byzantine faults, two- or three-phase commit protocols are sufficient. For replicated data, verification can be made Byzantine fault-tolerant in many ways. For example, in the BFT system [Castro02], clients broadcast their writes to all servers, and then servers reach agreement on the hash of the written data value by exchanging messages. In other systems, such as Phalanx [Malkhi00], an "echo" phase like that of Rampart [Reiter95] is used: clients propose a value to collect signed server echos of that value; such signatures force clients to commit the same value at all servers. The use of additional communication and digital signatures may be avoided in Byzantine fault-tolerant replica systems if replicas are *self-verifying* (i.e., if replicas are identified by a collision-resistant hash of their own contents) [Martin02].

Verification of erasure-coded data is more difficult, as described earlier. The validating timestamps of the PASIS read/write protocol, combined with read-time or lazy verification, are one approach to addressing this issue. Cachin and Tessaro [Cachin05b] recently proposed an approach based on asynchronous verifiable information dispersal (AVID) [Cachin05a]. AVID's verifiability is achieved by having each storage-node send their data fragment, when it is received, to all other storage-nodes. Such network-inefficiency reduces the benefits of erasure-coding, but avoids all read-time verification. Lazy verification goes beyond previous schemes, for both replication and erasure-coding, by separating the effort of verifying from write and read operations. Doing so allows significant reductions in the total effort (e.g., by exploiting data obsolescence and storage-node cooperation) as well as shifting the effort out of the critical path. Unrelated to content verification are client or storage-node failures (attacks) that increase the size of logical timestamps. Such a failure could significantly degrade system performance, but would not affect correctness. Bazzi and Ding recently proposed a protocol to ensure non-skipping timestamps for Byzantine fault-tolerant storage systems [Bazzi04] by using digital signatures. Cachin and Tessaro [Cachin05b] incorporate a similar scheme based on a non-interactive threshold signature scheme. Validating timestamps do not ensure non-skipping timestamps, but we believe that lazy verification could be extended to provide bounded-skipping timestamps without requiring digital signatures. This will be an interesting avenue for future work.

7.7 Lazy Verification

An ideal lazy verification mechanism has three properties: (i) it is a background task that never impacts foreground read and write operations, (ii) it verifies values of write operations before clients read them (so that clients need not perform verification), and (iii) it only verifies the value of write operations that clients actually read. This sec- tion describes a lazy verification mechanism that attempts to achieve this ideal in the PASIS storage system. It also describes the use of lazy verification to bound the impact of Byzantine-faulty clients (Section 7.7.3) and its application to the garbage collection of unneeded versions (Section 7.7.4).

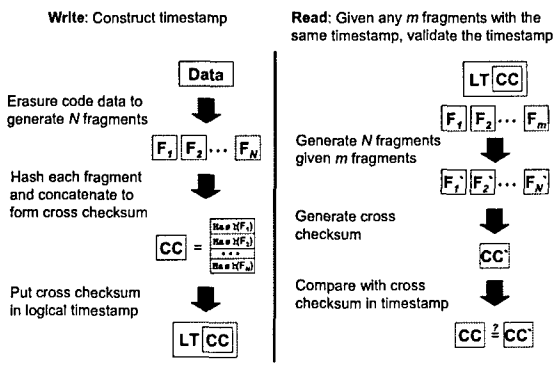


Figure 7.6: Illustration of the construction and validation of timestamps.

7.7.1 Lazy Verification Basics

Storage-nodes perform verification using a similar process to a read operation, as described in Section 7.6.2. First, a storage-node finds the latest complete write version. Second, the storage-node performs timestamp validation. If timestamp validation fails, the process is repeated: the storage-node finds the previous complete write version, and performs timestamp validation on that version. Figure 7.6 illustrates the construction and validation of a timestamp in the PASIS read/write protocol. Timestamp validation requires the storage-node to generate a cross checksum based on the erasure-coded fragments it reads. If the generated cross checksum matches that in the timestamp, then the timestamp validates.

Once a block version is successfully verified, a storage-node sets a flag indicating that verification has been performed. (A block version that fails verification is poisonous and is discarded.) This flag is returned to a reading client within the storage-node's response. A client that observes $b + 1$ storage-node responses that indicate a specific block version has been verified need not itself perform timestamp validation, since at least one of the responses must be from a correct storage-node.

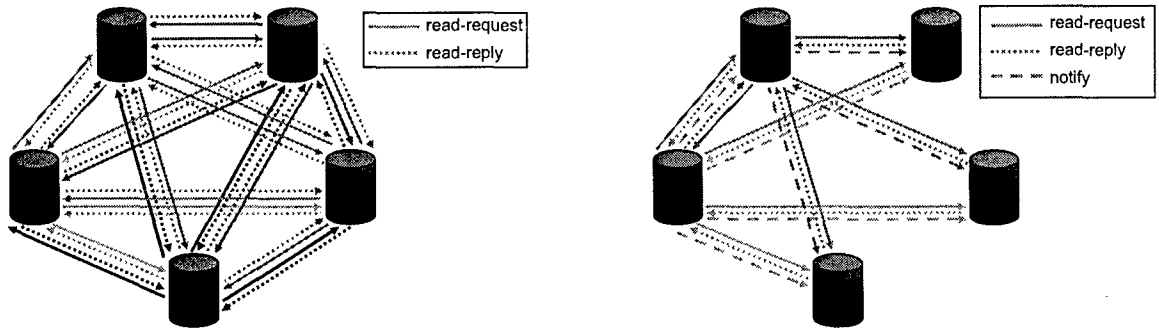
When possible, lazy verification is scheduled during idle time periods. Such scheduling minimizes the impact of verification on foreground requests. Significant idle periods exist in most storage systems, due to the bursty nature of storage workloads. Golding et al. [Golding95] evaluated various idle time detectors and found that a simple timer-based idle time detector accurately predicts idle time periods for storage systems. Another study showed that this type of idle time detector performs well even in a heavily loaded storage system [Blackwell95].

Although pre-read verification is the ideal, there is no guarantee that sufficient idle time will exist to lazily verify all writes prior to a read operation on an updated block. Then, there is the question of whether to let the client perform verification on its own (as is done in the original PASIS read/write protocol), or to perform verification *on demand*, prior to returning a read response, so that the client need not. The correct decision depends on the workload and current load. For example, in a mostly-read workload with a light system load, it is beneficial if the storage-nodes perform verification; this will save future clients from having to perform verification. In situations where the workload is mostly-write or the system load is high, then it is more beneficial if the clients perform verification; this increases the overall system throughput.

7.7.2 Cooperative Lazy Verification

Each storage-node can perform verification for itself. But, the overall cost of verification can be reduced if storage-nodes cooperate. As stated above, a client requires only $b + 1$ storage-nodes to perform lazy verification for it to trust the result. Likewise, any storage-node can trust a verification result confirmed by $b + 1$ other storage-nodes. Ideally, then, only $b + 1$ storage-nodes would perform lazy verification. Cooperative lazy verification targets this ideal.

With cooperative lazy verification, once a storage-node verifies a block version, it sends a *notify* message to the other storage-nodes. The notify message is a tuple of ⟨block number, timestamp, status⟩, where "status" indicates the result of the verification. Figures 7.7(a) and 7.7(b) illustrate the messages exchanged without and with cooperative lazy verification, respectively.



a) without cooperation

b) with cooperation

Figure 7.7: Communication pattern of lazy verification (a) without and (b) with cooperation.

We first describe the common-case message sequences for cooperative lazy verification, in concurrency- and fault-free operation. A single storage-node initiates lazy verification by performing verification and sending a notify message to the other storage-nodes. Another b storage-nodes then perform verification and send notify messages to the remaining storage-nodes. The remaining storage-nodes will thus receive $b + 1$ identical notify messages, allowing them to trust the notify messages, since at least one storage-node must be correct. To reduce the number of messages required and to distribute the work of performing verification among storage-nodes, each storage-node is responsible for leading the cooperative lazy verification of a distinct range of block numbers.

If a storage-node needs to verify a block before enough notify messages are received, it will perform verification (and send notify messages), even for blocks that are outside the range it is normally responsible for. In the event of faulty storage-nodes or concurrency, the notification messages may not match. The remaining storage-nodes will then perform verification and send notify messages to the other storage-nodes, until all storage-nodes have either performed verification or received identical $b + 1$ notify messages. In the worst case, all storage-nodes will perform verification.

The benefit of cooperative lazy verification is a reduction in the number of verification-related messages. Without cooperation, each storage-node must independently perform verification (i.e., effectively perform a read operation). This means each of the N storage-nodes sends read requests to, and receives read responses from, $N - b$ storage-nodes; this yields $O(N^2)$ messages without cooperation. In contrast, the common case for cooperative lazy verification is for just $b + 1$ storage-nodes to perform read requests (to $N - b$ storage-nodes) and then send $N - 1$ notify messages. Even in the worst case in which $2b + 1$ must perform read requests, the communication complexity is still $O(bN)$. For example, even in a minimal system configu ration (N = 4b+1) with $b = 1$, cooperation saves 33% of messages (20 messages with cooperation, versus 30 without), and this benefit improves to over 50% (128 vs. 260) at $b = 3$. Note that storage-nodes that perform cooperative lazy verification need not send notify messages back to storage-nodes from which they received a notify message.

7.7.3 Garbage Collection

If lazy verification passes for a block version, i.e., the block version is complete and its timestamp validates, then block versions with earlier timestamps are no longer needed. The storage-node can delete such obsolete versions. This application of lazy verification is called garbage collection. Garbage collection allows memory and storage capacity at storage-nodes to be reclaimed by deleting unnecessary versions.

In the original presentation of the PASIS read/write protocol, capacity is assumed to be unbounded. In practice, capacity is bounded, and garbage collection is necessary. We distinguish between useful storage, which is the user-visible capacity (i.e., number of blocks multiplied by fragment size), and the history pool. The *history pool* is the capacity used for possible versions. Since the useful storage capacity must usually be determined at system configuration time (e.g., during FORMAT), the maximum size of the his-

tory pool will usually be fixed (at raw storage-node capacity minus useful storage capacity). Thus, whenever the history pool's capacity is exhausted, the storage-node must perform garbage collection (i.e., verification plus space reclamation) before accepting new write requests.

Capacity bounds, garbage collection, and liveness. Capacity bounds and garbage collection can impact the liveness semantic of the PASIS read/write protocol. Without capacity bounds, reads and writes are wait-free [Herlihy91]. With a capacity bound on the history pool, however, unbounded numbers of faulty clients could collude to exhaust the history pool with incomplete write operations that cannot be garbage collected—this effectively denies service. This possibility can be eliminated by bounding the number of clients in the system and setting the history pool size appropriately—the history pool size must be the product of the maximum number of clients and the per-client possible version threshold. With this approach, a faulty client can deny service to itself but not to any other client.

Garbage collection itself also affects the liveness semantic, as it can interact with reads in an interesting manner. Specifically, if a read is concurrent to a write, and if garbage collection is concurrent to both, then it is possible for the read not to identify a complete candidate. For example, consider a read concurrent to the write of version-3 with version-2 complete and all other versions at all storage-nodes garbage collected. It is possible for the read to classify version-3 as incomplete, then for the write of version-3 to complete, then for garbage collection to run at all storage-nodes and delete version-2, and finally for the read to look for versions prior to version-3 and find none. Such a read operation must be retried. Because of this interaction, the PASIS read/write protocol with garbage collection is obstruction-free [Herlihy03] (assuming an appropriate history pool size) rather than wait-free.

Performing garbage collection. Garbage collection and lazy verification are tightly intertwined. An attempt to verify a possible version may be induced by history pool space issues, per-client-per-block and per-client thresholds, or the occurrence of a sufficient idle period. Previous versions of a block may be garbage-collected once a later version is verified.

Cooperative lazy verification is applicable to garbage collection. If a storage-node receives notify messages from $b + 1$ storage-nodes, agreeing that verification passed for a given version of a given block, then the storage-node can garbage collect prior versions of that block. Of course, a storage-node may receive $b + 1$ notify messages for a given block that have different timestamp values, if different storage-nodes classify different versions as the latest complete write. In this case, the timestamps are sorted in descending order and the $b + 1^{st}$ timestamp used for garbage collection (i.e., all versions prior to that timestamp may be deleted). This is safe because at least one of the notify messages must be from a correct storage-node.

There is an interesting policy decision regarding garbage collection and repairable candidates. If a client performing a read operation encounters a repairable candidate, it must perform repair and return it as the latest complete write. However, a storage-node performing garbage collection does not have to perform repair. A storage-node can read prior versions until it finds a complete candidate and then garbage collect versions behind the complete candidate. Doing so is safe, because garbage collection "reads" do not need to fit into the linearizable order of operations. By not performing repair, storage-nodes avoid performing unnecessary work when garbage collection is concurrent to a write. On the other hand, performing repair might enable storage-nodes to discard one more block version.

Prioritizing blocks for garbage collection. Careful selection of blocks for verification can improve efficiency, both for garbage collection and lazy verification. Here, we focus on two complementary metrics on which to prioritize this selection process: number of versions and presence in cache.

Since performing garbage collection (lazy verification) involves a number of network messages, it is beneficial to amortize the cost by collecting more than one block version at a time. Each storage-node remembers how many versions it has for each block. By keeping this list sorted, a storage-node can efficiently identify its high-yield blocks: blocks with many versions. Many storage workloads contain blocks that receive many over-writes [Ruemmler03a, Ruemmler03b], which thereby become high-yield blocks. When performing garbage collection, storage-nodes prefer to select *high-yield* blocks. In the common

case, all but one of the block's versions will have timestamps less than the latest candidate that passes lazy verification and hence can be deleted (and never verified). Selecting high-yield blocks amortizes the cost of verification over many block versions. This is particularly important when near the per-client possible version threshold or the history pool size, since it minimizes the frequency of on-demand verification.

Block version lifetimes are often short, either because of rapid overwrites or because of create-delete sequences (e.g., temporary files). To maximize the value of each verification operation, storage-nodes delay verification of recently written blocks. Delaying verification is intended to allow rapid sequences to complete, avoiding verification of short-lived blocks and increasing the average version yield by verifying the block once after an entire burst of over-writes. As well, such a delay reduces the likelihood that verification of a block is concurrent with writes to the block. Running verification concurrent to a write, especially if only two local versions exist at a storage-node, may not yield any versions to garbage collect.

Storage-nodes prefer to verify versions while they are in the write-back cache, because accessing them is much more efficient than when going to disk. Moreover, if a block is in one storage-node's cache, it is likely to be in the caches of other storage-nodes, and so verification of that block is likely to not require disk accesses on the other storage-nodes. Garbage collecting versions that are in the write-back cache, before they are sent to the disk, is an even bigger efficiency boost. Doing so eliminates both an initial disk write and all disk-related garbage collection work. Note that this goal matches well with garbage collecting high-yield blocks, if the versions were created in a burst of over-writes.

In-cache garbage collection raises an interesting possibility for storage-node implementation. If only complete writes are sent to disk, which would restrict the history pool size to being less than the cache size, one could use a non-versioning on-disk organization. This is of practical value for implementers who do not have access to an efficient versioning disk system implementation. The results from Section 7.9.3 suggest that this is feasible with a reasonably large storage-node cache (e.g., 500MB or more).

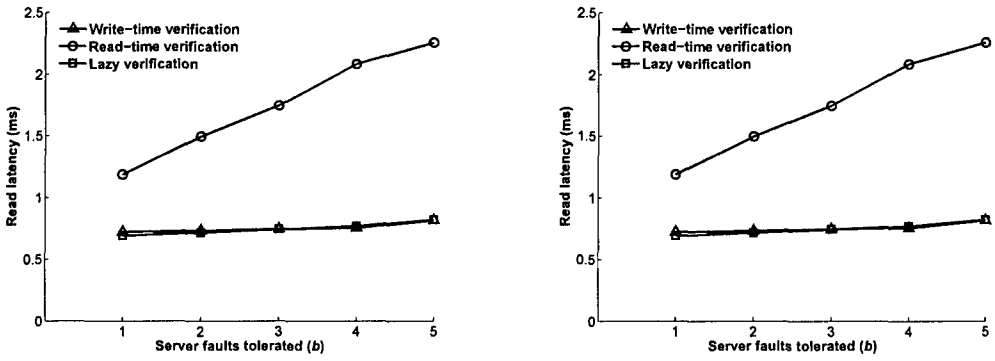
7.8 Implementation

To enable experimentation, we have added lazy verification to the PASIS storage system implementation [Goodson04a]. PASIS consists of storage-nodes and clients. Storage-nodes store fragments and their versions. Clients execute the protocol to read and write blocks. Clients communicate with storage-nodes via a TCP-based RPC interface.

Storage-nodes. Storage-nodes provide interfaces to write a fragment at a logical time, to query the greatest logical time, to read the fragment version with the greatest logical time, and to read the fragment with the greatest logical time before some logical time. With lazy verification, storage-nodes do not return version history or fragments for writes that have been classified as poisonous. In the common case, a client requests the fragment with the greatest logical time. If a client detects a poisonous or incomplete write, it reads the previous version history and earlier fragments.

Each write request creates a new version of the fragment (indexed by its logical timestamp) at the storage-node. The storage-node implementation is based on the S4 object store [Soules03, Strunk00]. A log-structured organization [Rosenblum92] is used to reduce the disk I/O cost of data versioning. Multi-version b-trees [Becker96, Soules03] are used by the storage-nodes to store fragments; all fragment versions are kept in a single b-tree indexed by a 2-tuple ⟨*blocknumber*, *timestamp*⟩. The storage-node uses a write-back cache for fragment versions, emulating non-volatile RAM.

We extended the base PASIS storage-node to perform lazy verification. Storage-nodes keep track of which blocks have possible versions and perform verification when they need the history pool space, when a possible version threshold is reached, or when idle time is detected. If verification is induced by a per-client-per-block possible version threshold, then that block is chosen for verification. Otherwise, the storage-node's *high-write-count* table is consulted to select the block for which verification is expected to eliminate the highest number of versions. The high-write-count table lists blocks for which the storage-node is *responsible* in descending order of the number of unverified versions associated with each.



a) read latency

b) write latency

Figure 7.8: Operation latencies for different verification policies.

Each storage-node is assigned verification responsibility for a subset of blocks in order to realize cooperative lazy verification. Responsibility is assigned by labeling storage-nodes from $0...(N-1)$ and by having only storage-nodes k **mod** N, $(k+1)$ **mod** N, $\cdots$, $(k+b)$ **mod** N be responsible for block k. In the event of failures, crash or Byzantine, some storage-nodes that are responsible for a particular block may not complete verification. Therefore, any storage-node will perform verification if it does not receive sufficient notify messages before they reach a possible version threshold or exceed history pool capacity. So, while $b + 1$ matching notify messages about a block will allow a storage-node to mark it as verified, failure to receive them will affect only performance.

Client module. The client module provides a block-level interface to higher-level software. The protocol implementation includes a number of performance enhancements that exploit its threshold nature. For example, to improve the responsiveness of write operations, clients return as soon as the minimum number of required success responses is received; the remainder of the requests complete in the background. To improve the read operation's performance, only m read requests fetch the fragment data and version history; the remaining requests only fetch version histories. This makes the read operation more network-efficient. If necessary, after classification, extra fragments are fetched according to the candidate's timestamp.

PASIS supports both replication and an m-of-N erasure coding scheme. If $m = 1$, then replication is employed. Otherwise, our base erasure code implementation stripes the block across the first m fragments; each stripe-fragment is 1 m the length of the original block. Thus, concatenation of the first m fragments produces the original block. (Because "decoding" with the m stripe-fragments is computationally less expensive, the implementation always tries to read from the first m storage-node for any block.) The stripe-fragments are used to generate the $N - m$ code-fragments, via polynomial interpolation within a Galois Field. The implementation of polynomial interpolation is based on publicly available code [Dai-b] for information dispersal [Rabin89]. As detailed in Section 7.5, this code was modified to make use of stripe-fragments and to add an implementation of Galois Fields of size 2^8 that use lookup tables for multiplication. MD5 is used for all hashes; thus, each cross checksum is $N \times 16$ bytes long.

We extended the PASIS client module to use the flag described in Section 7.7.1 to avoid the verification step normally involved in every read operation, when possible. It checks the "verified" flag returned from each contacted storage-node and does its own verification only if fewer than $b + 1$ of these flags are set. In the normal case, these flags will be set and read-time verification can be skipped.

7.9 Evaluation

This section quantifies benefits of lazy verification. It shows that significant increases in write throughput can be realized with reasonably small history pool sizes. In fact, lazy verification ap-

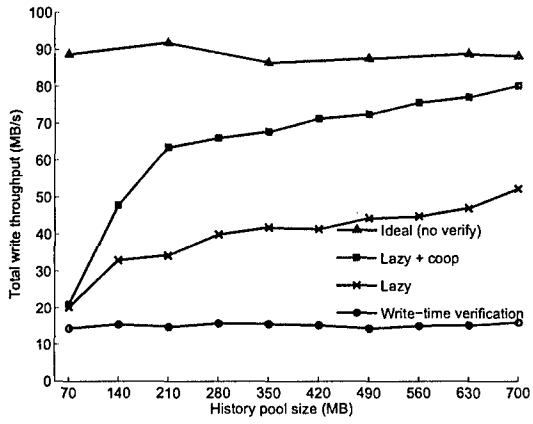


Figure 7.9: Write throughput, as a function of history pool size, for four verification policies.

proaches the ideal of zero performance cost for verification. It also confirms that the degradation-of-service vulnerability inherent to delayed verification can be bounded with minimal performance impact on correct clients. Experimental Setup

All experiments are performed on a collection of Intel Pentium 4 2.80GHz computers, each with 1GB of memory and an Intel PRO/1000 NIC. The computers are connected via an HP ProCurve Switch 4140gl specified to perform 18.3Gbps/35.7mpps. The computers run Linux kernel 2.6.11.5 (Debian 1:3.3.4-3).

Micro-benchmark experiments are used to focus on performance characteristics of lazy verification. Each experiment consists of some number of clients performing operations on a set of blocks. Experiments are run for 40 seconds and measurements are taken after 20 seconds. (The 20 second warm-up period is sufficient to ensure steady-state system operation.) For the benchmarks used in this section, the working set and history pool are sized to fit in the storage-nodes' RAM caches. Given that the storage-node cache is assumed to be non-volatile, this eliminates disk activity and allows focus to stay on the computation and networking costs of the protocol.

7.9.1 Verification Policies and Operation Latencies

A storage system's verification policy affects client read and write operation latencies. We ran an experiment to measure the operation latencies for three different verification policies: proactive write-time verification, read-time verification, and lazy verification. The write-time verification policy is emulated by having each storage-node perform verification for a block immediately after it accepts a write request—this is similar to the system described by Cachin and Tessaro [Cachin05b]. With the read-time verification policy, the client incurs the cost of verification on every read operation.

We ran experiments for different numbers of tolerated server faults (from $b = 1$ to $b = 5$). For each experiment, we used the minimal system configuration: $N = 4b + 1$. The erasure coding reconstruction threshold m equals $b + 1$ in each experiment. This provides the maximal space- and network-efficiency for the value of N employed. A single client performs one operation at a time. The client workload is an equal mix of read and write operations. After each operation, the client sleeps for 10ms. This introduces idle time into the workload.

Figure 7.8(a) shows the client write latency for the different verification policies. The write-time verification policy has a higher write latency then the two other verification policies. This is because the storage-nodes perform verification in the critical path of the write operation. For both the read-time and lazy verification policies, the client write latency increases slightly as more faults are tolerated. This is due to the increased computation and network cost of generating and sending fragments to more servers.

Figure 7.8(b) shows the client read latency for the different verification policies. For the read-time verification policy, the client read latency increases as more faults are tolerated. This is because the computation cost of generating the fragments to check the cross checksum and validate the timestamp increases as b increases. For the write-time verification policy, verification is done at write-time, and so does not affect read operation latency. For the lazy verification policy, sufficient idle time exists in the workload for servers to perform verification in the background. As such, the read latency for the lazy verification policy follows that for the write-time verification policy.

These experiments show that, in workloads with sufficient idle time, lazy verification removes the cost of verification from the critical path of client read and write operations. Lazy verification achieves the fast write operation latencies associated with read-time verification, yet avoids the overhead of client verification for read operations.

7.9.2 Impact on Foreground Requests

In many storage environments, bursty workloads will provide plenty of idle time to allow lazy verification and garbage collection to occur with no impact on client operation performance. To explore bounds on the benefits of lazy verification, this section evaluates lazy verification during non-stop high-load with no idle time. When there is no idle time, and clients do not exceed the per-client or per-block-per-client thresholds, verification is induced by history pool exhaustion. The more efficient verification and garbage collection are, the less impact there will be on client operations.

In this experiment, four clients perform write operations on 4096 32KB blocks, each keeping eight operations in progress at a time and randomly selecting a block for each operation. Also, the system is configured to use $N = 5$ storage-nodes, while tolerating one Byzantine storage-node fault ($b = 1$) and employing 2-of-5 erasure-coding (so, each fragment is 16KB in size, and the storage-node working set is 64MB).

Figure 7.9 shows the total client write throughput, as a function of history pool size, with different verification policies. The top and bottom lines correspond to the performance ideal (zero-cost verification) and the conventional approach of performing verification during each write operation, respectively. The ideal of zero-cost verification is emulated by having each storage-node replace the old version with the new without performing any verification at all. As expected, neither of these lines is affected by the history pool size, because versions are very short-lived for these schemes. Clearly, there is a significant performance gap (5×) between the conventional write-time verification approach and the ideal.

The middle two lines correspond to use of lazy verification with ("Lazy + coop") and without ("Lazy") cooperative lazy verification. Three points are worth noting. First, with lazy verification, client write throughput grows as the history pool size grows. This occurs because a larger history pool allows more versions of each block to accumulate before history pool space is exhausted. As a result, each verification can be amortized over a larger number of client writes. (Recall that all earlier versions can be garbage collected once a later version is verified.) Second, with a reasonably-sized 700MB history pool size, cooperative lazy verification provides client write throughput within 9% of the ideal. Thus, even without idle time, lazy verification eliminates most of the performance costs of verification and garbage collection, providing a factor of four performance increase over conventional write-time verification schemes. Third, cooperative lazy verification significantly reduces the impact of verification, increasing client write throughput by 53–86% over non-cooperative lazy verification for history pool sizes over 200MB.

7.9.3 Summary

Lazy verification can significantly improve the performance of Byzantine fault-tolerant distributed storage systems that employ erasure-coding. It shifts the work of verification out of the critical path of client operations and allows significant amortization of work. Measurements show that, for workloads with idle periods, the cost of verification can be hidden from both the client read and write operation. In workloads without idle periods, lazy verification and its concomitant techniques—storage-node cooperation and prioritizing the verification of high-yield blocks—provides a factor of four greater write bandwidth than a conventional write-time verification strategy.

8 REFERENCES

[3Com01a] 3Com. 3Com Embedded Firewall Architecture for E-Business. Technical Brief 100969-001. 3Com Corporation, April 2001.

[3Com01b] 3Com. Administration Guide, Embedded Firewall Software. Documentation. 3Com Corporation, August 2001.

[Abd-El-Malek05] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, Jay J. Wylie. Lazy Verification in Fault-Tolerant Distributed Storage Systems. 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005), October 26-28, 2005, Orlando, Florida.

[Aguilera03] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, Dave Andersen, Mike Burrows Timothy Mann, and Chandramohan A. Thekkath. Blocklevel security for network-attached disks. Conference on File and Storage Technologies (San Francisco, CA, 31 March–02 April 2003), pages 159–174. USENIX Association, 2003.

[Alexander99] D. Scott Alexander, Kostas G. Anagnostakis, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. The price of safety in an active network. MS–CIS–99–04. Department of Computer and Information Science, University of Pennsylvania, 1999.

[Ammann00] Paul Ammann, Sushil Jajodia, and Peng Liu. Rewriting histories: recovering from undesirable committed transactions. Distributed and Parallel Databases, 8(1):7–40. Kluwer Academic Publishers, January 2000.

[Anderson95] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). IEEE Micro, 15(1):54–64, February 1995.

[Arbaugh97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A secure and reliable bootstrap architecture. IEEE Symposium on Security and Privacy (Oakland, CA, 4–7 May 1997), pages 65–71. IEEE Computer Society Press, 1997.

[Axelsson98] S. Axelsson. Research in intrusion-detection systems: a survey. Technical report 98–17. Department of Computer Engineering, Chalmers University of Technology, December 1998.

[Bailey75] N. Bailey. The Mathematical Theory of Infectious Diseases and its Applications. Griffin, London, 1975.

[Baker91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. ACM Symposium on Operating System Principles. Published as Operating Systems Review, 25(5):198–212, 13– 16 October 1991.

[Bartal99] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. IEEE Symposium on Security and Privacy, pages 17–31, 1999.

[Bazzi04] R. A. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. DISC, 2004.

[Becker96] B. Becker, S. Gschwind, T. Ohler, P. Widmayer, and B. Seeger. An asymptotically optimal multiversion b-tree. Very large data bases journal, 5(4):264–275, 1996.

[Bellare96] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. Advances in Cryptology - CRYPTO, pages 1–15. Springer-Verlag, 1996.

[Bishop96] M. Bishop and M. Dilger. Checking for race conditions in file accesses. Computing Systems, 9(2):131–152, Spring 1996.

[Blackwell95] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. Annual USENIX Technical Conference (New Orleans), pages 277-288. Usenix Association, 16-20 January 1995.

[Burns96] R. C. Burns. Differential compression: a generalized solution for binary files. Masters thesis. University of California at Santa Cruz, December 1996.

[Burrows92] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. Architectural Support for Programming Languages and Operating Systems (Boston, MA, 12-15 October 1992). Published as Computer Architecture News, 20(special issue):2-9, October 1992.

[Burrows94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 124. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 10 May 1994.

[Cachin05a] C. Cachin and S. Tessaro. Asynchronous verifiable infromation dispersal. Symposium on Reliable Distributed Systems. IEEE, 2005.

[Cachin05b] C. Cachin and S. Tessaro. Brief announcement: Optimal resilience for erasure-coded Byzantine distributed storage. International Symposium on Distributed Computing. Springer, 2005.

[Card94] R'emy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the Second Extended Filesystem. First Dutch International Symposium on Linux (Amsterdam, The Netherlands, December 1994), 1994.

[Castro] M. Castro and R. Rodrigues. BFT library implementation. http://www.pmg.lcs.mit.edu/bft/#sw.

[Castro00] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. Symposium on Operating Systems Design and Implementation, pages 273–287. USENIX Association, 2000.

[Castro01] M. Castro and B. Liskov. Byzantine fault tolerance can be fast. Dependable Systems and Networks, pages 513–518, 2001.

[Castro02] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems, 20(4):398-461, November 2002.

[CERT01a] CERT. CERT Advisory CA-2001-19 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL, July 19, 2001. http://www.cert.org/advisories/CA-2001-19.html.

[CERT01b] CERT. CERT Advisory CA-2001-26 Nimda Worm, September 18, 2001. http://www.cert.org/advisories/CA-2001-26.html.

[CERT03] CERT. CERT Advisory CA-2003-04 MS-SQL Server Worm, January 25, 2003. http://www.cert.org/advisories/CA-2003-04.html.

[Chen01] P. M. Chen and B. D. Noble. When virtual is better than real. Hot Topics in Operating Systems, pages 133–138. IEEE Comput. Soc., 2001.

[ChenS04] Shigang Chen and Yong Tang. Slowing down internet worms. In Proceedings of 24th International Conference on Distributed Computing Systems, Tokyo, Japan, March 2004.

[ChenZ03] Z. Chen, L. Gao, and K. Kwiat. Modeling the spread of active worms. In Proceedings of IEEE INFOCOM 2003, San Francisco, CA, April 2003.

[Cheswick94] B. Cheswick and S. Bellovin. Firewalls and Internet security: repelling the wily hacker. Addison-Wesley, Reading, Mass. and London, 1994.

[Cooper90] Eric Cooper, Peter Steenkiste, Robert Sansom, and Brian Zill. Protocol implementation on the Nectar communication processor. ACM SIGCOMM Conference (Philadelphia, PA), September 1990.

[Crispin96] M. Crispin. Internet message access protocol – version 4rev1, RFC–2060. NetworkWorking Group, December 1996.

[Dai-a] W. Dai. Crypto++. http://cryptopp.sourceforge.net/docs/ref/.

[Dai-b] W. Dai. Crypto++. http://www.cryptopp.com/.

[Dalton93] Chris Dalton, GregWatson, David Banks, Costas Calamvokis, Aled Edwards, and John Lumley. Afterburner. IEEE Network, 7(4):36–43, July 1993.

[Decasper99] Dan S. Decasper, Bernhard Plattner, Guru M. Parulkar, Sumi Choi, John D. DeHart, and TilmanWolf. A scalable high-performance active network node. IEEE Network, 13(1):8–19. IEEE, January–February 1999.

[Denning87] D. Denning. An intrusion-detection model. IEEE Transactions on Software Engineering, SE-13(2):222–232, February 1987.

[Denning99] D. E. Denning. Information warfare and security. Addison-Wesley, 1999.

[Donoho02] David L. Donoho, Ana Georgina Flesia, Umesh Shankar, Vern Paxson, Jason Coit, and Stuart Staniford. Multiscale stepping-stone detection: detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. RAID (Zurich, Switzerland, 16–18 October 2002), 2002.

[Douceur99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Atlanta, GA, 1–4 May 1999). Published as ACM SIGMETRICS Performance Evaluation Review, 27(1):59–70. ACM Press, 1999.

[Ellard03] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of an Email and Research Workload. Conference on File and Storage Technologies. USENIX Association, 2003.

[EMC-McAfee02] McAfee NetShield for Celerra. EMC Corporation, August 2002. http://www.emc.com/pdf/partnersalliances/einfo/McAfee_netshield.pdf.

[Fall02] K. Fall and K. Varadhan, editors. The ns Manual. The VINT Project. UC Berkeley, LBL, USC/ISI, and Xerox PARC, 14 April 2002. World Wide Web, http://www.isi.edu/nsnam/ns/doc/. Ongoing.

[Farmer00a] Dan Farmer. What are MACtimes? Dr. Dobb's Journal, 25(10):68–74, October 2000.

[Farmer00b] Dan Farmer and Wietse Venema. Forensic computer analysis: an introduction. Dr. Dobb's Journal, 25(9):70–75, September 2000.

[Farmer01] Dan Farmer. Bring out your dead: the ins and outs of data recovery. Dr. Dobb's Journal, 26(1):102–108, January 2001.

[Feldman87] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. IEEE Symposium on Foundations of Computer Science, pages 427–437. IEEE, 1987.

[Fischer85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2):374–382. ACM Press, April 1985.

[Fiuczynski98] Marc E. Fiuczynski, Brian N. Bershad, Richard P. Martin, and David E. Culler. SPINE: an operating system for intelligent network adaptors. UW TR-98-08-01. 1998.

[Forrest96] Stephanie Forrest, Setven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for UNIX processes. IEEE Symposium on Security and Privacy (Oakland, CA, 6–8 May 1996), pages 120–128. IEEE, 1996.

[Freed96] N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part one: format of internet message bodies, RFC–2045. Network Working Group, November 1996.

[Friedman00] David Friedman and David F. Nagle. Building Scalable Firewalls with Intelligent Network Interface Cards. CMUCS- 00-173. Technical Report, Carnegie Mellon Univeristy School of Computer Science, December 2000.

[Friedman01] David Friedman and David Nagle. Building Firewalls with Intelligent Network Interface Cards. Technical Report CMU–CS–00–173. CMU, May 2001.

[Frolund03] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: enterprise storage systems on a shoestring. Hot Topics in Operating Systems, pages 133–138. USENIX Association, 2003.

[Ganger00] Gregory R. Ganger and David F. Nagle. Enabling Dynamic Security Management of via Device-Embedded Security. CMU SCS Technical Report CMU-CS-00-174, December 2000.

[Ganger01] G. R. Ganger and D. F. Nagle. Better security via smarter devices. Hot Topics in Operating Systems, pages 100–105. IEEE, 2001.

[Ganger02] Gregory R. Ganger, Gregg Economou, and Stanley M. Bielski. Self-securing network interfaces: what, why and how. CMU-CS 02-144. August 2002.

[Ganger03a] G. R. Ganger, G. Economou, and S. M. Bielski. Finding and Containing Enemies Within the Walls with Self-securing Network Interfaces. Carnegie Mellon University Technical Report CMU-CS-03-109. January 2003.

[Ganger03b] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. Self-* Storage: brick-based storage with automated administration. Technical Report CMU-CS-03-178. Carnegie Mellon University, August 2003.

[Garfinkel03] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. NDSS. The Internet Society, 2003.

[Gibson99] Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst, and Jim Zelenka. NASD scalable storage systems. USENIX.99 (Monterey, CA., June 1999), 1999.

[Gobioff99] H. Gobioff. Security for a high performance commodity storage subsystem. PhD thesis, published as TR CMU–CS–99–160. Carnegie-Mellon University, Pittsburgh, PA, July 1999.

[Golding95] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J.Wilkes. Idleness is not sloth. Winter USENIX Technical Conference, pages 201–212. USENIX Association, 1995.

[Gong89] L. Gong. Securely replicating authentication services. International Conference on Distributed Computing Systems, pages 85–91. IEEE Computer Society Press, 1989.

[Goodson02] Garth Goodson, Jay Wylie, Greg Ganger & Mike Reiter. Decentralized Storage Consistency via Versioning Servers. Carnegie Mellon University Technical Report CMU-CS-02-180, September 2002.

[Goodson03] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. Technical report CMU-PDL-03-104. CMU, December 2003.

[Goodson04a] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, Michael K. Reiter. Efficient Byzantine-tolerant Erasure-coded Storage. Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004). Palazzo dei Congressi, Florence, Italy. June 28th - July 1, 2004.

[Goodson04b] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. The safety and liveness properties of a protocol family for versatile survivable storage infrastructures. Technical report CMU–PDL–03–105. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[Griffin03] John Linwood Griffin, Adam Pennington, John S. Bucy, Deepa Choundappan, Nithya Muralidharan, Gregory R. Ganger. On the Feasibility of Intrusion Detection Inside Workstation Disks. Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-03-106. December, 2003.

[Grugg02] The Grugg. Defeating forensic analysis on Unix. Phrack Magazine, 11(59):6–6, 28, July 2002.

[Grune] D. Grune, B. Berliner, and J. Polk. Concurrent Versioning System, http://www.cvshome.org/.

[Hagmann87] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. ACM Symposium on Operating System Principles (Austin, Texas, 8-11 November 1987). Published as Operating Systems Review, 21(5):155-162, November 1987.

[Handley01] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. USENIX Security Symposium (Washington, DC, 13–17 August 2001), pages 115–131. USENIX Association, 2001.

[Herlihy87] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. Advances in Cryptology - CRYPTO, pages 379–391. Springer-Verlag, 1987.

[Herlihy90] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3):463–492, 1990.

[Herlihy91] M. Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages, 13(1):124–149. ACM Press, 1991.

[Herlihy03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. International Conference on Distributed Computing Systems, pages 522-529. IEEE, 2003.

[Hitz90] David Hitz, Guy Harris, James K. Lau, and Allan M. Schwartz. Using Unix as one component of a lightweight distributed kernel for multiprocessor file servers. Winter USENIX Technical Conference (Washington, DC), 23-26 January 1990.

[Hitz94] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. Winter USENIX Technical Conference (San Francisco, CA). Published as Proceedings of USENIX, pages 235-246. USENIX Association, 19 January 1994.

[Hogwash] Hogwash. Inline packet scrubber. http://sourceforge.net/projects/hogwash

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1):51-81, February 1988.

[Huang96] Y. N. Huang, C. M. R. Kintala, L. Bernstein, and Y. M. Wang. Components for software fault-tolerance and rejuvenation. AT&T Bell Laboratories Technical Journal, 75(2):29–37, March-April 1996.

[Hutchinson99] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. Symposium on Operating Systems Design and Implementation (New Orleans, LA, 22–25 February 1999), pages 239–249. ACM, Winter 1998.

[Ioannidis00] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. ACM Conference on Computer and Communications Security (Athens, Greece, 1–4 November 2000), pages 190–199, 2000.

[ISI81] Information Sciences Institute, USC. RFC 791 - DARPA Internet program protocol specification, September 1981.

[Jayanti98] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. Journal of the ACM, 45(3):451–500. ACM Press, May 1998.

[Jung01] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS performance and the effectiveness of caching. ACM SIGCOMM Workshop on Internet Measurement (San Francisco, CA, 01–02 November 2001), pages 153–167. ACM Press, 2001.

[Katcher97] Jeffrey Katcher. PostMark: a new file system benchmark. TR3022. Network Appliance, October 1997.

[Kephart91] Jeffrey O Kephart and Steve R White. Directed-graph epidemiological models of computer viruses. In Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy, pages 343–359, May 1991.

[Kephart93] J. O. Kephart and S. R. White. Measuring and modeling computer virus prevalence. In Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy, pages 2–15, May 1993.

[Kim94] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. Conference on Computer and Communications Security (Fairfax, VA, 2–4 November 1994), pages 18–29, 1994.

[Klosterman00] Andrew J. Klosterman and Gregory R. Ganger. Secure Continuous Biometric-Enhanced Authentication. CMU-CS-00- 134. Technical Report, Carnegie Mellon Univeristy School of Computer Science, May 2000.

[Ko97] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: a specification based approach. IEEE Symposium on Security and Privacy (Oakland, CA, 04–07 May 1997), pages 175–187. IEEE, 1997.

[Kruse02] Warren G. Kruse II and Jay G. Heiser. Computer forensics: incident response essentials. Addison-Wesley, 2002.

[Kubiatowicz00] J. Kubiatowicz et al. OceanStore: an architecture for globalscale persistent storage. Architectural Support for Programming Languages and Operating Systems, 2000.

[Kumar95] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. USENIX Annual Technical Conference, pages 95–106. USENIX Association, 1995.

[Kumar98] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. IEEE Transactions on Knowledge and Data Engineering, 10(1), February 1998.

[Lamport82] L. Lamport, R. E. Shostak and M. C. Pease. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3):382–401, July 1982.

[Lee96] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems. Published as SIGPLAN Notices, 31(9):84–92, 1996.

[Lemos02] R. Lemos. Putting fun back into hacking. ZDNet News, 5 August 2002. http://zdnet.com.com/2100-1105-948404.html.

[Lie00] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. Architectural Support for Programming Languages and Operating Systems (Cambridge, MA, November 2000), pages 169–177. ACM, 2000.

[Liu00] Peng Liu, Sushil Jajodia, and Catherine D. McCollum. Intrusion confinement by isolation in information systems. IFIP Working Conference on Database Security (Seattle, WA, 25–28 July 1999), pages 3–18, 2000.

[Lumb00] Christopher Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. Symposium on Operating Systems Design and Implementation (San Deigo, CA, 23-25 October 2000). ACM, October 2000.

[Lunt88] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. IEEE Symposium on Security and Privacy, pages 59–66. IEEE, 1988.

[MacDonald98] Josh MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. European Conference on Object-Oriented Programming (Brussels, Belgium, July, 20-21). Published as Proceedings of ECOOP, pages 33-45. Springer-Verlag, 1998.

[MacDonald00] Josh MacDonald. File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[Malan00] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and application protocol scrubbing. IEEE INFOCOM (Tel Aviv, Israel, 26–30 March 2000), pages 1381–1390. IEEE, 2000.

[Malkhi97] D. Malkhi and M. Reiter. Byzantine quorum systems. ACM Symposium on Theory of Computing, pages 569–578. ACM, 1997.

[Malkhi00] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large distributed systems. IEEE Transactions on Knowledge and Data Engineering, 12(2):187–202. IEEE, April 2000.

[Malkhi01] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the Fleet system. In Proceedings of the 2nd DARPA Information Survivability Conference and Exposition, Vol. II, pages 126– 136, June 2001.

[Martin97] David M. Martin Jr, Sivaramakrishnan Rajagopalan, and Aviel D. Rubin. Blocking java applets at the firewall. Symposium on Network and Distributed Systems Security (San Diego, CA, 10–11 February 1997), pages 16–26, 1997.

[Martin02] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. International Symposium on Distributed Computing, 2002.

[Matthews97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. ACM Symposium on Operating System Principles (Saint-Malo, France, 5-8 October 1997). Published as Operating Systems Review, 31(5):238-252. ACM, 1997.

[McCanne93] Steven McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. Winter USENIX Technical Conference (San Diego, CA, 25–29 January 1993), pages 259–269, January 1993.

[McCoy90] K. McCoy. VMS file system internals. Digital Press, 1990.

[McKendrick26] A. G. McKendrick. Applications of mathematics to medical problems. In Proceedings of Edin. Math. Society, volume 14, pages 98–130, 1926.

[McKusick84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. ACM Transactions on Computer Systems, 2(3):181–197, August 1984.

[Medina01] A. Medina, A. Lakhina, I. Matta, and J. Byers. Brite: Universal topology generation from a user's perspective. Technical Report BUCS-TR2001-003, Boston University, 2001. World Wide Web, http://www.cs.bu.edu/brite/publications/.

[Miller96] Mark Miller and Joe Morris. Centralized administration of distributed firewalls. Systems Administration Conference (Chicago, IL, 29 September – 4 October 1996), pages 19–23. USENIX, 1996.

[Mirkovic02] Jelena Mirkovic, Peter Reiher, and Greg Prier. Attacking DDoS at the source. ICNP (Paris, France, 12–15 November 2002), pages 312–321. IEEE, 2002.

[Mockapetris88] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. ACM SIGCOMM Conference (Stanford, CA, April 1988). Published as ACMSIGCOMM Computer Communication Review, 18(4):123–133. ACM Press, 1988.

[Mogul87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: an efficient mechanism for user-level network code. ACM Symposium on Operating System Principles (Austin, TX, 9–11 November 1987). Published as Operating Systems Review, 21(5):39–51, 1987.

[Moore01] D. Moore. The Spread of the Code-Red Worm (CRv2), 2001. http://www.caida.org/analysis/security/code-red/coderedv2 analysis.xml.

[Moore03] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In Proceedings of IEEE INFOCOM 2003, San Francisco, CA, April 2003.

[Morris02] R. Morris. Storage: from atoms to people. Keynote address at Conference on File and Storage Technologies, January 2002.

[Muthitacharoen01] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. ACM Symposium on Operating System Principles. Published as Operating System Review, 35(5):174–187. ACM, 2001.

[Nessett98] Dan Nessett and Polar Humenn. The multilayer firewall. Symposium on Network and Distributed Systems Security (San Diego, CA, 11–13 March 1998), 1998.

[NetworkAssoc00] Network Associates and 2000-05. Vbs/loveletter@mm. World Wide Web, http://vil.nai.com/vil/content/v 98617.htm, 2000.

[NetworkAssoc01] Network Associates and 2001-07. W32/sircam@mm. World Wide Web, http://vil.nai.com/vil/content/v 99141.htm, 2001.

[NetworkAssoc03] Network Associates and 2003-08. W32/sobig.f@mm. World Wide Web, http://vil.nai.com/vil/content/v 100561.htm, 2003.

[NetworkAssoc04] Network Associates and 2004-01. W32/mydoom@mm. World Wide Web, http://vil.nai.com/vil/content/v 100983.htm, 2004.

[Newsome05] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatic Signature Generation for Polymorphic Worms. IEEE Symposium on Security and Privacy (Oakland, CA, 08–11 May 2005), pages 226–241. IEEE Computer Society, 2005.

[NFR02] NFR Security. http://www.nfr.net/, August 2002.

[Noble94] B. D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. Technical Report CMU–CS– 94–120. Carnegie Mellon University, February 1994.

[Northcutt01] Stephen Northcutt, Mark Cooper, Matt Fearnow, and Karen Frederick. Intrusion Signatures and Analysis. New Riders, 2001.

[One96] Aleph One. Smashing the Stack for Fun and Profit. Phrack 49, 7(49), November 8, 1996.

[Paxson98] V. Paxson. Bro: a system for detecting network intruders in real-time. USENIX Security Symposium, pages 31–51. USENIX Association, 1998.

[Pedersen91] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. Advances in Cryptology - CRYPTO, pages 129–140. Springer- Verlag, 1991.

[Pennington03] 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. USENIX Security Symposium (Washington, DC, 06–08 August 2003), 2003.

[Phillips06] John Phillips. Antivirus scanning best practices guide. Technical report 3107. Network Appliance Inc. http://www.netapp.com/library/tr/3107.pdf

[Porras97] Phillip A. Porras and Peter G. Neumann. EMERALD: event monitoring enabling responses to anomalous live disturbances. National Information Systems Security Conference, pages 353–365, 1997.

[Postel80] J. Postel. Transmission Control Protocol, RFC–761. USC Information Sciences Institute, January 1980.

[Postel81] Jonathan B. Postel. Simple mail transfer protocol, RFC–788. Network Working Group, November 1981.

[PSS03] Packet Storm Security. Packet Storm, 26 January 2003. http://www.packetstormsecurity.org/.

[Ptacek98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: eluding network intrusion detection. Technical report. Secure Networks Inc., January 1998.

[Purczynski02] Wojciech Purczynski. GNU fileutils – recursive directory removal race condition. BugTraq mailing list, 11 March 2002.

[Qie02] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: using an annotation toolkit to build dos-resistant software. Symposium on Operating Systems Design and Implementation (Boston, MA, 09–11 December 2002), pages 45–60. USENIX Association, 2002.

[Quinlan02] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. Conference on File and Storage Technologies, pages 89–101. USENIX Association, 2002.

[Rabin89] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. Journal of the ACM, 36(2):335–348. ACM, April 1989.

[RedHat99] Red Hat Linux 6.1, 4 March 1999. ftp://ftp.redhat.com/pub/redhat/linux/6.1/.

[Reiter95] M. K. Reiter. The Rampart toolkit for building high-integrity services. Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938), pages 99–110, 1995.

[Rose88] M. Rose. Post office protocol – version 3, RFC–1081. Network Working Group, November 1988.

[Rosenblum91] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. ACM Symposium on Operating System Principles. Published as Operating Systems Review, 25(5):1–15, 1991.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems, 10(1):26-52, February 1992.

[Rowstron01] Antony Rowstron and Peter Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. IFIP/ACM International Conference on Distributed Systems Platforms (Heidelberg, Germany, 12–16 November 2001), pages 329–350, 2001.

[Ruemmler03a] C. Ruemmler and J. Wilkes. UNIX disk access patterns. Winter USENIX Technical Conference, pages 405–420, 1993.

[Ruemmler03b] C. Ruemmler and J. Wilkes. A trace-driven analysis of disk working set sizes. HPL–OSR–93–23. Hewlett- Packard Company, April 1993.

[Samar95] V. Samar and R. J. Schemers III. Unified login with pluggable authentication modules (PAM). Open Software Foundation RFC 86.0. Open Software Foundation, October 1995.

[SANS00] SANS Institute. IP Fragmentation and Fragrouter, December 10, 2000. http://rr.sans.org/encryption/IP frag.php.

[Santry99] D. S. Santry, M. J. Feeley, N. C. Hutchinson, R. W. Carton, J. Ofir, and A. C. Veitch. Deciding when to forget in the Elephant file system. ACM Symposium on Operating System Principles. Published as Operating Systems Review, 33(5):110–123. ACM, 1999.

[Savage99] Stefan Savage, Neal Cardwell, DavidWetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. ACM Computer Communications Review, 29(5):71–78. ACM, October 1999.

[Scambray01] J. Scambray, S. McClure, and G. Kurtz. Hacking exposed: network security secrets & solutions. Osborne/McGraw-Hill, 2001.

[Schecter04] S. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. RAID 2004.

[Schneider90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys 22(4):299– 319, December 1990.

[Schneier98] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. USENIX Security Symposium (San Antonio, TX, 26–29 January 1998), pages 53–62. USENIX Association, 1998.

[Schneier99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. ACM Transactions on Information and System Security, 2(2):159–176. ACM, May 1999.

[Seltzer95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. Annual USENIX Technical Conference (New Orleans), pages 249-264. Usenix Association, 16-20 January 1995.

[Seltzer00] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. USENIX Annual Technical Conference (San Diego, CA), 18-23 June 2000.

[Sivathanu03] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. Conference on File and Storage Technologies, pages 73–89. USENIX Association, 2003.

[Sophos02a] Sophos. MailMonitor, 2002. http://www.sophos.com/products/software/mailmonitor/.

[Sophos02b] Sophos. Sophos Anti-Virus, 2002. http://www.sophos.com/products/software/antivirus/.

[Soules02] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in a comprehensive versioning file system. Technical report CMU–CS–02–145. Carnegie Mellon University, 2002.

[Soules03] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, Gregory R. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. 2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, Mar 31 - Apr 2, 2003.

[Spafford89] Eugene H. Spafford. The Internet worm: crisis and aftermath. Communications of the ACM., 32(6):678–687.

[Spasojevic96] M. Spasojevic and M. Satyanarayanan. An empirical-study of a wide-area distributed file system. ACM Transactions on Computer Systems, 14(2):200–222. ACM Press, May 1996.

[Spitzner03] Lance Spitzner. Honeytokens: The Other Honeypot. Security Focus, 21 July, 2003. http://www.securityfocus.com/infocus/1713.

[Staniford02] 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.

[Staniford04] S. Staniford. Containment of Scanning Worms in Enterprise Networks. Journal of Computer Security 2004.

[Steiner88] J. G. Steiner, J. I. Schiller, and C. Neuman. Kerberos: an authentication service for open network systems. Winter USENIX Technical Conference, pages 191–202, 9–12 February 1988.

[Stoica01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Conference (San Diego, CA, 27–31 August 2001). Published as Computer Communication Review, 31(4):149–160, 2001.

[Strunk00] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation (San Diego, CA, 23–25 October 2000), pages 165–180. USENIX Association, 2000.

[Strunk02] J. D. Strunk, G. R. Goodson, A. G. Pennington, C. A. N. Soules, and G. R. Ganger. Intrusion detection, diagnosis, and recovery with self-securing storage. Technical report CMU–CS–02–140. Carnegie Mellon University, 2002.

[Sugerman01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMwareWorkstation's Hosted Virtual Machine Monitor. USENIX Annual Technical Conference (Boston, MA, 25–30 June 2001), pages 1–14. USENIX Association, 2001.

[Sun89] Sun Microsystems. NFS: network file system protocol specification, RFC–1094, March 1989.

[Symantec03a] Symantec. W32.Blaster.Worm, August 11, 2003.

[Symantec03b] Symantec. W32.Welchia.Worm, August 18, 2003.

[T10.org99] Object based storage devices: a command set proposal. Technical report. October 1999. http://www.T10.org/.

[Tennenhouse97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. IEEE Communications, 35(1):80–86, January 1997.

[Terry95] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. ACM Symposium on Operating System Principles (Copper Mountain Resort, CO). Published as Operating Systems Review, 29(5), 3–6 December 1995.

[Tichy80] W. F. Tichy. Software development control based on system structure description. PhD thesis. Carnegie Mellon University, Pittsburgh, PA, January 1980.

[Tripwire02] Tripwire Open Souce 2.3.1, August 2002. http://ftp4.sf.net/sourceforge/tripwire/tripwire-2.3.1-2.tar.gz.

[Vaidyanathan02] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. Published as Performance Evaluation Review, 29(1):62–71. ACM Press, 2002.

[Venema00] Wietse Venema. Strangers in the night. Dr. Dobbs Journal, 25(11):82–88, November 2000.

[Venkataraman05] Shobha Venkataraman, Dawn Song, Phil Gibbons, and Avrim Blum. New Streaming Algorithms for Superspreader Detection. Network and Distributed System Security Symposium (San Diego, CA, 03–04 February 2005), 2005.

[Vogels99] Werner Vogels. File system usage in Windows NT 4.0. ACM Symposium on Operating System Principles (Kiawah Island Resort, Charleston, South Carolina, 12-15 December 1999). Published as Operating System Review, 33(5):93-109. ACM, December 1999.

[Wang03a] Yang Wang, Deepayan Chakrabarti, Chenxi Wang, and Christos Faloutsos. Epidemic spreading in real networks: An eigenvalue viewpoint. In Proceedings of the 22nd International Symposium on Reliable Distributed Systems, 2003.

[Wang03b] Yang Wang and Chenxi Wang. Modeling the effects of timing parameters on virus propagation. In Proceedings of the 2003 ACM workshop on Rapid Malcode, pages 61–66. ACM Press, 2003.

[Weaver04] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. USENIX Security Symposium 2004.

[Weaver01] N. Weaver. Warhol Worms: The Potential for Very Fast Internet Plagues, posted August 15, 2001. http://www.cs.berkeley.edu/_nweaver/warhol.html.

[Wetherall99] David Wetherall. Active network vision and reality: lessons from a capsule-based system. Symposium on Operating Systems Principles (Kiawah Island Resort, SC., 12–15 December 1999). Published as Oper. Syst. Rev., 33(5):64–79. ACM, 1999.

[Whyte05] D. Whyte, E. Kranakis, and P. van Oorschot. DNS-based Detection of Scanning Worms in an Enterprise Network. NDSS 2005.

[Williamson02] Matthew M Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In Proceedings of the 18th Annual Computer Security Applications Conference, Las Vegas, Nevada, December 2002.

[Williamson03] Matthew M Williamson. Design, implementation and test of an email virus throttle. In Proceedings of the 19th Annual Computer Security Applications Conference, Las Vegas, Nevada, December 2003.

[Wong04a] Cynthia Wong, Chenxi Wang, Dawn Song, Stan Bielski, and Gregory R. Ganger. Dynamic Quarantine of InternetWorms. International Conference on Dependable Systems and Networks (Florence, Italy, 28 June 2004). IEEE Computer Society, 2004.

[Wong04b] Cynthia Wong, Stan Bielski, Jonathan M. McCune, Chenxi Wang. A Study of Mass-mailing Worms. WORM'04, October 29, 2004, Washington, DC, USA.

[Wong05] Cynthia Wong, Stan Bielski, Ahren Studer, Chenxi Wang. Empirical Analysis of Rate Limiting Mechanisms. 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005), September 7-9, 2005, Seattle, Washington.

[Wylie00] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. IEEE Computer, 33(8):61–68. IEEE, August 2000.

[Wylie04] J. J. Wylie, G. R. Goodson, G. R. Ganger, and M. K. Reiter. A protocol family approach to survivable storage infrastructures. FuDiCo II: S.O.S. (Survivability: Obstacles and Solutions), 2nd Bertinoro Workshop on Future Directions in Distributed Computing, 2004.

[Yaar03] Abraham (Avi) Yaar, Adrian Perrig, and Dawn Song. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. IEEE Symposium on Security and Privacy (Oakland, CA, 11–14 May 2003). IEEE Computer Society, 2003.

[Yaar05] Abraham Yaar, Adrian Perrig, and Dawn Song. FIT: Fast Internet Traceback. IEEE INFOCOM (Miami, FL, 13–16 March 2005), 2005.

[Ylonen96] Tatu Ylonen. SSH | Secure login connections over the internet. USENIX Security Symposium (San Jose, CA). USENIX Association, 22-25 July 1996.

[Zhang00] Yin Zhang and Vern Paxson. Detecting stepping stones. USENIX Security Symposium (Denver, CO, 14–17 August 2000). USENIX Association, 2000.

[Zhang02] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. ACM SIGOPS European Workshop. ACM, 2002.

[Zhao01] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: an infrastructure for fault-tolerant wide-area location and routing. UCB Technical Report UCB/CSD–01–1141. Computer Science Division (EECS) University of California, Berkeley, April 2001.

[Zou02] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code red worm propagation modeling and analysis. In Proceedings of the 9th ACM Conference on Computer and Communication Security, November 2002.