Disk-Level Behavioral Malware Detection

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

Nathanael R. Paul

May 2008

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

Nathanael R. Paul

Approved:

David E. Evans (Advisor)

Jack W. Davidson (Chair)

Sudhanva Gurumurthi

Nina Mishra

Ronald D. Williams

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

May 2008

Abstract

We present a new malware detection method that takes advantage of the processing power now available on disk drives. Our method uses the disk processor to monitor disk requests and identifies malicious programs based on characteristic properties of the disk requests they make. Disk-level behavioral detection offers several advantages over traditional approaches since the disk processor can perform computation without burdening the host processor and can mediate disk accesses before they reach the physical medium. This dissertation describes and evaluates two instances of our approach: one uses a simple, general infection signature to reliably detect a class of file-infecting viruses; the other illustrates how our approach can be used with behavior-specific signatures to recognize known malware.

By identifying a large class of common disk-level virus behavior, we develop simple rules that can be enforced by the disk processor. These rules are able to detect unknown viruses by recognizing their characteristic file-infecting behavior. Two of the rules are able to detect all but two types of the file-infecting viruses in our test set. From our testing based on traces of disk activity collected from eight different users, we identify a small set of activities that generate false positives. We present mechanisms to mitigate or avoid these false positives.

Some malware performs other malicious actions besides the recognized file-infecting behavior. We develop a process for finding behavior-specific signatures to precisely identify disk-based malware using a candidate set of three viruses and one worm. We can detect a family of malware (i.e., its variants) using a single disk-level behavior-specific

signature.

We present the design of a disk-level malware detector that can enforce these rules and signatures. The disk infers high-level file system activity from disk-level events and is able to map disk blocks to files. Based on this design, we implemented a prototype disk-level detector that demonstrates the feasibility of disk-level virus detection.

Our detection is resilient to many traditional obfuscation techniques, because our approach is behavior-based. It is more difficult to change a program's behavior than it is to change a few bytes to generate a new variant. Further, by using the disk processor to recognize disk-based malware, a disk-level detector can use general behavioral rules to detect unknown malware and use behavior-specific signatures to precisely identify other more sophisticated malware. Since the detector is running below the host level, it cannot be circumvented even if the host is compromised.

Acknowledgments

Throughout graduate school, there have been many people that have helped me in ways both small and great. My advisor, Dave Evans, always encouraged me to pursue and study interesting topics. His mentorship was "handsoff" to let me learn from my mistakes, but he was also always there to steer me in the right direction.

Other professors have also had a significant impact. Although I came to know Sudhanva Gurumurthi later in graduate school, I learned much from our work together. I am grateful to Nina Mishra, Jack Davidson, and Ron Williams for serving on my thesis committee. From Clemson, I thank Harold Grossman for his friendship and guidance.

I have been blessed with many friends along the way, and I am grateful for these friendships. Many of my friends and family back at home have helped me, and I am grateful for their prayers. While our distance to each other may have grown, they were still much appreciated. To my UVa friends, I thank Mark, John, and Carrie for being there from the beginning; Jeremy, Chris, and Shahrukh for sharing in my passion of food and sports (and an extra thank you to Jeremy for helping me get my thesis turned in 4000 miles away); and Adrienne for her help in the thesis work. I enjoyed and learned a lot from all of the many times I shared with all of you.

I am indebted to the systems staff, especially Mark, for incredible systems support. The administrative staff was helpful in helping me get the necessary things done and on time.

I thank Brian and Jolynn for their encouragement and many fun meals, Jim and Cindy for letting their daughter go around the world with me, and my family for their support.

I am fortunate to have such loving, supporting, and encouraging parents. Even while young, I was always encouraged to ask questions and learn about interesting things. My dadś patience in teaching me, even in the face of strong jet blasts, helped me to have fun while learning.

And last, while my wife has shared in many of my experiences, she has always shown her sacrificial love and support. My thesis and work are far better because of her.

"*Be still, and know that I am God...*"

Psalm 46:10

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Current malware detection has limited success because of essential problems in its design. We discuss limitations with traditional anti-virus (AV) programs and then describe how disk-level behavioral detection can overcome these limitations.

1.1 Flaws in Current Malware Detection

Prevailing malware detection approaches suffer from three fundamental flaws that give malicious software creators a seemingly insurmountable advantage in the arms race between viruses and anti-virus software: host-level detectors can be circumvented, detectors can be easily evaded, and detectors cannot detect new threats because of reliance on blacklists.

Host-level Detection is Circumventable. One problem with most malware detection is that monitoring and response are performed at the level of the host operating system. If malware can compromise the host operating system (OS), or worse, below the host OS, it can circumvent detection while causing arbitrary damage to the host. Rootkits are examples of malware that circumvent OS-based protection often by changing the OS itself (e.g., creating a backdoor in the system for future unauthorized access) [King and Chen, 2005; Halderman and Felten, 2006]. Recent research has proposed rootkits that attack a layer below the OS [King and Chen, 2005]. This presents a new and challenging problem.

The OS must now detect if it is running within a malicious virtual environment. OS-based approaches offer no reliable solution to detect or recover from low-level rootkit infections, since the malware may be able to circumvent detection monitoring points by hiding its location below the detection software residing in the OS.

Static Code Detection is Easily Evaded. A second problem is that current approaches are predominantly static: they attempt to detect malware by recognizing its code, not by observing its behavior. This is problematic, since changing the code of a program without disrupting its behavior is easy. There are infinitely many different instruction sequences that can be inserted in a program without changing its observed behavior. To evade static detectors, virus authors developed metamorphic viruses that transform the virus code as it propagates in ways that are likely to circumvent static signatures. Static analysis techniques have been developed to detect these obfuscated viruses [Szor, 2005a; Christodorescu et al., 2005; Preda et al., 2007], but virus authors seem to have a permanent advantage here since producing new obfuscation techniques is easier than producing counter de-obfuscators [Song et al., 2007]. Given a program, there are infinitely many different programs that will produce the same behavior. It is much more difficult to determine (and undecidable in general) if two arbitrary programs behave the same way than it is to produce a new code variant with the same behavior. Most host-level malware detection is constrained to follow the static approach, however, since it is difficult to dynamically observe malware behavior with high precision without incurring unacceptable execution overhead (see Section 2.4).

Blacklists Fail to Detect New Threats. The third problem is that current virus detection approaches rely on blacklisting of static attributes of the malware (a *signature*). With traditional scanning, if a file matches a signature on the blacklist, it is declared to be a virus; otherwise, it is considered non-malicious. A program matches a signature when some static attributes of the program (e.g., file size, certain bytes in the header, other bytes of code) are the same as specified in a signature. While the signatures are often of bytes within the code, signatures may contain static attributes such as the file size (assuming

that the file size will not match some benign program's file size) to decrease the false positives. Other methods to avoid false positives include specifying sequences of bytes that it is hoped match all instances of the virus, but no non-malicious programs. This means that there is little chance of detecting new viruses, or even most simple variations of known viruses.

To illustrate how easily simple variations can be used to evade current detectors, we downloaded the source code to a known virus, W32.Cabanas, and we then compiled it. The Symantec anti-virus engine recognized the virus as Cabanas (we did not try other AV engines). We then inserted a `nop` into the decryption routine, and the AV engine failed to recognize it. These findings confirmed earlier results that found detectors had difficulties with variable renaming, garbage insertion (instruction sequences that are equivalent to `nops`), and run-time code obfuscation/deobfuscation [Christodorescu and Jha, 2004]. Others have found similar problems of detectors having trouble with obfuscated malware [Keizer, 2006; Moser et al., 2007b]. This is not a failure that is singular to Symantec but widespread among traditional AV engines, because they use blacklists based on static attributes of the malware.

1.2 Overcoming Flaws in Current Detector Designs

As an attempt to mitigate the problems with static detection and blacklists, AV engines have developed emulated environments. The detection engine models the registers, stack, and other important functionality and then monitors malware while "executing" in this environment. This prompted malware authors to employ anti-emulation techniques [Stepan, 2006] that either disrupt the emulator or recognize when the malware is executing inside an emulator and change its behavior to evade detection [Ciubotariu, 2006]. Early techniques used floating point instructions that early emulators did not emulate. The IDEA.6155 virus used the IDEA encryption cipher that caused some emulators to run for as long as a half hour [Hayes, 2002]. Current viruses often detect the emulator and

may change their behavior based on detecting the emulator [Ferrie, 2006b]. These evasion problems are not unique to host-level detectors (Chapter 6), but they impact the malware's ability to circumvent host-level detection techniques (Section 2.4.1).

This work seeks to overcome these flaws by using the disk drive processor to detect malware based on its characteristic disk-level behavior.

Thesis Statement: The disk drive processor can be used to improve malware detection.

Disk drive processors today have extra processing capabilities that allow us to put AV detection into the disk drive. By putting AV detection into the disk drive, we can overcome many of the flaws with current detectors.

The disk lies beneath the OS where the disk processor can monitor every disk request. Every OS request to or for data must be serviced by the disk processor (unless a read is for data in the host's cache that was serviced earlier by the disk). By operating independently of the host, the disk processor is isolated from attacks that circumvent host-level detection by modifying the host OS. The disk-level detector is not circumvented even when malware compromises the host OS.

Our approach is dynamic: instead of examining the program text of malware, it relies on observing the sequence of disk requests it makes. Creating malware that changes its disk usage without diminishing its effectiveness is much more difficult than changing its static code. Rather than relying on blacklisting of static code bytes of known malware, we demonstrate behavioral rules that capture the fundamental file-infecting behavior that is common to most file-infecting viruses. This means our detector can detect previously unknown viruses. In addition to these generic behavioral rules, we use behavior-specific signatures that are resistant to obfuscations while detecting malware more precisely.

Our work focuses on the Windows platform, since it is the predominant platform for virus creators. The methods we present could be applied to other platforms. Our goal

is not to supplant host AV detection, but to instead provide improved detection of disk-based malware (malware that uses the disk). In particular, we focus on an important class of malware: parasitic file-infecting viruses (Section 4.1). This malware class is large and important, but it does not cover all malware.

Our behavioral rules can recognize unknown viruses through their behavior, but some malware deviates from the observed general behavior and require other methods for detection. To detect this deviant malware (e.g., droppers, worms), we can create disk-level behavior-specific signatures. We describe behavior-specific signatures and demonstrate their use in Chapter 5. Our approach will not work for malware that propagates without using the disk, such as worms that use the network as their primary vector and memory-resident viruses that infect running processes in memory.

We also assume a strict definition of "executable" to include only files that adhere to the Microsoft PE/COFF file format [Microsoft, 2006]. Many other types of files can inadvertently host viruses since the boundary between executable and non-executable content is fuzzy. For example, application-level viruses like Microsoft Word Macro viruses are not in our target class. Although these viruses may generate recognizable disk-level activity as they propagate, they are much better dealt with at the level of the hosting application.

1.3 Contributions

Our key contributions are:

(1) identifying general file-infecting virus behavior that is exhibited by a large class of viruses (Chapter 4). We examined disk-level activity from several viruses and developed rules that are able to detect a large class of viruses based on their file-infecting behavior. We tested candidate rules on a random sample of viruses (300+) and demonstrated that we could identify all the file-infecting viruses. For those viruses that compromised executables through other malicious behaviors, we detail interesting disk-level behavior that may be detected with other behavioral rules.

(2) diagnosing causes of false positives and designing methods to deal with false positives (Chapter 4). To test the rules' false positive rates, we gathered data both from eight different users and also other behaviors that would likely raise false positives. We then tested four different behavioral rules with the collected data. The general behavioral rules triggered many false positives, but we were able to identify classes of behaviors that caused the false positives. Knowing the causes of the false positives, we developed ways to deal with them.

(3) developing a process for capturing specific disk-level behavior of particular malware (Chapter 5). Our disk-based detection is applicable to any different type of malware that uses the disk. We demonstrated the effectiveness of behavior-specific signatures to detect both worms and viruses.

(4) designing a disk-level malware detector and implementing a prototype (Chapter 3). Using higher-level semantic disk request information, our prototype malware detector demonstrated detection of viruses at the disk-level is possible.

1.4 Roadmap

In the next chapter, we present the background and related work. Chapter 3 describes the design of a disk-level detector and then goes on to discuss the disk-level detector prototype's implementation. In Chapter 4, we introduce our general behavioral rules that are able to identify unknown viruses by recognizing their file-infecting behavior. We study some example malware that has interesting disk behavior and present disk-level behavior-specific signatures for them in Chapter 5. We analyze the security of our approach in Chapter 6. Chapter 7 ends the thesis by drawing some conclusions about our contributions and speculating on future work.

Chapter 2

Background

Malware is software that does something that matches the intent of the creator but which the owner of the host running the program considers bad. We primarily focus on viruses in this thesis, but our general approach is applicable to other forms of malware, including worms, spyware, and rootkits — each of which uses the disk in its own malicious way. All these types of software have a similar goal: to infiltrate or damage a computer system against the owner's will.

This thesis focuses on the classic class of malware known as parasitic or file-infecting viruses. Despite the widespread deployment of anti-virus software, file-infecting viruses remain a serious threat. In the first half of 2006, the polymorphic viruses W32.Polip and W32.Detnat ranked first and fourth on Symantec's top new malicious code threat list, which ranks the frequency of previously unseen malware captured by Symantec honeypots [Turner, 2006a]. In the second half of 2006, the polymorphic virus W32.Bacalid ranked fourth [Turner, 2007]. In November 2007, Virus Bulletin released statistics showing 25% (eight of 32) of the most prevalent malware were viruses [Virus Bulletin, 2007]. In addition to the virus threat, our general approach is applicable to any type of malware that uses the disk.

The term virus alludes to the ability of these programs to infect other, normally non-malicious host programs, by attaching themselves to host programs and then propagating

to other programs [Cohen, 1987]. Viruses can spread through a variety of methods and can attack executables as well as different types of host files that support executable content, such as Microsoft Word documents [Symantec, 1997b] and screensaver files [Symantec, 2003]. Simpler viruses propagate by copying themselves into the target host, so that once the virus is known, it can be detected using a signature that matches its constant virus body. Because viruses both continually cause problems for AV engines [Braverman et al., 2006] and have intrinsically interesting disk activity because of their interaction with other host programs, we focus on examining the disk-level behavior of viruses. We consider more sophisticated viruses that can thwart these naive AV scanners (e.g., using polymorphism and metamorphism) in Section 2.1. We explore the disk requests produced by file-infecting viruses and block their propagation by recognizing the general characteristic traits of their I/O activity using general behavioral rules (see Section 4).

2.1 Current Approaches in Malware Detection

This section highlights current techniques for malware detection and discusses their shortcomings. Detection and removal of computer viruses and worms involves using a host of techniques. Most anti-virus software does scanning using what is known as on-access semantics, where they check a file for a virus when it is created, opened, or closed.

The most popular technique is signature matching, also known as *string scanning*. Early malware detectors began comparing strings in files to a database of signatures (i.e., sequences of bytes) of known malware, and this method is still the predominant detection method today. String scanning suffers from both potentially high *false positives*, identifying a harmless program as malware,[1] and high *false negatives*, failing to detect malware. Longer, more precise signatures lower the false positive rate, but also increase the database size, scanning time, and false negative rate (variants that may not have the same exact signature may escape detection).

[1] In May, 2007, Symantec updated their malware signatures and crippled thousands of Chinese PCs by mistakenly identifying two core Windows .dll files as a Trojan horse [Keizer, 2007].

Simple string scanning techniques are defeated by malware that better hide their location and alter their code as they propagate using *polymorphism*. A polymorphic virus is a virus that uses encryption, and it changes decryption routines in each infection to resist static string matching on the decryption routines. Encrypting strings defeats string scanning as long as the encryption keys change as the virus propagates. The increased obfuscation in the malware forced malware detection engines to use a variety of scanning techniques from simple string scanning to regular expression matching, to more complex techniques like X-RAY scanning [Perriot and Ferrie, 2004], which scans multiple parts of a file and performs differencing analyses to detect polymorphic malware. X-RAY scanning works because the malware authors use simple home-grown methods of encryption and decryption that fail to hide relationships with the text.

Although encryption can be used to change the body of the virus, the decryption routine must be capable of running directly. Earlier viruses used static decryption routines that did not change across infections, and string scanning continued to detect these viruses using byte sequences in the decryption routines. To avoid static decryption routines, malware authors developed techniques that change the code itself.

Viruses that achieve a non-constant or dynamic virus body without necessarily using encryption are known as *metamorphic*. They can accomplish this metamorphism by transformations to the code such as equivalent instruction substitution (e.g., `sub eax, 4` instead of `add eax, -4`). The distinguishing factor between polymorphic and metamorphic viruses is the virus body. In polymorphism, the virus body is encrypted with a different encryption key each infection, but the virus' body will decrypt to the same code across multiple infections. By using different code transformations, metamorphic virus bodies are different across infections.

To detect complex polymorphic and metamorphic viruses, anti-virus software must resort to high-overhead detection methods like partial emulation. Emulation has a high execution overhead. For every program scanned, it can take over a million instructions to emulate it [Szor, 2005b; Nachenburg, 1997]. With emulation, the anti-virus engine emu-

lates the program until a certain point of execution (i.e., a breakpoint) and then compares the state (e.g., register and stack contents) to a known malware state [Szor, 2005a]. This is useful for detecting polymorphic malware that encrypts itself when propagating, but must first execute a decryption routine before the malware can execute. Because emulation has high computational costs, however, the anti-virus engine must stop emulating eventually and allow the program to run normally. If malware can execute enough garbage instructions without being matched, it can avoid detection.

Moreover, unless the emulator models the architecture of the host system accurately and completely emulates all executions, polymorphic and metamorphic malware can still evade detection by recognizing when they are running in an emulator and changing their behavior. Malicious programs have used specific features that are not implemented in the emulated environment to break emulation. Examples of this include using floating point arithmetic [Black Jack, 2001] or multiple threads (e.g., W32.Chiton [Ferrie, 2002b]). Simpler attacks (e.g., floating point arithmetic) can be fixed by implementing the missing feature.

However, when the emulated environment is as complex as efficiently emulating Windows on top of an Intel or AMD chip, hiding all side effects of emulation, even with hardware supported virtualization, is extremely difficult. This has led to an arms race between the malicious code that detects side effects of emulation and the emulators that attempt to hide the side effects [Ferrie, 2006a]. Ferrie argues that, in theory, we should be able to achieve a *virtual reality* where emulation-based detection is unreliable, because of indistinguishable behavior transparently virtualized in software and realized, for example, by anomalous behavior of bugs in hardware components [Ferrie, 2006a].

In practice, precise emulation-based detection and software virtualization is difficult. Multiple issues exist that could be exploited to detect an emulator including the sharing of hardware resources, multiple timing differences, and performance trade-offs [Garfinkel et al., 2007]. In using the same hardware, the emulated program can run code to detect shared resources. Timing differences exist in multiple places including privileged instruc-

tions or remote server time sources. For performance differences, better hardware can fix some problems (e.g., hardware support for shadow page tables), but any single instruction that is not emulated precisely affords an opportunity for emulator detection.

Another detection technique is behavior blocking [Szor, 2005a]. This technique is used to detect hitherto unknown malware by looking for certain types of suspicious behavior. A distinguishing factor of a behavior blocker compared to an emulator is that the behavior blocker continuously monitors real execution. For example, one common trait of computer viruses is to append themselves to a Windows Portable Executable file. A major problem with behavior blocking is that monitoring behavior involves run-time overhead; if it is all done by the host processor, there are practical limits on the complexity and precision of the analysis. By using the disk processor, we can implement a form of behavioral detection with low overhead (see Section 2.4.2).

Another problem with behavior blocking is that it is difficult to develop the behaviors that correctly identify viruses without also generating false positives (that is, misidentifying non-malicious program behaviors as malicious). Because our techniques are a form of behavior blocking, they are also susceptible to false positives when benign programs exhibit similar behaviors to the targeted malicious behavior. Section 4.5 discusses the identified sources of the false positives, and how we can address them.

A key problem associated with traditional AV techniques is they are host-based. Many AV modules and firewalls monitor the machine by filtering the data in the kernel. On-access detection algorithms are typically implemented as kernel modules. For example, Windows has an I/O stack where a kernel module can insert itself to monitor all file system operations [Nagar, 2003], and a firewall or AV module can use this functionality to trigger a scan of a file. Malware can be designed to attack a machine in ways that circumvent or disable host-level monitors.

One motivating example of malware that attacks host-based detection is the rootkit Mailbot [Kasslin, 2006]. Mailbot can circumvent the host-based monitoring points rendering kernel-level detection modules useless. It uses thread-level hooking of interrupt

handlers and circumvents the detector's monitoring points inside the OS. One feature of Mailbot that is particularly difficult to detect is its use of thread-level hooks of the handler functions used to transfer control to kernel mode (`sysenter` and `INT 0x2E`). The malware wants control to go to its malicious code but directly inserting a jump to code outside of the kernel module might be recognized as malicious. Instead, the malware searches the kernel module for unused memory and places its hook inside the module to transfer control to its own code (some tools check the addresses of the `sysinter` and `INT 0x2E` handler functions for jumps outside of the kernel module but do not find a jump to an address within the kernel module questionable). When a thread tries calling the hooked functions via a `sysenter` or `INT 0x2E` handler, it will be redirected to an address within the kernel module that will jump to the malicious code [Kasslin, 2006]. Because of evasive malware like Mailbot and virtual machine based rootkits, we need detection outside of the host.

2.2 Malware Authors' Motivation

In the 1990s, most malware authors were not financially motivated. Many malware authors made viruses from a combination of boredom, of fame, or of curiosity [Gordon, 1995]. These authors were typically young teenagers who eventually outgrew their virus writing behavior. One study [Gordon, 1994] found that many of these earlier writers never had specific targets to compromise and did not condone the viruses being released into the *wild* (spreading via normal operations).

Times have changed. Malicious adversaries now have many reasons for compromising a machine including spamming and phishing attacks. To coordinate large-scale attacks, attackers will gain remote control of machines to form a network of machines known as a *botnet*. This botnet is used for financial gain to carry out spamming and phishing attacks. In 2004, at least 100,000 machines were added to a botnet each week [Ilet, 2004]. An army of 10,000 machines can currently be purchased for $1,000 [Gutmann, 2007]. Others have

advertised prices as low as $2,000 to $3,000 for 20,000 machines [Acohido and Swartz, 2004]. These figures illustrate the financial motivation malware authors have to acquire and maintain control of a large number of machines.

To build and maintain control of a botnet, an attacker needs to compromise a machine in a persistent way. Although network-based malware such as fast-spreading worms has captured much recent attention due to their replication speed and the prevalence of unpatched machines with known vulnerabilities, disk-based malware endures as a prevalent way to deploy persistent malware [Virus Bulletin, 2007]. In particular, file-infecting viruses remain attractive to attackers due to the widespread popularity of peer-to-peer file-sharing networks, the ability to compromise a machine without needing a vulnerability, and the opportunity they provide attackers to persist stealthily on a compromised machine [Rutkowska, 2006; Shin et al., 2006]. To keep a machine as a member of a botnet, an attacker can use a virus to infect an executable to re-establish control after a reboot. Rather than later attempting reinfection, there are two main reasons for an attacker to use a persistent infection to maintain control of a compromised machine. If the machine needs to be reinfected after a reboot, it could be patched in the interim rendering reinfection impossible using the same techniques. Another reason for persistent malware is that other attackers may infect the machine, and this has caused some malware to patch the infected machine to guard against other possible compromises.

2.3 Malware Signature Database Growth

Because much known malware can be recognized by AV engines after some period of time, attackers must continually develop new malware that will not be recognized by AV signatures to compromise more machines. Once new malware is released, the AV engines will eventually update their signatures to combat the new malware. The growing size of the signature databases illustrates the mounting threat of malware.

The increasing size of malware signature databases forces AV engines to consume more

resources to check the signatures. Because many of these host-based detectors use static signatures, they often need a different signature for each variant of malware released. Many companies see hundreds of malware samples a day, each of which requires developing a new signature. The computational overhead of checking these signatures is significant, and some have suggested using separate microprocessors for malware detection [Silberstein, 2004; Tarari, 2006].

Figure 2.1 shows the size of two commercial signature databases from 2001 to the present using data from the RAV [Reliable Antivirus, 2006] (RAV's database does not grow as quickly after mid-2003 when Mirosoft began its acquisition of them in June of that year [Microsoft, 2003a]) and Norton [Symantec, 2006] (acquired by Microsoft in 2003 [Microsoft, 2003a]) and Norton [Symantec, 2006] anti-virus products. TrendMicro's malware database follows a similar trend: it is now as large as 30.1 MB [TrendMicro, 2007], about 80% larger than it was (16.8 MB) approximately one year ago.

One reason for the rapid growth of signature databases is the use of new malware distribution methods. Attackers have taken advantage of the static nature of signatures by timing the release of new variants according to released signatures. Once a new signature is released to match that malware, they release a malware variant that does not match the signature [Drori et al., 2005]. The key is waiting until the signature has been released to match the previous malware instance before distributing the new variant. This delay maximizes the effectiveness of each released variant. Thus, adding a signature to the signature database for new malware begins a cycle of releasing slightly modified signatures for slightly modified malware variants. This cycle of releasing signatures for variants can only be broken if a method is used that is resilient to easily-made changes to a virus implementation. These distribution methods contribute to the exponential increase in malware. The number of malware families grows at a linear pace, but the number of virus variants grows exponentially [Securitas Technologies, 2006].

In a study to find out how quickly 24 AV companies respond to a malware outbreak, Marx found that AV companies took an average of 10 hours to respond with a publicly

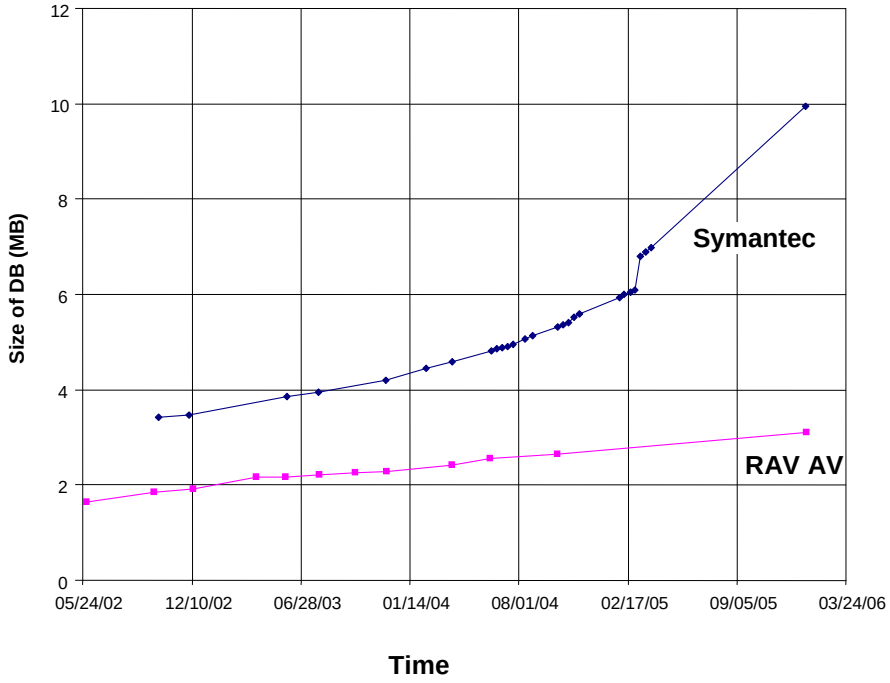


Figure 2.1: Norton and RAV signature database size.

available update [Marx, 2004]. In addition to response times, the frequency of updates is key. On one day there were as many as 89 update releases (an average of almost four updates per AV engine) with each update containing one or more signatures. Over the 8 months covered by this study, signature database sizes grew about 25%. Marx goes on to test the heuristic detection of these malware scanners, and finds the best malware heuristics catch 39% of a test set of malware. Not only does this show that static signatures do not work, but that work still remains to be done on more general types of detection [Marx, 2004]. Although not many public details of the specifics of the study was released, more recent results by Consumer Reports found that current detection software still has problems detecting malware variants [Matasano Security, 2006]. After creating modified versions of known samples of malware, many AV engines had problems with these variants. These

engines were poor at detecting unknown malware that did not already have a signature in a signature database.

Sometimes malware authors release variants of older malware instead of creating something completely new. A signature may still need to be created to recognize this variant. A report by Microsoft in 2006 [Braverman et al., 2006] shows virus authors are recycling older viruses such as W32.Parite. In fact, the report goes on to show that viruses were the fourth most common malware family removed by Windows Malicious Software Removal tool. Much of the reasons for virus popularity can be attributed to the W32.Parite virus (see Section 5.2.4), originating from 2001. Complex viruses such as W32.Parite can be used to deliver a rootkit or backdoor [Turner, 2006b] that allows the attacker to have remote control of the machine.

Malware production has become financially lucrative [Goudey, 2004], and viruses have become increasingly sophisticated, stealthy, and targeted. The historical example of the W32.ZMist illustrates the danger. A year after it was released (in 2001), companies were still tweaking signatures to detect it more precisely [wizard, 2002]. Today, it is believed that two percent of infections using the virus' infection engine (mistfall) are still not reliably detected [Rutkowska, 2006]. As the W32.Parite and W32.ZMist examples show, older viruses still give AV engines trouble, and attackers are taking advantage.

2.4 Related Work

This section presents prior work on virus detection, including recent work on improving traditional detection techniques, other work focusing on dynamic behavior, and other approaches to moving malware detection off the host processor. We highlight the most relevant work here. Singh and Lakhotia's annotated bibliography provides a broader overview of work on research detecting malware [Singh and Lakhotia, 2002].

2.4.1 Static Signature Detection

Several recent projects aim to improve traditional approaches to virus detection. These works use static code analysis to detect viruses, but employ semantic analysis techniques to infer the underlying behavior of obfuscated viruses. Christodorescu et al. investigated the shortcomings of many current virus scanning techniques [Christodorescu and Jha, 2003; Christodorescu and Jha, 2004]. They demonstrated how simple obfuscations including code transposition, register reassignment, dead-code insertion, and instruction substitution to known malware can fool malware detection engines [Christodorescu and Jha, 2003]. All of these techniques can easily defeat static signature-based malware scanners.

Code transposition simply reorders instructions that are independent. In register reassignment, different registers are used in a segment of code to achieve the same functionality. Dead-code insertion inserts instructions that do not affect the program's state in a meaningful way (i.e., a `nop`). Using equivalent instructions to perform the same action (e.g., `xor eax, eax` is the same as `mov eax, 0`) is known as instruction substitution.

One static analysis approach that can detect certain obfuscated viruses is a static analysis to derive behavioral malware signatures based on memory accesses [Christodorescu et al., 2005]. The detector works by creating a template of instructions with abstractions for memory locations (e.g., variables and registers) and constants made from a disassembly of the malware binary. If a program's use of memory matches a template, it is recognized as a virus. The template abstractions are not susceptible to simple variations on memory locations or constants. For example, the detector can deal with substituting `ebx` instead of `eax` inside a specific instruction. However, their method is dependent on successful disassembly of the binary (a difficult problem in itself), and their templates are dependent on the ordering of memory locations. An attacker who crafts a variant whose memory usage does not match a template can evade detection.

Many malware detectors must disassemble the malicious binaries to produce assembly code to analyze. Relying on static analysis is not a good approach when virus authors attempt to intentionally and successfully write malware to thwart disassembly [Szor, 2005a].

Lakhotia et al. give examples when malware attempts to thwart static analysis by obfuscating call instructions [Lakhotia and Kumar, 2004]. They explore ways to statically detect obfuscations used in metamorphic viruses by modeling the dynamic usage of the stack. More complicated stack use through memory can circumvent their measures. For example, many viruses obfuscate call instructions to break static analysis [Lakhotia and Kumar, 2004].

Using an abstract stack, Lakhotia et al. store the addresses of instructions that perform the stack operations push and pop [Lakhotia and Kumar, 2004]. They do not claim to be able to detect all stack obfuscations and instead suggest adapting the approach of Balakrishan and Reps [Balakrishnan and Reps, 2004] to create a complete system for detecting obfuscations. Balakrishan and Reps attempt to identify high-level data objects in a program given a binary, and much of their analysis is based on common stack manipulations. It is not clear how their approach could handle a malicious binary that may use the stack in a non-standard way. The abstract stack graph approach [Lakhotia and Kumar, 2004] is good for catching certain obfuscations of call instructions, but more work is needed to show that they can handle more general forms of obfuscation or if the adaptation of the Balakrishan and Reps algorithm is feasible for this application.

In another approach, by attempting to match a canonical form of the virus created by reversing some obfuscations, variants are matched even after a reordering obfuscation [Mohammed, 2003; Lakhotia and Mohammed, 2004]. In some sense matching the ordering of statements in a program is similar to Christodorescu et al.'s templates in that they seek to identify virus code using a more abstract form that is resistant to known obfuscations. Like the semantics-aware malware detector, the stack operations are abstracted in order to deal with obfuscated stack operations that can be used to perform a call, and they deal with reordering obfuscations by working with a hierarchical ordering of statements within the virus. While they are successful at decreasing the number of reorderable statements in some programs, this approach is difficult in general [Mohammed, 2003; Lakhotia and Mohammed, 2004].

The problem of determining if an arbitrary program has a particular malicious behavior is undecidable [Cohen, 1987], so there will always be opportunities for virus authors to evade static detectors. Even recognizing known viruses that have been obfuscated is not generally possible statically. For example, in relation to dead-code insertion, determining if a sequence of instructions is equivalent to a `nop` is an undecidable problem. For the problem of instruction substitution, determining if two arbitrary sequences of program statements are equivalent is also undecidable.

2.4.2 Behavioral Detection

Our disk-level virus detection is similar to a form of behavioral detection known as behavior blocking. Behavior blocking observes the actions of a program and intervenes when malicious actions are detected. Behavior blockers hold much promise, because they can detect new malware that would not be recognized by static signatures.

Szor identified two problems typically associated with behavior blocking detection techniques: detection of slow infections and circumvention of monitoring points [Szor, 2005a]. A third problematic area is resource exhaustion. All of these areas cause problems for all commercial AV solutions today. Moving detection to the disk processor avoids the circumvention problem, and enables at least partial solutions to the other problems (as we discuss further in Section 6.3).

One feature that distinguishes run-time behavioral detection from host-level signature detection is that behavioral detectors do not detect malware until it is actually running. This is in contrast to other techniques that attempt to detect the malware before it executes. Run-time detection makes it easier to monitor and observe behavior that static analysis may not be able to reveal. Behavioral detectors must be careful to catch the malicious activity and respond appropriately when malware is detected.

Another approach is to deploy behavioral detection by running a potential virus in a sandboxed environment before it is allowed to executed on the host. Symantec's Bloodhound technology monitors system call behavior while executing malware in an emulated

environment [Symantec, 1997a]. When the emulated OS API is called, the behavior is recorded. After the emulation, the emulated program's recorded behavior is analyzed by a heuristics-based expert system to determine if the program is actually malware. The Bloodhound system could recognize 80% of new and unknown executable file viruses. One problem with Bloodhound is the emulation overhead, and it is cited to potentially be slower than a static detector [Symantec, 1997a]. For instance, one problem is that the detector first must identify the most important areas of the file to scan to speed up the scanning. If the Bloodhound system missed an area of the file to scan, this could cause a false negative.

Behavioral detection depends on first defining a malicious behavior and then reliably detecting that malicious behavior. A related problem to defining the malicious behavior used with behavioral detection is that the virus may not exhibit its malicious behavior when a malware analyst runs the virus in an emulated environment to create a signature, even though it will later perform malicious actions on the host. Malware may not demonstrate certain malicious behaviors unless a specific trigger is executed. Triggers include the time of the day, specific files on the system, registry entries, etc. Malware may also behave non-deterministically, so it may require many executions before its malicious behavior is revealed.

Moser et al. addressed the problem of observing malicious actions triggered on certain conditions by building a system that tracks input inside the monitored program and records the program's state when a branching point occurs based on certain input [Moser et al., 2007a]. Later, the program's state can be restored from a previous branch, and another path can be taken. This automated dynamic analysis reveals more execution paths (and likely more malicious execution paths). This type of work is useful to any malware detection based on dynamic analysis of behavior (including ours) and would be beneficial to making detection more robust through better signatures that are based on the analysis of observed behaviors.

Behavioral detection has been used to identify other malicious programs in intrusion

detection systems and in spyware. These example systems provide additional lessons in the design of a behavioral detector. Ilgun et al. developed state-based transition diagrams using host-level states to detect intrusions [Ilgun et al., 1995]. Once a sequence of state changes can be recorded and specified (e.g., modifying the permissions of a file), the specified intrusion can be matched. By using a table, each row corresponds to a state machine signature, and an intrusion is matched by matching every entry in that row to an event (the final row entry is the matching state). Once a compromise is detected, the intrusion detection system can alert a security officer via a special interface. The authors recognize that the effectiveness of the approach is limited by the auditing mechanism being able to detect the compromised state while the host itself is compromised.

Eckmann et al. developed STATL, a domain independent language that can be used to specify an intrusion after a specific application intrusion is recorded and identified [Eckmann et al., 2000]. States are current system snapshots, and transitions represent system events (e.g., sending a UDP packet). Their language is rich in its ability to use describe different host and network attack scenarios including attacks on a ftp server or a TCP denial of service attack. While our work monitors events at a much lower level, we also use a state-machine to detect a compromise. Their language is much larger than what we need to express our signatures, but their use of state machines to detect high-level intrusions with event-based states is similar.

Some intrusion detection systems (IDS) first categorize normal behavior in certain programs, and then identify any previously categorized program behaving abnormally as malware [Forrest et al., 1996; Hofmeyr et al., 1998]. The IDS observes benign processes and records their normal system call usage behavior. An intrusion is detected when a program deviates from its previously recorded normal system call usage. The system records sequences of system calls, and then compares system calls of the same length from a new trace. An anomalous behavior will be matched when a sequence of calls from the new trace that do not have a match in the normal system call trace.

One way an intruder may attempt to avoid such IDS systems is by constructing an

attack that does not appear to deviate from normal system call usage. Wagner and Soto introduce *mimicry attacks* that can defeat these types of defenses [Wagner and Soto, 2002]. Because the detector is based on matching sequences of system calls, it can be defeated if an attacker can insert system calls that have no disruptive side effects. For example, if the detector would treat two consecutive `read()` calls as anomalous, an attacker may issue the sequence: `read(), open(), read()`. The `open()` may fail, but the attacker does not care. Many system calls are available to an attacker that produce no side effects. These mimicry attacks are similar to a slower type of attack where the attacking code does not immediately carry out the attack to avoid detection. We consider slower types of attacks in Section 6.3.

Another example of behavioral analysis is in the detection of spyware. Kirda et al. [Kirda et al., 2006] use a combination of static and dynamic analysis to determine if installed browser helper objects (BHOs), libraries used by the browser, are spyware. Their research provides yet another example of dynamic behavioral detection, and the difficulty of detecting malware at the host-level. They consider a BHO to be spyware if the BHO interacts with the browser to monitor user behavior and if the BHO suspiciously calls the Windows API to leak data about the monitored behavior. They use dynamic analysis to determine which pieces of code in the BHO handle interactions with the browser, and they use static analysis on the code found by the dynamic analysis to find invocations to the Windows API in response to browser events. Using both static and dynamic analysis, the techniques achieve a lower number of false positives than relying on just the static or dynamic approach.

For spyware to evade detection, it must cloak its monitoring of the user or avoid the flagged Windows API calls. Their spyware analyzer [Kirda et al., 2006] is able to detect unknown spyware, but their detection can be evaded. This illustrates a problem of a behavioral detector in that the accesses to the protected resource (malicious use of the browser in this case) can happen without mediation by the detector. One possible method for the spyware to operate without detection is by using browser functions to pass data to

an attacker-controlled server instead of using the Windows API. Other possible avenues of attack are using covert channels or code obfuscation to thwart static analysis.

To combat code obfuscation, they use dynamic taint analysis in order to track sensitive data through Internet Explorer and its BHOs (i.e., the possible spyware [Egele et al., 2007]). When sensitive data is leaked because of a BHO, then the BHO is considered to be spyware. The behavior-based dynamic taint propagation by Egele et al. added one byte of shadow memory for each byte of emulated physical memory. Their goal is to comprehensively analyze a given sample of spyware. Malware analysts having to analyze hundreds of samples of malware each day can benefit from the analysis their tool performs. Unfortunately, while the analysis information is helpful, the monitoring points are still vulnerable to evasion and may miss key malicious events in a compromise (e.g., through imprecise tainting).

Despite these exceptions, host-level malware detection is mostly limited to the static approach, since it is not possible to dynamically observe malware behavior at the host level with high precision without incurring high overhead. Malware emulators are resource constrained by time and space, as well as the fundamental limitations of dynamic analysis to only observe a subset of program paths. We address these programs by performing detection at the disk processor so it can always be running without disrupting normal execution, using simple general behavioral rules (Chapter 4) and behavior-specific signatures (Chapter 5).

2.4.3 Outside Host Detection

Although this is the first work to detect viruses using behavioral rules checked by the disk processor, previous works have used outside host detection to thwart malware using both the disk and other co-processors. Silberstein [Silberstein, 2004], Tarari [Tarari, 2006], and Symantec [Hile et al., 1994] have proposed offloading virus scanning from the main host using dedicated co-processors. Our approach is focused on improving detection precision,

not on improving performance of existing detectors.

Pennington et al. argued for moving intrusion detection into storage systems and developed a prototype storage-based IDS system inside an NFS file server [Pennington et al., 2003]. Intrusion Detection for Disks (IDD) [Griffin et al., 2003] extends server-side intrusion detection [Pennington et al., 2003] by implementing Tripwire-like [Kim and Spafford, 1994] rules on a client machine instead of a server [Strunk et al., 2003]. In IDD, an administrator on another machine specifies static rules (such as monitoring new files in the `/sbin` directory) to the disk via an encrypted channel that passes through the host. These rules are transmitted from an administrator's machine, through the host OS, to the file server over a trusted channel. This approach is well-suited to machines that are managed by a system administrator on a trusted network, but is not suitable for our scenario where the host machine may be compromised. In addition, their work is based on specifying access control policies on writes, not on recognizing characteristic sequences of disk requests. Such an approach could not detect new viruses unless they target files that are protected by access controls. They could have a rule to protect all executable files, but this would raise many false positives (see Chapter 4.1) forcing the administrator to deal with each of these alerts.

Zhang et al. suggested using a secure coprocessor for intrusion detection and demonstrated its effectiveness using a Linux kernel module [Zhang et al., 2002]. This secure coprocessor is much less common in desktop machines than a disk drive. Garfinkel and Rosenblum introduce an intrusion detection system (IDS) on top of a Virtual Machine Monitor (VMM) that monitors a host OS for intrusions [Garfinkel and Rosenblum, 2003]. The isolated IDS interfaces with the VMM to glean information about the executing guest OS and then enforce a policy based on the guest OS actions. They must constantly monitor the guest OS for compromise, and this causes them to incur overhead by suspending the VMM. Although this approach is able to defend against certain attacks, there is a threat that their monitoring point is not seeing the true state of the system. Others have improved upon the VMM introspection approach by detecting malware through a comparison of the

events seen inside the guest OS and outside the guest OS [Jiang et al., 2007]. This assumes that malware will hide certain events that will be caught at the VMM, but this may not be correct. As does the original VMM introspection work, their system relies on a guest OS adhering to their semantic view, but some malware could significantly change the kernel in a way that would render their semantic view of the guest OS incorrect.

Copilot [Petroni Jr. et al., 2004] implemented an intrusion detection system for rootkits on a commodity desktop PCI card and monitored a host's behavior by observing kernel memory at run-time. Their approach incurred high overhead to sample kernel memory. They recommended sampling every 30 seconds for 1% overhead. This is effective for detecting rootkits, but a very low sample rate for virus execution. In our experiments, we found many viruses that can infect large portions of a hard drive in 30 seconds. An evasive virus could be constructed to avoid detection by completing all of its damage and erasing itself from memory before the sampling occurs.

Newer work by Petroni and Hicks checks the control-flow of the kernel for malware [Petroni and Hicks, 2007]. They found that 96% of 25 Linux rootkits violate the kernel's control-flow integrity. The monitoring software is separated from the monitored OS by checking at certain points of execution. The control-flow of the kernel is checked by ensuring the kernel text remains unchanged and verifying all reachable function pointers. While promising, their approach requires annotation to the kernel source to find its function pointers, and a missed annotation would open the door to compromise.

2.5 Summary

Malware detection remains a challenging and important problem, and traditional approaches appear insufficient to tackle increasingly sophisticated malware. A disk-level detector has a big advantage over traditional host-based approaches: it will always see updates to the disk while a traditional detector is not guaranteed to see malware activity in the host. Because the host can be compromised, traditional host-level detectors can be

rendered useless while a disk-level detector can continue to function even after the host is compromised.

Comparing our solution to traditional host-level approaches reveals the two approaches are not disjoint but rather complementary solutions. We cannot deal with in-memory compromise, and our solution can better deal with malicious disk-level attacks by focusing on disk access behavior instead of static properties. Dynamic solutions (e.g., a disk-level detector) cannot detect malware unless it is executing, static detection approaches can be evaded by viruses that alter their code as they propagate. While host-level detectors must endeavor to win the arms race to maintain their own integrity, our approach transfers the monitoring point outside of the host.

Instead of analyzing code in an emulator or matching the code against static signatures, our disk-level detector dynamically monitors disk requests. Our focus is on viruses, but any malware that uses the disk can potentially be identified by a disk-level detector whether it be virus or trojan.

Chapter 3

Detector Design

This section presents the detector's design and describes the components that are needed to make a disk-level malware detector. Section 3.1 introduces the disk processor and its use in a disk-level malware detector. The following section (Section 3.2) describes how a disk-level detector would gain the higher-level semantic information it needs to implement our detection rules. After describing the disk model for our disk-level detector, we introduce the map that gives context to the low-level disk requests (Section 3.3).

Since getting code to run on the disk processor requires updating undocumented software (the disk firmware), we evaluate our design using a proof-of-concept implementation in which a separate PC acts as the disk detector as described in Section 3.4. Section 3.5 details the cost of providing the semantic information to a disk-level detector, and we end our discussion of the detector's design by presenting designs for updates (Section 3.6) and recovery and response (Section 3.7).

Our goal is not to supplant host AV detection, but to instead support it with the disk when applicable. Our approach only works for malware that uses the disk; other techniques must be used to thwart worms that use the network as their primary vector or memory-resident viruses that infect running processes in memory. We target our work on viruses that infect executable files, where an "executable" is defined as a file that adheres to the Microsoft PE/COFF file format (this includes exe, obj, dll, and sys files) [Microsoft,

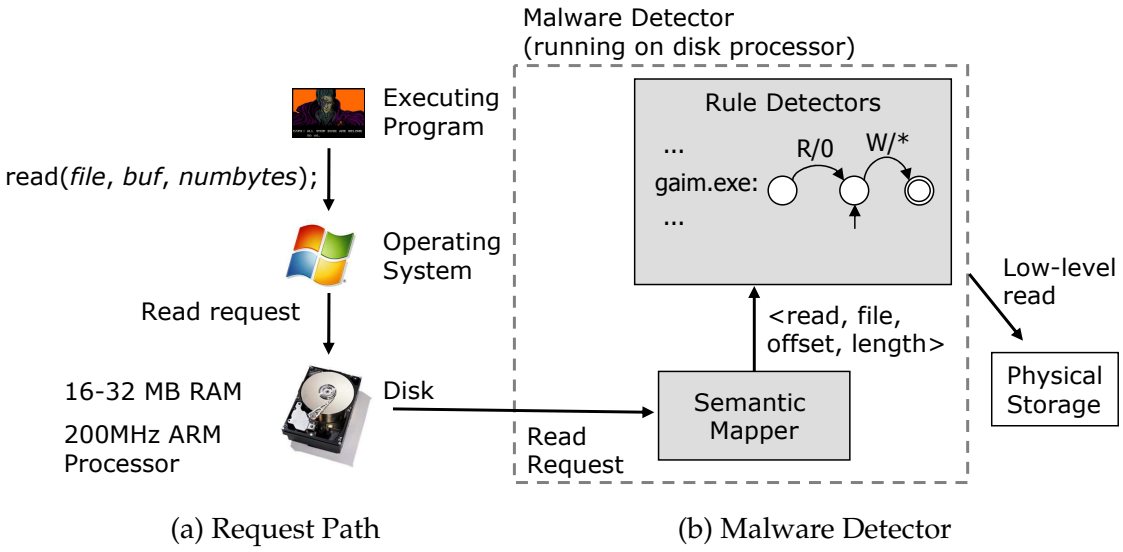


Figure 3.1: A Malware Disk Detector

2006]. Application-level viruses like Microsoft Word Macro viruses are not in our target class; these threats are better dealt with by the target application.

3.1 Detector Design

We propose to use the disk processor to recognize and respond to disk-based malware as shown in Figure 3.1. In the left side of Figure 3.1(a), we depict a top-down view of an application running on a machine incorporating disk-level malware detection. The application makes a request to read or write from some file on the disk.

A normal disk processor will receive the request and carry out the requested action, returning the result of a read to the host. The disk detector, depicted in Figure 3.1(b), processes a request by updating the state of the malware detector. In order to recognize malware, the detector needs more information about the request than just the sector address, for example it may need to know the name and type of the corresponding file. The semantic mapper maintains information on the file system running on the disk and maps a low-level request into a meaningful file system-level request including a file and offset

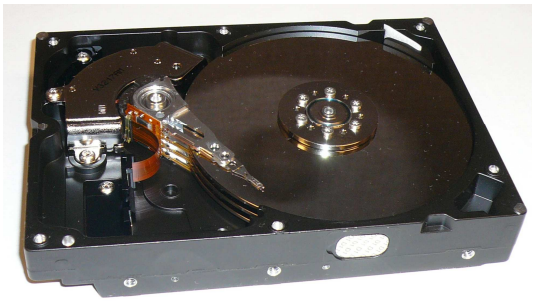
(Section 3.3 describes this process in more detail). Then, it updates the state machines according to the request. There is one state machine for a general infection rule for each active executable file (described in Section 4.1), and there are additional state machines for implementing the behavior-specific signatures (Chapter 5).

The state machine depicted in Figure 3.1(b) is a simple behavioral rule that detects file infections instantiated for the executable file `gaim.exe`. After the matching read request, the state machine has advanced to the second state, where a matching write request will be recognized as an infection. Other state machines similar to this one exist for all other previously read executable files. Additional state machines are maintained for the behavior-specific rules (described in Chapter 5), but these typically need one state machine for each rule that keeps state on multiple files, not for a single executable file. When a state machine reaches an accepting state, the malware has been recognized. The requested disk access is rejected by the disk, and the response process is initiated.

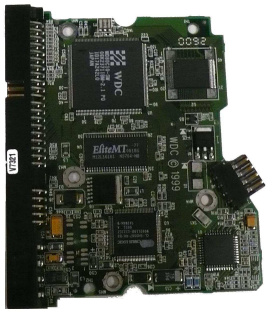
3.2 Smarter Disks

Due to advances in semiconductor technology, the processing power of the disk's electronic components, especially the hard disk controller, has been steadily increasing. For example, many Seagate disk drives use a 200 MHz to 250 MHz ARM processor as the hard disk controller [ARM Enterprise, 2007; ARM9E Family, 2007]. During the time required for a typical seek (3 ms to 22 ms [IBM, 2007; Toshiba America Information Systems, Inc., 2007]), the disk processor can execute hundreds of thousands of instructions while the drive head is moving to the necessary location. Disk processors also include a sizable amount of on-board embedded DRAM memory, typically 16-32 MB. This memory is typically used to cache data to provide faster access times to requests. It can also be used for disk activities besides servicing requests.

In Figure 3.2(a) the main disk components can be seen including the disk arm that reads the data from the disk platters holding the drive's data. The disk arm is powered by



(a) Components



(b) Circuit Board

Figure 3.2: Parts of a Disk

a voice-coil motor at the base. The internal circuit board of a laptop hard drive is shown in Figure 3.2(b).

As a result of these trends, we have reached a point where disk drives are now powerful enough to be data processing devices, rather than merely be used for data storage and transfer. Such *active disks* have previously been used to boost the performance of data intensive applications [Acharya et al., 1998; Riedel et al., 1998; Keeton et al., 1998]. Disks with processing capabilities are now appearing on the market, such as the Seagate Momentus disk drive [Seagate Press Release, 2005] that provides encryption capabilities.

There are other commercial offerings that perform computation near the disk including EMC Centera [EMC, 2007], Netezza Performance Server [Netezza, 2007], and Exegy TextMiner [Exegy, 2007]. Although these products use more powerful CPUs (Ghz) than are available on typical current disk processors (200-300 MHz), they provide examples of problems solved by moving computation closer (or tightly integrating) with the disks. Our strategy will be similar in using the disk as a malware detector.

Another notable use of disk processors is Seagate's recently introduced new technology dubbed DriveTrust [Seagate, 2006]. They highlight four main parts: an enhanced firmware, a trusted send/receive command set (i.e., a trusted channel), secure partitions,

and a protocol to interact with a secure partition. Although specific details on their implementation are scant, this type of functionality could be used in various ways by a disk-level malware detector. For example, after detecting malware the disk detector needs a dependable channel to inform the user of compromise. To perform recovery, a disk detector can use a secure recovery partition similar to that provided by DriveTrust. Once malware is on the system, whether in memory or on the actual disk, protecting the keys is very difficult, because the malware can snoop or actively retrieve sensitive information (e.g., find a key on the disk).

3.3 The Semantic Gap

The disk-level detector must bridge the semantic gap between disk-level requests and host-level file system activities. A normal disk sees reads and writes to a segment of blocks, but has no knowledge about higher-level file system data. To implement disk-level malware detection, the disk-level detector needs enough semantic information to interpret disk requests meaningfully. This information includes mapping physical locations on the disk to files and knowing each file's inode location on the disk. Knowing this information allows the disk to derive higher-level file system semantics like a create or write to a specific file in the file system. There are two ways for a disk to gain the semantic knowledge of the file system: through a change in the disk interface in which the OS explicitly gives the semantic information or by implicitly inferring it based on what is stored on the disk and the processed requests.

3.3.1 Richer Interfaces

One solution to the semantic gap is to use a richer OS-disk interface that is less complex in its implementation than using implicit inference techniques to bridge the semantic gap. The alternative interface to inference is providing extra semantic information explicitly through the disk interface. This would necessitate a change in the disk interface and a

change in the file system to use the new interface [Sivathanu et al., 2004a]. Changing the traditional OS-disk interface forces operating systems and disk manufacturers to change an interface they have been reluctant to change for decades; hence we describe these approaches here, but focus on a solution that does not require changing the host-disk interface in the next section.

Object-Based Storage Devices. One richer interface is that provided by the Object-based Storage Devices (OSD) standard [T10 Technical Committee, 2007; Mesnier et al., 2005]. OSD presents an object, rather than file, interface to the upper layers of software and allows attributes to be associated with the objects. Several OSD-based drives have been demonstrated [Anderson, 2005; Emulex, 2005; Cluster File Systems, Inc., 2007; Center, 2001] and OSD-based disk drives are expected to appear in the enterprise storage market within the next few years.

With OSD, the disk requests are made in the form of objects. If a program needs some data, then the data is requested as an object, and this allows the disk to optimize other low-level information like block placement. Having the disk manage metadata instead of the file system alleviates overhead at the host. Access controls may be enforced on the requested object, and the requesting program will need to provide a capability to access this object. With these object-based requests, the disk knows the high-level file system semantics, because requests are made at a higher level providing much more information about the request.

Type-Safe Disks. Another example of an explicit change to provide additional semantics to the disk interface is *type-safe* disks. Type-safe disks export an API to the file system to obtain the necessary semantic information [Sivathanu et al., 2006]. In addition to the traditional disk drive interface, they export an API that deals with pointers between blocks. An inode has incoming and outgoing pointers; a data block has incoming pointers; and a root block has outgoing pointers (e.g., a file system super block). The host can create and delete pointers through their API that allows a disk to explicitly track high-level data structures in the file system. Their approach extends earlier work [Sivathanu et al., 2004a]

by adding a more complete disk API.

The primary advantage of type-safe disks over OSD is that they resemble a more traditional file system lessening the implementation changes required for implementation. Some functions included in a type-safe disk are those that track block creation, set the block size, and track pointer creation. One function of the exported API is the deletion of a block pointer to track *liveness*. Liveness involves tracking a *cluster* (an NTFS cluster is the same as a block of eight sectors) of data or the cluster's use in a specific file and detecting a modification to either.[1]

One concern with any semantically-rich interface, is that if the host is compromised the information included in the disk request may not be legitimate. Malware can explicitly call exported functions in the API to change its disk-level behavior. This disk-level behavioral-morphing malware will be motivated by a wide deployment of detectors that can adapt to different attacks. However, the detector can be programmed to monitor for new behavioral changes, because it will always have the ability to monitor disk requests. Our design is meant to protect data even in the face of compromise, and we discuss malware purposefully changing its disk-level behavior to evade detection in Chapter 6.

Both type-safe and OSD disks change the disk interface to add disk-level functionality. It is likely to be several years before OSD is common on desktop systems, and type-safe disks have only been proposed in recent research. We do not consider their use further, and instead focus on developing a design that can work for current desktop systems.

3.3.2 Unchanged Interfaces

With some added complexity, we gain the advantage of not having to change the disk interface through an explicit approach, but the disk processor can still infer the same high-

[1]Hereafter, we will refer to an NTFS cluster as a block as both are terms for referencing a group of sectors (one block/cluster is typically eight sectors), and we will refer to an inode by its similar NTFS term, a Master File Table (MFT) record.

level events. Not changing the interface will speed adoption of a disk-level malware detector. For these reasons, we assume a standard disk interface in our prototype implementation, and we bridge the semantic gap by gathering the semantic information at the disk processor [Sivathanu et al., 2003; Arpaci-Dusseau et al., 2006]. To function correctly, the detector must infer the high-level file system operations from the limited interface provided by the disk requests. Because the source code to NTFS is unavailable, we cannot confirm the correctness of our assumptions on some of the detector inferences. We discuss this problem in more detail in Section 3.3.3.

The original *semantically-smart disk systems* (SDS) were implemented for the ext2, ext3, and VFAT file systems. Each file system directly impacts the amount of semantic information available to the disk, and they found that certain basic properties must hold in a file system to enable an SDS (see Section 3.3.3). A disk-level detector requires stringent properties to hold in the file system, because errors in inference can directly lead to false positives or false negatives. Their work indicates that the Microsoft Windows NT File System (NTFS) [Microsoft, 2003c]) should uphold the required file system properties of an SDS [Sivathanu, 2005].

Opting to keep the interface unchanged, we use an implicit method to infer file system activity in the detector similar to an SDS [Sivathanu et al., 2004a]. Our prototype implementation builds the static semantic map by analyzing the disk contents, but does not update it dynamically (see Section 3.4).

3.3.3 The Semantic Map

At the disk-level interface, the requests for data are reads or writes in the form of <*read*, *sector*, *number of sectors*> (or <*write*, *sector*, *number of sectors*, *data*> for a write). Because the disk enforces rules and signatures based on files and file offsets, the disk must be able to map a sector to the file (and corresponding file block offset) to which the sector belongs. Using this *semantic map* of sectors to files, the disk can quickly apply a rule or signature after using the semantic map.

One challenge in using a semantic map is that the disk must understand the file system structure to build the *semantic map* associating sectors to files. A file is made up of blocks (typically a minimum of eight sectors or one block containing 4 KB) of data, and the detector will be inferring actions on that file and its blocks from the accesses it sees on disk sectors. When the OS makes a request to the disk for some data starting at a sector s, the disk must map s (and the request's data) to the associated file (or files when a request crosses a file boundary). Thus, we can think of the semantic map as a function:

$$\text{map (disk sector address, num sectors)} \rightarrow \text{<file id, offset, num blocks>+}$$

Using s and the number of sectors accessed, the disk performs a fast lookup in the semantic map to find the file (unique file id), the offset, and the number of blocks accessed for each file accessed in the request. If any file is an executable, the detector will attempt to match a new state in one of its behavioral rules (and possibly any behavior-specific signatures).

Building the Initial Semantic Map. To construct an initial semantic map, a disk-level detector must first understand the file system data structures on the disk (i.e., for NTFS, the locations of the MFT records and their structure). The disk can analyze the needed data structures, the file system metadata, to build the semantic map. In Windows, each file has an associated MFT record (similar to inodes in ext3). The disk will parse the file system's metadata (MFT records) using the metadata as an index to each file in the file system.

The disk builds the first semantic map immediately after the disk has been formatted with its file system (NTFS in Windows) and stores the map in a protected area on the disk. A disk format happens when the disk is configured to install a specific file system. All the data structures that make up the file system are created and written to disk, and the data previously on the disk is lost. For NTFS, some of the actions include creating and initializing the MFT, setting a default cluster (block) size, initializing the log file, creating

a list to store the known bad clusters, and creating and initializing other special files and data structures [Altaparmakov et al., 2007]. For a longer format, the disk may fill empty disk blocks with zeroes, but a quicker format will skip this initialization step.

Immediately after a disk format, any requested sector s of an existing file will have a correct semantic mapping to a file, f, f's offset, and the starting offset to the blocks accessed, o. At each bootup, the disk can load the semantic map from this protected area.

Updating the Semantic Map. After the disk has built the semantic map, the disk must also maintain this map to ensure that it can correctly enforce behavior rules and signatures. As data is read or written after the initial mapping, the mappings will change, and this must be reflected in the semantic map.

For example, a disk must infer when a file system block (of sectors) is allocated or deallocated, because it is enforcing rules on files that are made up of blocks (of sectors). A process can append or truncate a file's data. With stale information about the appended or truncated file's data (blocks), a disk-level detector could generate a false positive or false negative. The challenge in keeping this mapping maintained is inferring the file system semantics from the low-level reads and writes of the disk sectors. This is made possible by the disk processor's analysis of the file system data structures.

Although NTFS seems to write out updates in a timely fashion, we cannot know for sure if the inferred file system events will always be a timely reflection of the file system when dealing with an undocumented system. Tests can reveal a violation of our assumption, but they cannot prove our assumption correct. Correct implementation of a disk-level detector must verify that the file system adheres to our assumptions. Some of our mechanisms for maintaining file system information address this problem. One such mechanism, the file system's journal log file, is undocumented and not well understood, but it can help in maintaining the semantic map. Using and understanding the log file would require reverse engineering its format or perhaps forming a partnership with Microsoft to create a detector.

Immediately on boot, the semantic map is loaded, and the disk requests will begin.

The disk processor can monitor all accesses to the storage device, and this ensures that any persistent data will be seen by the disk processor.

As the disk services its requests, the disk's data, or rather the file system, changes, and the semantic map will become old without constant updates. For NTFS, a file can be created, renamed, truncated, appended, etc., and all these actions will change file system metadata and data. For a disk-level detector, it needs to infer reads, writes, and creates. Other file system events may be potentially interesting, but we did not use them in our behavioral rules or behavior-specific signatures.

Each of the higher-level actions in the file system can be broken down into lower-level disk requests. Consider when the file system creates a file. The file system will need to create a new MFT record for the file, write an index entry into the directory that will include its filename and MFT record number, set some bits in the parent directory's bitmap data structure that keeps track of the in-use file entries within the directory, and set another bit in the file system-wide bitmap data structure that keeps track of in-use MFT records [Russinovich and Solomon, 2004; The Linux-NTFS Project, 2007]. As it journals the metadata, NTFS will write the metadata journal log records before it commits the final writes to the actual metadata. The disk can use the writes that happen before the final commits to track modifications and keep the semantic map up to date [Russinovich and Solomon, 2004; Sivathanu et al., 2004a].

Recognizing File System Events with the Semantic Map. In using the semantic map, the detector can recognize different events. For example, as the detector sees the MFT journal log records written, it will know that the create file system event occurred. Using either the new MFT log record modification or the new index created in the parent directory's MFT record, the disk can infer the newly created file and attempt to match the event with a rule or signature. If the detector triggers an alert, one possibility for response is that the disk could block the commit of malicious journal records from being written to disk, and recovery could begin.

Deletes are similar. If the detector were to see that a directory bitmap (tracks in-use

MFT records) or the file system bitmap (tracks in-use blocks, or the NTFS term, *clusters*) having one or more bits cleared, then the detector will know that a delete is about to occur, and it can update the semantic map by clearing the appropriate mappings used by the deleted file. For the truncate and append append operations, metadata is affected in a similar way. The file system bitmap and file's MFT record edits can be seen in the journal's log file prior to the commit of the data. With this semantic information, a disk-level detector can enforce general behavioral rules (see Chapter 4) and behavior-specific signatures (see Chapter 5).

3.4 Implementing a Prototype Detector

Since we are not able to modify the code on the disk processor, we evaluate our approach by building a prototype implementation using an external machine. Our primary goal in building the prototype is to evaluate the effectiveness of our behavioral rules. Using a separate machine to process all disk requests accurately models a real disk detector, while minimally disrupting the disk request behavior. This experimental setup would not be useful for timing analysis (e.g., the acting disk detector has a processor well over seven times the power of a normal disk processor), but it models a standard disk by using a fully compliant SCSI protocol, the iSCSI protocol. The local machine treats the remote disk no differently than its local disks [Radkov et al., 2004].

Most modern file systems are organized with data and metadata consisting of data units, or blocks, of size 4KB (and some times larger). Because most malware targets Microsoft operating systems, we assume the NTFS file system for our detector. Figure 3.3 depicts our setup of Windows running on a remote VMWare partition residing on a disk inside a Linux machine.

Our model is similar to IDD by Griffin et al. [Griffin et al., 2003]. In our disk detector prototype, a Windows machine (possibly running malware) sends disk requests using an iSCSI 4050c QLogic card [QLogic, 2007] to a Linux machine that acts as the disk detector.

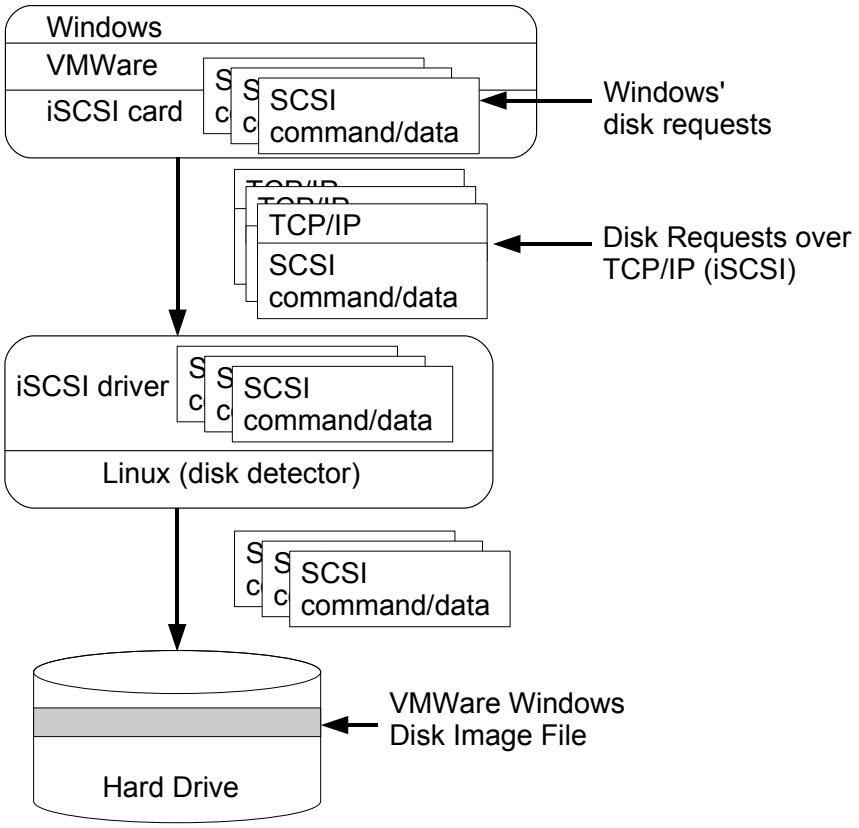


Figure 3.3: Prototype Setup

The Windows machine makes disk requests through the iSCSI protocol which is built on top of the TCP/IP protocol [Satran et al., 2004]. The QLogic card offloads the TCP/IP processing from the Windows host, and the powerful Linux machine, acting as the disk detector, must reassemble the SCSI commands and data packets before sending them to the disk.

The Windows host runs in VMWare (top of Figure 3.3) in order to boot from partition exported by the Linux machine over iSCSI, and Windows has a low-level ext3 driver installed to allow Windows to interact with the ext3 file system that houses NTFS. On the other machine that represents the disk, we modified the SCSI driver in Linux kernel 2.6.18 to examine each disk request just before the request is issued to the disk. The detector is inserted as a Linux kernel module. When a disk request is received in the modified Linux

SCSI driver, the driver sends the request to be checked to the detector. The detector uses the semantic map to map it to a file and offset, and then updates the relevant state machine(s) if necessary. If a state machine moves into an accept state, then the detector will then respond to the matched virus (at this point, the prototype detector just logs the match for further analysis). Otherwise, the detector allows the request to proceed as in normal operation.

The proof-of-concept detector creates a semantic map at boot time, but does not update the map during execution. This means if the file system is modified after booting, the semantic map may be incorrect. Our experiments were designed around this limitation, and they did not depend on dynamically updating the semantic map. The potential inaccuracies in the semantic map do not invalidate our experimental results, since it is straightforward to verify none of the results were impacted by incorrect mappings. We can check a false positive by inspecting the relevant disk information that causes an alert (e.g., checking the flagged file for an infection in many cases). Likewise, underreporting of false positives can occur if the mapping is incorrect (i.e., a correct map would report more false positives). This case is difficult to detect, but we can increase the likelihood this does not happen with repeated testing (each time a test is done the mappings change). We can check false negatives by examining files placed on the disk for the sole purpose of infection (known as *goat files*) to see if their infection is missed.

3.4.1 Building a Semantic Map

Our detector statically gathers semantic information (e.g., locations of executable files) at boot time in order to build the semantic map table. The MFT indexes each file, and the specific file information including its location on disk, its file size, and its file name can all be read from the file's MFT entry. By using the MFT records, the detector first builds a mapping of each file within the NTFS disk image to its corresponding data on the physical disk (e.g., sector 239832 maps to block address 8192, gaim.exe, and block offset 0).

As we traverse the MFT, we fill in the entries of the semantic map table. Each entry

in the semantic map table represents a file fragment and is of the form (block address, file id, b_{offset}, num_blocks_accessed). The block address is the file system block address (i.e. a logical cluster number [The Linux-NTFS Project, 2007]), the file id uniquely identifies the file, the block offset (b_{offset}) is the block offset of the first block accessed in this file fragment, and num_blocks_accessed is the number of blocks in this fragment.

Our implementation of the semantic map is complicated by our experimental setup. To test a detector without access to the disk's firmware, our setup uses a separate Linux machine as the disk detector. By forcing the test machine to use a Linux-controlled remote disk, we can modify the Linux kernel to act as the disk detector, but this makes creating the initial semantic map more complex. These mappings do not change the semantic map, but it does change the initial calculations in building the semantic map.

Using the ntfsprogs utilities and the Linux-NTFS Project's documentation, we gather file locations in Windows [The Linux-NTFS Project, 2007]. Because we are working with a VMWare disk image, we mount the disk image as a loopback device to use with ntfsprogs, and the prototype then calculates the indexes into a table that maps the files as an offset from the beginning of the disk image. Later, when monitoring disk accesses, the detector can semantically map a sector s by using this table.

Using the Semantic Map. A main piece of the semantic map is the starting file system block of each semantic map file entry. Given a sector address, s, the semantic map can find the entry that corresponds to the sector address, but the relationship between the sector address and a semantic map entry is the file's physical location on disk. The file's physical location is the combination of many offsets:

$$s = \mathrm{s}_{\text{partition offset}} + \mathrm{s}_{\text{ntfs offset}} + \mathrm{s}_{\text{block offset}}$$

These offsets include the partition's offset relative to the beginning of the disk ($\mathrm{s}_{\text{partition offset}}$), the file system's offset relative to the beginning of the partition ($\mathrm{s}_{\text{ntfs offset}}$), and the file's sector offset relative to the beginning of the file system ($\mathrm{s}_{\text{block offset}}$).

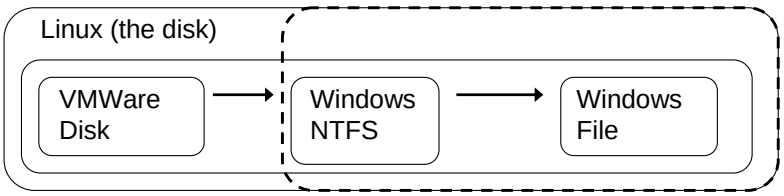


Figure 3.4: Disk Detector Prototype Storage Setup

Inside the disk-level detector in our implementation (the Linux machine), we use a semantic map that represents the local disk of the test machine. This setup is made possible by running the test machine on a Windows VMWare image located on the remote disk. The semantic map maps sectors to their corresponding files, and the relationship between a sector address and file system blocks in the semantic map is the combination of multiple levels of indirection as with the `Windows File` shown in Figure 3.4.

While Windows is running within VMWare on a remote Linux disk partition, there is an extra level of indirection to calculate a block address from a sector address, because the Windows disk image is actually a large file encapsulated inside the remote file system. In addition to the offset from the disk image to the file system, there is the file offset to the beginning of the disk image's file.

The offset from the `Windows File` in Figure 3.4 to the beginning of the VMWare disk image is not a simple subtraction calculation using the disk's image sector address. There is an additional layer of indirection because the VMWare disk image is a file itself that can become fragmented. Different sectors of a fragmented file will lie at different offsets from the disk's image initial sector.

Thus, a sector address, s, of a file in the semantic map, is the sum of the following offsets:

$$s = s_{\text{partition offset}} + s_{\text{VMdisk offset}} + s_{\text{ntfs offset}} + s_{\text{block offset}}$$

The offset of the VMWare disk image, $s_{\text{VMdisk offset}}$, lies at a location relative to the disk

partition's sector address, $s_{\text{partition offset}}$. From that information, we can compute the first sector of the Windows file system, $s_{\text{ntfs offset}}$, that is stored inside the Linux partition (NTFS starts at an offset from the beginning of the VMWare disk image file). The final file's sector offset relative to the file system, $s_{\text{block offset}}$, is the difference between the file sector and $s_{\text{ntfs offset}}$. These offsets illustrate the relationship between a disk sector address and the layout of the data on the disk in our implementation.

3.4.2 Implementing an Updatable Semantic Map

Using the semantic map allows the detector to implement the semantics to enforce the disk-level rules and signatures. A real disk-level virus detector would also need to update its semantic mappings dynamically as the file system changes (i.e., change semantic mappings as MFT records change). As discussed earlier, this is not done by our proof-of-concept detector (since it was not needed for our goal of testing detection feasibility), but it could be done using techniques described in Section 3.3.3.

Understanding the NTFS journal is necessary to implement an updatable semantic map. With the information supplied by journal records, the disk detector could then keep the semantic map updated while the disk is running. Recent research has shown inferring the file system at the disk-level is possible and applicable to different applications including, for example, the secure deletion of data [Sivathanu et al., 2003; Sivathanu et al., 2004a; Sivathanu et al., 2004b; Sivathanu, 2005; Sivathanu et al., 2006; Arpaci-Dusseau et al., 2006]. Using our implementation, we are able to test a proof-of-concept detector on different malware without needing to reverse-engineer a proprietary and undocumented file system journal.

3.5 Cost Analysis

We now present an analysis of the memory and computation required to implement a disk-level detector. This section analyzes the cost of maintaining the semantic map; Section 4.3

and Section 5.3 consider the costs of implementing the detection rules. For the disk model in our analysis, we assume 100 GB of storage with 16-32 MB of RAM. For a conservative estimate, we assume a 200 MHz disk processor (similar to a Pentium I).

The cost of the semantic map can be calculated by analyzing boot times, initial build-time, space for storing the semantic map, and the computational cost of maintaining it. Each block in the file system will need a mapping to its associated file, and the offset of that block in the file system. In our analysis, we consider a speculative design based on file system activity from a recent five-year study on Windows file system data usage [Agrawal et al., 2007].

Semantic Map Storage Costs. All of the semantic map costs depend on the size of a semantic map, and the semantic map's size is dependent on the file system's makeup, or specifically, its fragmentation. In a recent five-year study on Windows file systems, Agrawal et al. [Agrawal et al., 2007] find the average number of files on a PC (the NTFS file system represented 80% of file systems in their study) is 90K with an average file size of 189 KB. Figure 3.5 shows the number of file fragments against differently sized fragmented files (100K, 200K, and 300K file sizes) and against three file systems with varying numbers of files (50K, 75K, and 100K).

From Figure 3.5, the worst-case is 1.5M fragments with 20% fragmentation with an average file size of 300K and 100000 files. The average-case of 90000 files and a 189K average file size yields 500K fragments with 10% fragmentation. In practice, fragmentation can be kept fairly low. As done in Microsoft Vista [Microsoft, 2008], the disk could defragment files while the disk is idle as a possible solution to excessive fragmentation.

We would expect the actual number of semantic map entries to be lower than our given analysis, because the median number of files (52K) is 48% lower than our worst case. In addition to the expected lower file number, we also expect file fragmentation to be low given that most machines will use less than half of their storage space [Agrawal et al., 2007]. All of these factors contribute to a smaller semantic map.

Each semantic map element is a file fragment of a sequence of contiguous sec-

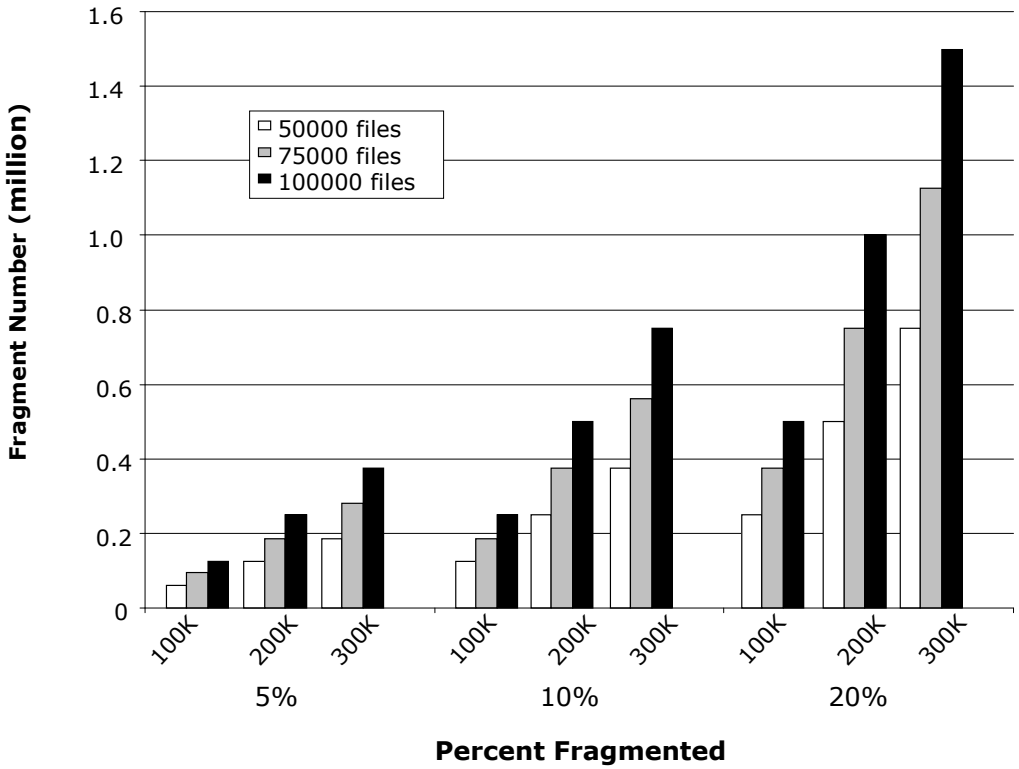


Figure 3.5: File System Fragmentation. This figure shows the number of file fragments with 5, 10, and 20% fragmentation across average file sizes of 100K, 200K, and 300K (average file size is 189K) and across a different number of files (50000, 75000, and 100000).

tors (or blocks). Given a single semantic map entry, (block address, file id, b_{offset}, num_blocks_accessed), the block address can be encoded in four bytes, the file id in 20 bits, the block offset (b_{offset}) in four bytes, and the number of blocks accessed in 20 bits. The total storage required for the semantic map is (13 bytes) $\times$ 1.5M = 18.6 MB for our worst-case analysis (300K file size, 100000 files, and 20% fragmentation).

A 18.6 MB semantic map consumes all (16 MB buffer) or much of the memory (32 MB) available in disks today. However, this is the worst case we would expect. The expected, or average, case would use only 6.4 MB memory (90000 files, 10% fragmentation, 189 KB average file size). Given the trend of increasing disk cache sizes, these costs should diminish over time.

Our implementation of a static semantic map supported much of this analysis including the speedup of representing contiguous sectors in a single semantic map entry and the small space required for semantic map storage. However, many parts of our implementation would need to be retooled to measure for performance bottlenecks. For instance, our fragmented file count, the number of fragments for each file, and the number of files were all low. Because we do not dynamically update the semantic map and to speed up the testing our implementation, we do not index every single file on the disk (e.g., data files). Out of 1569 executable files, our average number of fragments was 1.53. The highest fragmented file, a Norton Antivirus installation file, was 284 fragments. Without the Norton install file, the average fragmentation per file drops to 1.35. Although other lab machines' file system fragmentation had similar measurements, a more in-depth analysis of file system fragmentation across more machines is desirable.

Initial Build Costs, Boot Costs, and Computational Overhead. Building the initial semantic map is a one-time cost incurred after a file system format. This requires parsing the file system metadata in order to build a complete semantic map. To speed up this process, one could create a semantic map while the disk is being formatted, but we analyze a post-format semantic map creation.

When a file system is first created, there are very few files. Each MFT record will be stored as a contiguous set of 1 KB records on disk [The Linux-NTFS Project, 2007], and all non-empty MFT records will need to be processed. Given that modern hard drives can achieve internal data transfer rates of 125 MB/second [Seagate Technology, 2007], the time required for transferring the non-empty MFT records from the disk will take under a second (the initial set of used MFT records will not exceed 125 MB). After fetching each MFT record, the MFT information will tell exactly where on disk a file is located, and the semantic map can be filled in accordingly.

Once the initial semantic map is built, the map can be stored in a protected area across reboots. For a normal boot, the extra boot time is the time taken for the disk to load the semantic map from its protected area into memory. For our analysis, we assume the disk

can load the worst-case 18 MB map (or the expected 6.4 MB map) from a protected area of the disk, and the time for this load should be negligible (under a second).

Maintaining the semantic map requires parsing the journal's log file and then updating the semantic map. For our analysis we consider a file creation. In the log file, there are three separate sub-operations or update records for this file system event: allocating and initializing a MFT record, adding the file into the parent's MFT record, and setting the appropriate bits in the file system bitmap [Russinovich and Solomon, 2004]. Once the detector sees that a file is being created by the first of the records, it must update the semantic map using the new sectors that will be used. This will require inserting a new mapping into the semantic map. Assuming the semantic map to be ordered, the insertion time depends on the number, n, of semantic map entries. If the created file's data is fragmented into k pieces, then there will be at least k insertions for the data and three insertions for the metadata (adding file's MFT record, changing parent's MFT record, and changing file system bitmap) into a semantic map having approximately n indexes. Depending on the MFT update behavior, these updates could be grouped together.

When deleting a file, the detector can know which mappings of a file's data are deleted by reading the file's MFT record. The run-time of these deletions may depend on the number of entries in the semantic map (some entries may need to be shifted on the deletion). Other operations like appending and truncating involve similar operations as both creating and deleting a file.

Some algorithmic improvements could be done on semantic map operations at the cost of space. For instance, when inserting elements into the ordered semantic map table, recent results show the gapped insertion sort can be done in $O(n \log n)$ time with high probability by adding gaps in between elements [Bender et al., 2006].

3.6 Updates

There are two types of updates for a disk detector: signature updates and firmware updates. The disk's firmware uses behavior-specific signatures to identify malicious disk traffic, and corruption in either the firmware or the signatures could cause problems in the form of false positives or false negatives.

3.6.1 Updating Signatures

Since the behavior-specific signatures are designed to detect known malware, they will need to be updated frequently as new malware are discovered and analyzed. The signatures are stored in a protected area on the disk. Our updating mechanism should not rely on the host processor since it may have already been compromised by an undetected virus. The first step towards this goal is to use cryptographically signed updates. The disk vendor could embed a public key for checking signatures in the disk drive or in a TPM, and the disk processor could check the signature on an update using this key. This prevents invalid updates, but it does not prevent a compromised host from blocking updates from reaching the disk. Since the host OS controls the disk's access to the network, we have no way to ensure updates reach the disk (see Figure 3.6).

When we update the behavior-specific signatures, we face the problem of a denial-of-service attack without a dependable channel between the disk and the vendor making the signatures. Without a dependable channel, the vendor can issue a signed signature update, but an attacker could deny the disk of the update. The user may continue to use the machine while remaining unaware that malware is actively blocking the delivery of a signature update. An ideal solution would supply updates directly to the disk without using the host.

One possibility to address the problem of updating signatures is the use of a trusted BIOS. If the machine can provide a secure bootup, then the user could boot to a trusted BIOS and update the disk's behavior-specific signatures via an authenticated BIOS. This

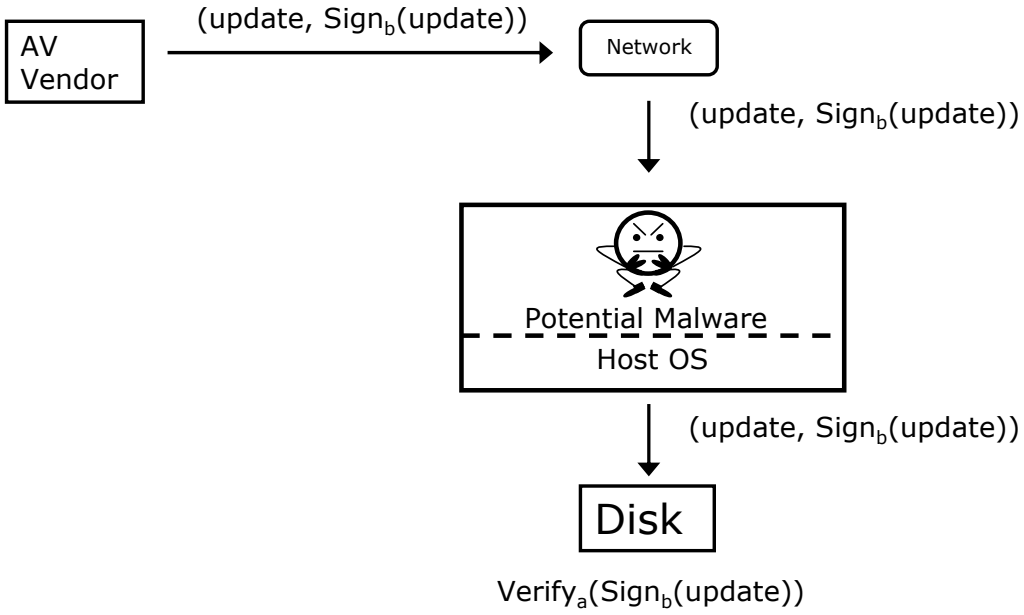


Figure 3.6: Updating Signatures.

increases the TCB to include the BIOS, but we get the dependable channel we need to update the signatures to the disk.

Another alternative may be suited to especially paranoid organizations: the disk drive could enter an alarm state if it does not receive an expected update within an established time frame. This would prevent a compromised host from blocking updates silently, but possibly at too great a disruption to the end user.

3.6.2 Updating the Firmware

There are times when signature updates alone will not be sufficient to address a specific type of malware. The frequency of this situation in modern AV engines has been lessened by supporting signatures that contain dynamic code to detect a specific virus [Szor, 2005a]. Processing the signature will cause the AV engine to execute the signature's interpreted, or executable, code.

If the signature language is not expressive enough to specify certain malicious behav-

ior, the language will need to be changed (e.g., specifying Ganda's use of two identical dropped files is not supported by our specification language), and then the disk's firmware will need to be changed to support the changed signature specification language. Disk vendors need to be careful of a firmware update. If a bug is introduced into the firmware, it could disable the disk. Similar to many popular devices including cell phones and TV set-top boxes that regularly update their firmware, the disk can also update its own firmware. By building an initial set of disk-level signatures, patterns should emerge of common signature states. The disk-level AV engine can be made to support more general state properties, and firmware updates should be an infrequent operation.

In the event that a firmware update is needed, the normal process of updating firmware cannot be used, because it opens the firmware up to compromise. Normal updates to disk firmware are done by vendor specific commands [aimtrading, 2006; Wells et al., 1999]. The host first notifies the disk to begin the firmware update process. The disk receives the notification and copies its EEPROM contents (i.e., the firmware) to its Random Access Memory (RAM). After the copy, the disk will execute code now copied to RAM that will first erase the EEPROM and then check that the EEPROM has been erased. The EEPROM must first be erased before being written with the new firmware. Now that the EEPROM has been erased, the disk tells the host that it is ready to begin the firmware update. The host then begins writing the firmware update sector by sector to the disk via the standard ATA interface. The disk writes a sector to the EEPROM, reads back the sector to check the write's integrity, and then repeats the process until the update is done [Wells et al., 1999].

One potential solution is for the disk processor itself (without aid of the host) to perform the update after checking the update's signature. This solution assumes that an update would be issued via a CD or USB stick while the disk is not in use by the host (e.g., during normal operation). This does not solve the problem of malware stopping an update from happening, but it does stop an attacker's ability to modify an update. A direct channel from the user to the disk solves this problem. If we can change the disk architec-

ture and supply disk firmware updates directly to the disk without data going through the host (e.g., by adding a USB interface directly to the disk), then we could reliably issue firmware updates.

3.7 Recovery and Response

Recovering and responding to a detected virus is an important problem. Because the detector is running at the disk-level, it can prevent any writes from a suspected malicious program from reaching the physical media. For instance, the disk can store recovery data in a safe place that is inaccessible to the host. Many machines currently ship with a recovery partition that is hidden from the user, but a malicious program could corrupt this partition. A disk detector could monitor for malicious accesses to the recovery partition.

The main problem in both recovery and response is not being able to trust the host machine. Without being able to rely on the host, recovery must be initiated but in a way that minimizes or eliminates malware interference on the host. For response, the issue is similar in that the user must be notified of a breach.

3.7.1 Recovery

There has been a lot of work done on backups and versioning of data [Santry et al., 1999; Peterson and Burns, 2005; Cornell et al., 2004]. Many of the newer approaches to versioning take advantage of the rapidly decreased cost and availability of storage today. For many consumer drives available today, prices have dropped below 40 cents per GB [PC World, 2007]. The low cost of storage allows us to choose recovery designs ranging from backing up any data that are modified to taking periodic snapshots.

Unlike the aforementioned recovery schemes, we advocate putting the recovery mechanism below the file system and into the disk. Hutchinson et al. discuss the tradeoffs of a logical or physical backup system [Hutchinson et al., 1999]. The main advantage of the physical backups is speed and support of multiple backup strategies while a logical

backup system is portable and is able to restore specific files. With a disk-level solution that knows about file system data structures, we should be able to have the best of both techniques. Once the disk that can process the on-disk data structures, it could restore a specific file. One example of a backup system that uses the disk is S4 [Strunk et al., 2003]. It keeps a backup copy of every modified version of a file for some period of time. The S4 storage system is similar to the Elephant file system in its use of a history pool that versions each modification [Santry et al., 1999], but S4 resides on a server with the Network File System (NFS) rather than a regular file system.

In testing and using our prototype detector, we identified some files that are frequently targeted by our malware samples. Some of the more commonly modified files are in specific locations like the system directories `c:\windows\system\` or `c:\windows\system32\`. These files coincide with commonly targeted files [Szor, 2005a]. This type of information can be used to prolong the time that backups are kept by prioritizing backups according to likelihood of infection or decreasing the number of versions made of a specific file (if not versioning every modification).

These targeted system files provide a good example of how a disk-level recovery system can enhance OS integrity today. For Windows 2000, Microsoft introduced a mechanism called Windows File Protection (WFP) to protect critical OS data from tampering [Russinovich and Solomon, 2001]. For instance, when a protected file is modified, the OS will automatically replace the new version with an old copy of the overwritten file. In June 2000, Collake observed that a simple registry hack disabled the file protection feature [Collake, 2000]. Later, Microsoft attempted to patch the problems with WFP, but more hacks were released to circumvent the patched version [Collake, 2006]. The WFP mechanism is representative of problems associated with host-level defenses. A disk-level integrity enforcer could enforce a similar policy by disallowing any modification that is not authorized.

Another problem in recovery is not trusting the host which could still be compromised via malware that does not use the disk. One promising opportunity is that the disk can

store recovery information in a safe backup area that would be accessible only to the disk processor. Seagate's DriveTrust (see Section 3.2) has a secure partition for which the disk is used to enforce access controls [Seagate, 2006]. If only one snapshot is kept on the secure partition, the user could simply press a "trusted button" that is physically connected to the disk processor. The button provides a limited interface to the disk, but the physical action of pressing the button authenticates the request. (This assumes a legitimate user has physical access, but an attacker does not). Once the trusted button is pressed, the recovery process can begin. The disk will then begin to overwrite data on the host from the recovery area using the single snapshot. When the disk is done with the copy, the user would reboot the machine into a fresh installation.

Another viable option is to use a secure bootup. If the machine can securely boot into a recovery interface via the BIOS, then an authenticated BIOS could initiate a recovery. This increases the trusted computing base (although still small) to include the machine's BIOS plus the disk's firmware.

3.7.2 Response

The disk can provide the best recovery possible, but it is useless if the malware can prevent the recovery process from being initiated. Host-level malware can eavesdrop or deny communication from the disk to alert the user of compromise. If the user does not know of an intrusion, then recovery will not be initiated. This section addresses the problem of the disk responding to an intrusion and alerting the user.

The previous recovery schemes bring to light the problem of communication between the user and the disk in the situation where a malicious adversary may have compromised the host machine. Ideally the disk would also notify the user through a uncompromised communication channel when a virus is detected. However, the possibly corrupted host OS cannot be trusted to do this notification. Anything the disk can do that relies on the host can be intercepted by a malicious adversary. Without being too obtrusive, the disk can modify some data to let the host AV engine know a compromise has been made (of

course, the host AV engine may be compromised by the attacker to disrupt this).

Without changing the architecture of a disk too drastically, we could use a small display (or even LED lights) on the disk drive to notify the user of a matched virus. This assumes (perhaps unrealistically), that the disk drive is somewhere visible to the user.

If we have no way of reliably contacting the user, the disk can simply stop servicing some or all requests. Stopping all requests would frustrate the user, eventually forcing a reboot which would wipe the malware from memory. Similarly, the disk could deny the disk requests that are deemed malicious, and response may not even be necessary (e.g., malware exits after attempting to write its data).

Chapter 4

General Behavior Detection

Our general detector is designed to be general enough to detect most file-infecting behaviors (high detection rate), but also be precise enough to avoid misrecognizing behavior generated by non-malicious processes (low false positive rate). In Section 4.1, we discuss the common behaviors across file-infecting viruses, and then present four behavioral rules (Section 4.2) for detecting file infections that have differing degrees of generality and precision. We analyze the cost of these behavioral rules in Section 4.3; we evaluate the rules by measuring the detection rate against a set of randomly selected viruses (Section 4.4) and estimating the false positive rate using disk trace data collected from multiple users (Section 4.5). Without other changes, several of the rules have unacceptably high false positive rates, as would be expected for such generic rules. Later in Section 4.5, we analyze the sources of false positives generated by legitimate programs with "virus-like" behavior (such as program updates) and present techniques for reducing the false positive rates.

4.1 Common Virus Behaviors

A common characteristic of all file-infecting viruses is that they replicate by infecting executable files. If we can identify the disk-level behavior that is common to file infection, we can use that behavior to detect even previously unknown viruses. This section describes common infection patterns and introduces four generic virus detection rules that

recognize file infections.

Although our rules are designed to capture the disk-level behavior inherent in infecting a file, a malware author could create a virus that would not be detected by our rules. The key observation is that these simple rules appear to be effective in detecting nearly all current file-infecting viruses without needing any specific signatures. Section 6.1 discusses how viruses could be designed to evade these rules and presents ideas for improving our detector to make evasion more difficult.

Figure 4.1: Windows PE File Format

The general behavior of a file-infecting virus is dictated by the structure of a Windows executable, which follows the PE file format depicted in Figure 4.1. The first block is a header that contains information about the structure of the target file. The rest of the executable file is broken into sections (e.g. code, data) marked as Section 0 to Section N in Figure 4.1. The section headers indicate the size and location of each section.

A general characteristic is that a virus must first read the header of an executable file to gather useful information in order to reliably infect files. For example, cavity-infecting viruses use this information to find exploitable slack space between sections. Hence, the first event expected in a file infection is a read from the file header, which is visible to the disk as a read at file offset 0.

In order to modify the executable, the virus must also write to it. Most reliable infection strategies require also modifying the file header. For example, one simple infection strategy is to infect an executable by pre-pending or appending a new section. If a virus infects using one of these methods but does not update the file header, Windows will detect that the executable is not a valid application and will not load it. Consequently, it is necessary to modify the file header if any new sections are added to the file. Another infection technique is to find slack (unused) space at the end of a section in an executable

section and append the virus to an existing section. Even this, however, still requires updating the header. According to the executable file format, the unused portion should be (and is) filled with zeroes [Microsoft, 2006]. If this area is not zeroed out, Windows will not throw an error on the program's execution, but the code will not be loaded into the program's address space [Szor, 2005a]. Thus, the header must be updated to increase the virtual size of an infected section when writing to its slack space.

Another reason a virus may write to the beginning of a file is to insert a file marker. Some viruses modify one or more bytes in a header in order to know if the file has already been infected. W32.Zmist, for example, writes a 'Z' at offset 0x1C in the header [Szor, 2005a]. Some anti-virus programs provide virus authors with an additional explicit motivation to both read and write into the file header. For example, Kaspersky uses weak checksums of 10-12 bytes that are written into the file header to avoid the need to rescan files [Kaspersky Lab, 2005]. A virus can easily change these checksums (as was done by W32.Chiton, which recalculates and updated the file's checksum after infection [Ferrie, 2002a]). This illustrates a nice synergy between our behavioral detector and traditional static detectors: a static detector could check a file property that a virus cannot maintain without generating a recognizable disk-level event that would be observed by the behavioral detector.

4.2 Detection Rules

Because of the nature of the Windows executable format, we expect most infecting viruses to read and write to the file header, and to also write somewhere else in the file. This behavior is captured by the R_oW_oW rule:

read [*name*@offset:0];
write [*name*@offset:0],
write [*name*]
where *name* is an executable file

The first read from block 0 matches the read to learn the file structure. The first write to block 0 matches the update of the executable file's header, and the last write matches the rest of the virus being added. Names in italics (e.g., *name*) are variables that are bound to a specific file name on their first occurrence. Events separated by a comma may occur in any order (such as the first two read events). Our behavioral rules are ordering dependent. Events separated by semi-colons must occur sequentially (e.g., in the R_oW_oW rule, the read event must occur before the write events). Any number of additional events can be ignored (e.g., if there are additional read events after the first write event, this would not prevent the rule from matching).

A somewhat stricter rule includes an additional read. Most viruses will need to both read the header to determine the executable structure, and then read another location in the file to identify code to change to insert a jump to the virus. For example, an entry-point obscuring virus can overwrite a jmp instruction to jump to its code (e.g. [Szor, 1996]). To capture this additional expected read, we define the R_oRW_oW Rule:

read	[*name*@offset:0],
read	[*name*];
write	[*name*@offset:0],
write	[*name*]
	where *name* is an executable file

In addition to these two rules, we also considered two rules that relax the requirements on the infection behavior. Relaxing the requirements makes it more likely the rules will detect virus infections, but also increases the likelihood that the rules will match benign behaviors.

First, we eliminate the requirement for two writes in the R_oW_oW rule. If a virus can fit its data at the beginning of the file, it could infect a file with a single write. The Update-Header Rule captures this:

Our final experimental rule removes the requirement that the virus read the target

read [*name*@offset:0];
write [*name*@offset:0]
where *name* is an executable file

file at all. In theory, a virus could attempt to infect a file without reading the header by guessing where to insert code. We capture this using the Write-Anywhere rule that matches any write to an existing executable file:

write [*name*]
where *name* is an existing executable file

This behavior usually leads to the undesirable behavior of the application crashing. Thus, such viruses that do not read the file header are rare and do not propagate effectively [Szor, 2005a].

4.3 General Behavioral Rules Costs

After inferring the file system events (e.g., file creation), the detector uses general behavioral rules to check the disk requests. The disk uses the semantic map to learn what file is being affected and then will check the file's behavioral rule (if the file is an executable).

To identify the file requires a lookup in the semantic map, which can be implemented efficiently using a hash table. The rules require keeping state on each executable, but the memory cost is low. The W rule is one state, the R_oW_o rule is two states, the R_oW_oW is three states, and the R_oRW_oW is four states. The memory required to store the state can be stored in at most 2 bits.

Agrawal et al. performed a five-year study that collected snapshots of file-system metadata from over 60,000 Windows PCs [Agrawal et al., 2007]. From this study, they were able to find different characteristics of file system use. For example, the machines typically had

an average of 90K files, and the number of executable files on the machine was approximately three percent of the total number of files during each year of the study. If we assume 100000 files on a Windows PC, this gives us approximately 3000 executable files. The memory consumption is about 750 bytes (3000 $\times$ 2 bits) when a general behavioral rule is used for every executable.

4.4 Detection Rates

We used two sets of experiments to evaluate the rules. The first set of experiments used a smaller set of 70 malware samples that we used to further refine our rules (Section 4.4.1). These initial results helped to motivate the final form of the four behavioral rules, and the results indicated that the rules could likely detect file-infecting viruses (and even worms). Because we used the same set of malware to refine the signatures, they were not a fair way to evaulate the effectiveness of the rules. The second set of experiments was performed on a randomly selected set of malware that contained over 300 samples (Section 4.4.2), and this experiment was done after the behavioral rules were fully developed. Some other factors affect our detection results: caching and merging. We discuss the effects of caching and merging in Section 4.4.3.

4.4.1 Initial Experiments

To test the accuracy of the behavioral rules, we randomly selected 70 samples from a large virus repository [Offensive Computing, 2007]. We eliminated those that did not execute (e.g., the virus did not execute for various reasons like needing a specific version of kernel32.dll), those that did not infect any system or goat files upon execution, and those that appeared to be minor variants of others in our sample. To make sure we observed different disk-level behavior, we eliminated variants (variants are likely to be similar). Additionally, we included five viruses we had previously chosen for study (W32.Detnat, W32.Efish, W32.Ganda, W32.Simile, and W32.Tuareg). W32.Aula.a,

W32.Billrus.a, W32.Kriz, W32.Oblion.a, W32.Stupid.b were eliminated from our set, because they did not exhibit any malicious virus behavior in our test executions. W32.Aula.a is a variant of the mass mailing W32.Bagle worm. There is not much information available on W32.Billrus. W32.Kriz is classified as a worm and a dropper, and some classify W32.Oblion.a as a virus or a worm. W32.Stupid.b dropped some malicious files, but it did not proceed to infect files on the disk.

We encountered another problem with two malicious programs that would not run in VMWare: W32.Detnat and W32.Alcaul. There are times where a virtual environment does not behave in the same way as the actual hardware. Although programs typically run unchanged in VMWare, some applications do have difficulty. Among other malicious actions, W32.Detnat stealthily hides from the user as a rootkit, and W32.Alcaul hooks Windows timer events. These low-level actions are likely an incompatibility with VMWare. After running the software on real hardware (outside of our proof-of-concept environment), we found that the malicious disk accesses were present and similar to other malicious disk accesses of other malware we had studied. This left us with twenty-two valid samples with four of them (W32.Ganda, W32.Parite, W32.Detnat, and W32.Sality) still found in the latest wildlist [Wildlist Organization, 2007].

We ran each sample individually while watching its treatment of some planted goat files to generate multiple observable infections. The detector would output when a monitored block would be read or written to and the associated file name. If a virus was detected, the detector would simply output that a virus had infected a specific file. Two of the sample viruses, W32.Detnat and W32.Alcaul, would not run in the Windows host running in VMWare. To see if our rules could capture these viruses we ran them outside of VMWare while we recorded their file system activity with a modified Windows kernel driver. This allowed us to verify that the malware was reading and writing to the beginning of executable files. We could not use our proof-of-concept implementation for these two tests, because the implementation needs VMWare to run. Table 4.1 shows the detection results for each virus sample with the different rules.

Virus	R_oRW_oW	R_oW_oW	Update-Header (R_oW_o)	Write-Anywhere (W)
Alcaul.o, Chiton.b, Detnat, Enerlam.b, Ganda, Harrier, Jetto, Magic.1590, Matrix.750, Maya.4108, NWU, Oroch.5420, Parite.b*, Resur.f, Sality.l*, Savior.1832, Seppuku.2764, Simile, Tuareg (19 viruses)	All infections detected			
Aliser.7825	70%	83%	All infections detected	
Efish*	87%	All infections detected		
Evyl	91%	All infections detected		

Table 4.1: Virus Detection Results. The results indicate the percentage of test infections of the given viruses detected by each rule. All infections of all of the viruses are detected by the R_oW_o and W rules. Viruses marked with a * perform some malicious disk activity before the file infection activity that is detected by the rule.

When testing W32.Eletiamo, it did not infect executable files inside or outside of our testing environment as an actual virus would by definition; it was not included in our test set of viruses. From McAfee's description [McAfee, 2002], we know W32.Eletiamo is an overwriting virus (overwrites its target). To detect overwriting viruses, either we would need to track data flow behavior more deeply (Section 7.3), design general behavior rules to capture viruses that replace existing files, or develop a behavior-specific rule for a known overwriting virus.

In these experiments the R_oW_o and the W rules were able to match all viruses in our test set. The R_oRW_oW and the R_oW_oW rules failed to detect some infections of three of the viruses. Although the majority of infections were matched, these viruses infected some of the goat files without detection. The virus itself always makes multiple reads and writes, but because the OS may merge disk requests the behavior observed by the disk detector does not always exhibit multiple reads and writes. For the Evyl virus, the R_oRW_oW rule

missed four of 47 virus infections due to the reads being merged by the OS into a single read event. Similarly, six infections by Aliser.7825 were missed by the R_oRW_oW rule due to merged reads. For the Efish virus, the R_oW_oW rule missed eight out of 47 infections because of merged writes; the R_oRW_oW rule missed those infections as well as an additional six infections because of merged reads. Requests are merged based on various factors in the OS including other currently pending disk requests. Our infrastructure runs malware in a host that endeavors to create a realistic virtual environment (i.e., VMWare). The merged disk requests are due to the guest OS' disk requests and the disk with which it interacts. We confirmed that the malware would exhibit similar disk traffic by running some malware outside of the tracing infrastructure while recording the disk activity. Although the results due to merged requests are non-deterministic, they appeard to be fairly stable across our repeated experiments.

The R_oW_o and the W rules match all the infections in our sample, but they do let some malicious activity not related to the file infection happen. Four of the viruses — W32.Efish, W32.Ganda, W32.Parite, and W32.Sality — performed other malicious activities prior to beginning their infection sequence. Detection by infection sequence alone is insufficient for stopping all malicious activity of these viruses. The behavior-specific signatures described in Chapter 5 can be used to thwart these viruses before they perform any malicious activity.

4.4.2 Random Sample Experiments

While our results on a smaller sample of viruses were successful, there are those viruses whose disk-level behavior can alter an executable file without conforming to our behavioral rules. Overwriting viruses replace files on disk by overwriting the files with their own data. Others will pick a random part of a file to overwrite with a copy of its malicious code [Szor, 2005a]. Overwriting viruses are easily detected when executing an infected application (all or portions of the original application's code have been overwritten). Although they do not always alter the original executable, overwriting viruses achieve the

same effects as a normal file-infecting virus.

Companion viruses will take advantage of the operating system environment to do harm. These viruses can, for example, find an executable file, and then write a malware program to the same directory with the same file name but with a `.com` extension. The `.com` program will be executed before a program of the same name with a `.exe` extension. Another way of compromise is for the virus to install itself somewhere on the search path for programs. When a program on a user's path is executed, the virus is executed instead of the program. Once executed, the virus can call the original program to remain unnoticed. Although these viruses are less common [Szor, 2005a], their different method of compromise achieves similar results to a regular file-infecting virus without the same disk-level behavior. To remain undetected, a companion virus may make a backup of the original executable file, and then execute that file after the virus executes.

Most of our simple behavioral rules do not catch overwriting or companion viruses (the W rule would catch an overwriting virus). These types of viruses are fundamentally different in their approach to compromising a program, and as a result, are also fundamentally different in their disk-level behavior. The overwriting virus overwrites the original program's data, making its compromise easier to detect when the malware is run. We would like to detect this malware earlier than an infected (replaced) program's execution, and this could be done with other rules or signatures. Companion viruses can be more difficult to detect, but they often make backups of the original executable file. This behavior is very noticeable and could also be detected with other rules and signatures.

After our initial success with our first sample of 70 viruses, we tested over 300 more viruses from the VX Heavens repository (downloaded in March 2007). Their repository includes worms, backdoors, trojans, viruses, virus kits, and spam tools. We selected all malware labeled as 32-bit Windows viruses to potentially include in this sample. We ran one-third of the sample by automating their execution and logging the file system requests in VMWare (only those requests that reach the disk are logged). At the end of a timeout period, we diffed the file system for infected or dropped executable files. If the diff of the

Virus	R_oRW_oW	R_oW_oW	Update-Header (R_oW_o)	Write-Anywhere (W)
Adson.1559, Adson.1703, Apathy.5378 Belial.2609, Bika.1906, Chiton (variants e, q and r), Evyl.e HLL.Fugo	All infections detected			
Blateroz, Chiton, Chiton (variants a, c, j, and p) Cornad, Eclipse.c, Egolet.a, Elkern.b, Emotion.a, Evul.8192.a, Evul.8192.f HLLC.Asive, HLLC.Nan, HLLC.Nosyst HLLC.Ext	No infections detected.			
Evyl.h	98%	All infections detected		
Alma.2714	92%	All infections detected		
Giri.4937.a	91%		All infections detected	
Driller	83%		All infections detected	
Evul.8192.b, Evul.8192.c	56%	78%	All infections detected	
Barum.1536, Champ.5714, Delikon	0%		All infections detected	
Doser.4187	97%			All infections detected

Table 4.2: Virus Detection Results on Larger Sample. We tested a random selection of malware from the VXHeavens (March 2007) repository using VMWare and logged all disk activity. If the malware added, deleted, or changed an executable file, we saved a log of the disk activity to scan.

file system showed different executable files at the end of the malware's execution (either dropped or infected), then we would save the log for scanning with our behavioral rules. In this manner, we could test a wider range of viruses. Our results are in Table 4.2.

From the 300+ viruses tested, there were 39 viruses that altered or created an executable on the file system. Two of these viruses were in our original sample set that we used to develop the rules; we eliminated the previously used viruses from the new sample set. Of the remaining 37, our behavioral rules matched all 20 viruses that attempted to infect executable files. Ten of the infecting viruses were not matched by all the rules. Neither the

R_oW_oW rule or the R_oRW_oW rule matched Barum.1536, Champ.5714, and Delikon, but these viruses only infected two files, one file, and one file respectively.

These results are encouraging. Our rules succesfully matched over half the viruses, and the main problem with the other viruses was their different methods of compromise (e.g., companion viruses). However, these different methods of compromise produce identifiable disk-level behavior.

Companion viruses often rename executables by using the same new extension (e.g., `.bin`). W32.Blateroz, W32.Cornad, W32.Eclipse.c, W32.Egolet.a, and W32.Emotion.a renamed executable files using a different file extension than the original (e.g., Eclipse.c, like many other viruses, used the `.bin` extension for all the newly renamed executables). They then create another executable file using the old executable file's name. For instance, before replacing `explorer.exe`, they would back it up to a file named `explorer.bin`.

HLLC.Ext renamed executable files using a random three-character extension for each executable file, but it also created a file `ext.ddd` in each directory it altered executable files. HLLC.Nan replaces some popular applications (`notepad.exe` and `iexplore.exe`) and then drops a malicious executable into the root directory. HLLC.Nosyst renamed two help files using the extension `.vir`, dropped another executable file in the Windows directory, then edited the registry to point to the dropped file. Other viruses, like W32.Cornad, marked the renamed executable files (the original clean copies) as hidden. Using these different types of disk-level behaviors, all of these viruses could be detected through a disk-level behavioral rule or behavior-specific signature.

Other viruses were undetected due to a combination of reasons. Some variants of the Chiton family of viruses, of which Efish is a part (i.e., Chiton.d; See Section 5.2.1), were not detected. Two of the variants, Chiton.j and Chiton.p, only infect executable programs once every hour [Ferrie, 2003], and the others (Chiton.a, Chiton.c, and Chiton.i) dropped files and edited the registry but infected no files. Chiton.i, Chiton.j, and Chiton.p dropped the same file as Efish to disguise it like explorer.exe (`C:\Windows\expIorer.exe`). Chiton.c and Elkern.b are different in their creation of a dropped DLL (i.e., a Windows dynamically

linked library). Our behavioral rules would match executable infections, but the beginning malicious actions of these variants were altogether different than most of the observed malicious disk traffic.

Another virus, W32.Asive, dropped an executable file and then proceeded to find and infect Microsoft office documents. While the virus does infect application-level files, this type of application malware is outside the scope of this work. Two other viruses, W32.Evul.8192.a and W32.Evul.8192.f, dropped a file just before entering an endless loop. They did not create any further interesting disk-level behavior.

From this larger sample of malware and our initial test set, our rules can not only detect file-infecting viruses, but they can detect unknown file-infecting viruses. While the W rule had the highest detection rate, the R_oRW_oW rule also had a similarly high detection rate. Some viruses, like the overwriting or companion viruses, are not detected with our rules, but rules or signatures could be developed for their specific disk-level behaviors.

4.4.3 Caching and Merging Effects

There are two main caches that can affect a disk detector: the disk buffer and the OS file system cache. Before the OS forwards the request to the disk, the OS will first attempt to service the request if it can be serviced from the file system cache in memory. If the request cannot be serviced by data already in memory, the OS may then merge the disk request with other outstanding requests (for example, two requests for adjacent locations may be merged into a single request), queue the request, or issue the request. Before the request can be issued to the disk, a low-level driver transforms the request to the form of a physical sector address on the disk, the request type (read or write), and the amount of data requested.

Once the disk receives a request, it may use the disk buffer to temporarily buffer the request. The disk buffer is typically 16 and 32 MB, and the disk can use this onboard buffer to increase its transfer rate by delaying writes or by performing readahead. Disk requests by the OS may be serviced from the disk buffer, but the requests must be first scanned by

the detector. On a readahead, more sectors will be fetched and stored in the disk buffer than requested by the OS. Before disk buffer data is transferred to the host, it must be checked by the detector. In our testing the detector recorded data seen at the SCSI driver, the lowest-level point of data transfer in the OS. Thus, reads that were serviced from the disk buffer or by a non-cached disk read (disk buffer cache miss) would all be checked and treated equally.

When writes are issued to the disk, they may be temporarily placed in its disk buffer. Recent advances have increased the complexity in the disk controller to reorder requests for optimal performance [Intel Corporation and Seagate Technology, 2003]. These writes should be checked as any other write to the disk. Our implementation treats all writes the same (all writes seen at the SCSI driver level are treated equally).

The other cache that can potentially affect the disk detector is the OS file system cache. Writes in the file system cache may linger in the cache before being periodically written to the disk. If a write from this cache triggers detection of malware, then the detector will not detect the malware until this write occurs at the disk-level. If an attack is kept entirely within the memory of the machine, then it is outside the scope of our detection, and the disk's integrity is still protected. Our detector operates below the file system cache, and it is able to see all writes to the disk.

Some read requests will not reach the disk because of the file system cache. This cache may service read requests without allowing them to reach the disk. However, if data is in the file system cache, it was read at some point earlier, and the disk can see these earlier reads.

A cache will have an impact on a detector when merging requests. Through merging, the OS may be able to combine outstanding disk requests in the cache into fewer disk requests (e.g., the requests' data are in physical proximity on the disk). The file system cache is especially relevant — a larger file system cache will create opportunities for more merged requests (i.e., a larger cache can hold more requests). For a cached read, an application disk request must already have been serviced by the disk at some earlier time. A

target file will always require at least one initial read, and the file will require at least one final write for infection (thus, the R_oW_o and the W rules will still be effective). For our example rule, we consider the R_oRW_oW rule. Assume the first read, R_o, will happen at some point in time prior to the second read R. The second read, R, may not access the disk if the first read was able to read its data (merged requests). We observed false negatives due to these merging effects in our testing of the R_oRW_oW and R_oW_oW rules discussed earlier in this section.

Another potential possibility of cache interference is when an application reads from the middle of a file without any application first reading from the file's beginning (irregular, but possible). For example, the R in a R_oRW_oW sequence takes place before the R_o. Any subsequent read for the data, R, after matching R_o may have a cache hit. This would not cause a false negative on the R_oRW_oW rule, because these read events can take place in any order (the writes must follow the reads).

Many viruses already make use of caching functionality by writing to files through memory-mapped I/O allowing the OS to lazily flush multiple application writes to the disk. We saw these merged write (and read) requests in our test data for the R_oRW_oW and R_oW_oW rules. This did cause the rules to miss some detections. When creating and using behavior-specific signatures, the signature creator must bear in mind how caching and merging could affect the signature. For example, two reads to the same file in a row may be merged. Care must be taken depending on the file and offsets involved in the reads (e.g., two reads that are far apart in a large file will not be merged but closer reads will be merged).

Our testing infrastructure encountered some caching problems. When first testing the W32.Jetto virus, we had a false negative. On closer inspection, we saw that the malcious disk access to read the infected program was a readahead happening before the detector was started. After the detector was started, any subsequent reads would hit the file system cache. We corrected the problem by starting the detector before the boot sequence.

Causes	R_oRW_oW	R_oW_oW	R_oW_o	W
Updates	0	0	0	73
Software Development	1	2	2	2
System Restores	2	13	33	33
Installations	0	0	0	10
Totals	3	15	35	118

Table 4.3: False Positive Causes

4.5 False Positives

We evaluated the false positive rate for our detector by testing the detection rules against collected traces of disk activity.

4.5.1 Traces

We recorded disk activity using a modified file system filter driver of the Minispy filter driver included in the Microsoft Installable Filesystem Kit [Microsoft, 2007a]. This disk activity came from disk-level traces of 8 different users for a period between one week and up to over three months for each user. Six users were computer science graduate students, and two were more typical computer users (the author's father and fiancee). The tracing tool was scheduled to record much of the disk activity during users' normal working time (each user was asked when they typically used their machine). This tracing tool included portions of data in each request, and this could make the machine slow after periods of high disk activity (i.e., when the machine made disk requests that would queue and could not be quickly written to disk). As a compromise, every half hour during the tracing, the tool would flush its log file and reload its driver.

Table 4.3 summarizes the false positives exhibited by each of the generic virus rules on the traces. The user activities included updating and installing programs, browsing the web, reading and sending email, instant messaging, writing papers, developing software, and listening to audio streams.

Almost 100 million total disk events were recorded. We recorded false positives for four types of activities: compilation, updating programs, installations, and system restores. These false positives are unsurprising, since these activities may involve modifying an executable file.

Table 4.4 reports the false positives for each rule on each user's traces. This came out to 637 total hours of recorded disk activity and one false positive for 212 hours of active computer use for the R_oRW_oW rule, and one false positive for 5 hours of computer use with the W rule. The R_oW_o rule had one false positive for approximately 14 hours hours of computer use, and the W had the highest false positive rate with about 5 hours of computer use. Although any of these rates would be too high (except perhaps the R_oRW_oW rule and possibly the R_oW_oW and R_oW_o rules if User 1 is an anomaly) for realistic deployment, all the false positives resulted from a few specific behaviors.

Four of the eight users experienced no false positives. Two users experienced 65 total false positives from Windows updates, but 63 of these false positives came within an approximate six minute period during a single update. A third user experienced 43 false positives, 33 resulting from system restores matched by the R_oW_o rule (the other 10 were from installations matching the W rule). When these 33 R_oW_o false positives were produced, they were grouped in sizes ranging from two to eight false positives in each group

User	Active Time (hours)	Disk Events (millions)	False Positives			
			R_oRW_oW	R_oW_oW	R_oW_o	W
User 1	52	9.4	2	13	33	43
User 2	25	11.7	1	1	1	4
User 3	12	4.7	0	0	0	0
User 4	311	11.2	0	1	1	8
User 5	110	23.4	0	0	0	0
User 6	44	2.3	0	0	0	0
User 7	61	10.0	0	0	0	0
User 8	22	22.0	0	0	0	63
Total	637	94.7	3	15	35	118

Table 4.4: False Positives Across Users (637 Total User Hours). User 1 and User 3 are non-technical users, and User 4 is the author.

across several different days.

Although these rates are encouraging for such generic rules, even the three false positives observed for the R_oRW_oW rule is too high for any wide scale deployment. We consider each of the causes of false positives and discuss solutions. In order to gain more understanding of potential false positives, we also collected additional records by deliberately performing activities we thought would be likely to cause false positives. These results are not included in our experimental data, but we discuss them in Section 4.5.4.

4.5.2 Program Updates and Installation

Typical Windows users occasionally install software and frequently update programs so it is essential to handle updates without any user disruption. Program updates recognized by the W rule were the single largest source of our false positives in our data. These updates did not match other rules, because they did not read and write to the same executable file. There were 73 false positives caused by updates in our data (of which 65 were for Windows updates). There were also many program updates that did not generate false positives, even for the W rule. Updates that create a new file and then perform a rename (overwriting the older version of the program) would not match any of our detection rules. Note, however, that a virus could currently infect files without detection by using the same strategy, as discussed in Chapter 6.

An installation program is one example of a benign program with virus-like behavior. In our user testing, the SanDisk USB stick caused 10 false positives with the W rule. This USB device is designed to run software from the USB stick [SanDisk, 2007]. Specific details about how the software, U3 [SanDisk, 2007], works with programs running from the USB stick are not available, but some of its features include program updates, program installations, and program downloads [Clark, 2005]. The U3 software has even been used to implement a form of Digital Rights Management [Gruener, 2007]. These false positives were likely program updates by the U3 software.

We did not encounter any false positives due to program installation in our trace data,

but to investigate program installers more thoroughly we also generated traces of five programs using Microsoft's Installer (MSI) [Microsoft, 2007c] and another three programs using the popular Nullsoft Scriptable Install System (NSIS) installer [Nullsoft, 2007]. Table 4.5 summarizes the installers we tried.

NSIS	MSI
7-zip 4.42	FlashStats 2006
FileZilla 2.2.30a	ntp 4.2.0a-1.1370
Miranda 0.6.2-unicode	putty suite 0.58
	TortoiseSVN 1.4.1.7992
	TrueCrypt 4.2

Table 4.5: Installation Files Used for False Positive Testing

For these installation traces, we registered false positives with only the W rule: two of the three NSIS installers and four of the five MSI installers (These false positives are not included in any of our previous tables above in Section 4.5.1). We propose dealing with these activities by changing the way programs are installed to avoid overwriting existing executables. Any overwrites needed to install a program should instead be done using the secure mechanisms described for program updates.

One solution is to change the way that updates are done, ideally by requiring cryptographic signatures. Signing updates has other benefits, regardless of our detector, since unsigned updates are inherently vulnerable [Bellissimo et al., 2006]. To enable secure updates with our disk-level detector, the disk processor must be able to verify the signature on the update without trusting the host. Hence, the public key used to check the signature must be stored in a protected way on the disk and the signature check should be performed by the disk drive processor. The most obvious solution is to embed it in the original executable (at installation) where our infection rules would prevent the public key from being modified. When a program is updated, the signed update would arrive at the disk which would verify the signature, and allow the update without advancing the detection rule. This could be deployed with existing operating systems without any modification, but it

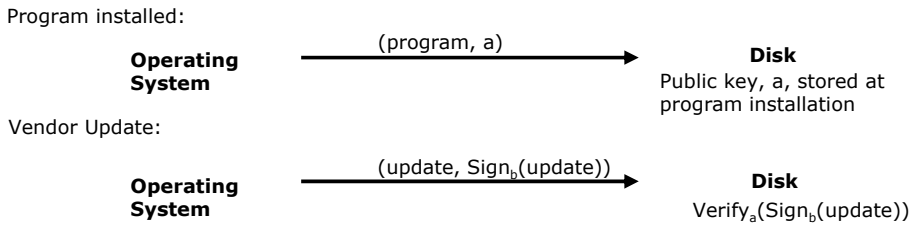


Figure 4.2: Program Installation and Updates.

would require cooperation from program vendors or trusted intermediary proxies. This proposal of using an embedded key is related to security functionality provided in some disk drives today, such as Seagate's DriveTrust [Seagate, 2006].

Figure 4.2 depicts a solution using an embedded public key in the executable. A public key is embedded in the file header of the original program executable, for which the program vendor owns the corresponding private key. Before applying a program update, the disk reads the public key stored in the executable targeted for update and checks that the update is signed with that key.

4.5.3 System Restores

System restores allow a user to revert back to a previous state on the machine [Microsoft, 2001]. This is accomplished through restore points created at important system events (e.g., when an application is installed). One user in our test group had this feature turned on, and it caused the second largest bulk of the observed false positives (33 matched by the R_oW_o and W rule, as well as the only positives not related to software development caused by the R_oRW_oW and R_oRW_oW rules). Windows system restore causes false positives when it is turned on, even if no restore is actually performed.

For legacy systems, the disk can note where the restoration data will be placed at installation. Then, the disk can follow the restoration data through the lifetime of the installation ensuring no writes take place to the restoration data. When a restore occurs, the disk processor can safely determine data integrity by checking the data written matches the saved

restoration data for the corresponding block. In a complete redesign, the OS would not manage system restoration at all. Instead, the disk would create restoration points, saving restoration data in protected blocks that are not visible to the host OS. When a system restore is done, it would be conducted directly by the disk using the protected blocks.

4.5.4 Software Development

We observed a few false positives from Visual Studio 2005 for all of the rules for two users in our user traces. The single false positive that matched the R_oRW_oW rule and the two false positives that matched the R_oW_oW rule were all caused by benchmarking using Visual Studio 2005. We generated some additional traces to better understand false positives caused by software development by performing various activities using Microsoft Visual Studio 2005, LCC-Win32, and Borland's C++ compiler (version 5.5). In these traces, we encountered false positives for all the compilers with the W rule, but did not encounter false positives for any of the other rules.

4.5.5 Other Virus-Like Programs

Various other activities have the potential to exhibit virus-like behavior including antivirus software and Digital Rights Management (DRM) applications. Although we did not encounter any false positives due to AV software in our test traces, some AV software is designed to exhibit virus-like behavior itself. As mentioned in Section 4.1, AV software may write into executables storing a checksum in the file header in order to speed up scanning [Kaspersky Lab, 2005]. Ideally, the disk-level detector would be closely integrated with the host-level scanning software, so these updates could be done in a recognizable way, perhaps even by the disk processor itself. Another solution would be to modify the AV software to use an external database to store checksums making it unnecessary to write into executables [Kaspersky Lab, 2005]. Some DRM schemes attempt to limit executable file use by directly writing how many times the program has been executed into the file (of course, this offers very little protection against a motivated adversary).

4.6 Summary

Our overall approach in dealing with other activities that cause false positives is to whitelist specific disk behaviors that are associated with known, trusted activities, ideally using cryptographic signatures to ensure that virus authors cannot exploit these exceptions. The approach of characterizing a general file-infecting behavior, and using a whitelist to allow certain non-malicious virus-like programs, is a promising alternative to the traditional approach of allowing all programs except for those included in a list of signatures of known malicious programs. The rules vary in their detection and false positive rates. The W rule demonstrated perfect detection in our experiments, but could not be deployed without significant changes in computer systems and cooperation from vendors. The R_oW_oW and R_oRW_oW rules missed a few infections, but exhibited only a few false positives which could be avoided by changes in how system restores are done.

The four rules we presented — *Multi-Read/Write* (R_oRW_oW), *Single-Read/Multi-Write* (R_oW_oW), *Update-Header* (R_oW_o), and *Write-Anywhere* (W) — are all able to detect most instances of the file-infecting behavior that is common to a large class of viruses. The languages the rules recognize are strict subsets of each other with the language of W being the largest, and their relationship to each other ($L(R_oRW_oW) \subset L(R_oW_oW) \subset L(R_oW_o) \subset L(W)$) was evident in our results.

In our experiments, the R_oRW_oW and the R_oW_oW rules detected most of the file-infecting viruses, but missed some infections of a few of the viruses because of merged disk requests. Both the W and the R_oW_o rules detected almost all the sample file-infecting virus infections in our experiments (the R_oW_o rule missed four infections of 133 files infected on Doser.4187). Each of the rules has different desirable properties. The W rule had the most false positives (about two and a half times the next highest rate of the R_oW_o rule), but most of the sources of false positives are well understood and could be mitigated.

The R_oW_o rule had the next highest amount of false positives, but it had fewer negatives than all but the W rule. In contrast, the R_oW_oW rule had slightly higher false pos-

itives, but it did have more false negatives. Although the R_oRW_oW rule registered the lowest false positives, its false negative rate may be too high. The acceptance of our false positive mechanisms countered with an acceptable false negative rate will likely determine which rule would be most useful in practice.

All but the W rule may be able to be used in widespread use (in some cases, the W rule may be acceptable). For all but one user, the R_oRW_oW, R_oW_oW, and R_oW_o rules had the same false positive count. The only user for whom the false postive rates differed was one of the less experienced users who used system restoration. Many of the false positives were due to a process involved in the system restoration utility. This may indicate that different rules may be suitable for different types of users.

Chapter 5

Behavior-Specific Signatures

Malicious behavior extends beyond simple file infectors, so a detector that only detects infections is not sufficient. To detect and stop other malicious programs, we need detectors that recognize other malicious behaviors. Behavior-specific signatures capture the disk-level behavior that is characteristic of a particular malware. Although this approach suffers from the serious drawback (shared with traditional anti-virus software) that it cannot detect previously unknown malware, it has significant advantages over traditional static blacklisting, which can be fooled by replacing code fragments with functionally equivalent but syntactically different code. Automatic obfuscators, such as the W32.Tuareg polymorphic engine also used in W95.Drill, have existed for quite some time [Szor, 2001].

Behavior is much harder to alter without affecting the malicious functionality of a program, and no automatic behavioral obfuscators are known to exist. There are, however, some malicious programs that randomize aspects of their disk-level behavior, such as the W32.Ganda worm described in Section 5.2.2, that randomly generates file names and W32.Simile which randomly skips a found executable file 50% of the time [Perriot et al., 2003].

Behavior-specific signatures characterize specific disk usage for a malware and its variants. We demonstrate our approach with four malware samples (W32.Efish in Section 5.2.1, W32.Ganda in Section 5.2.2, W32.Sality in Section 5.2.3, and W32.Parite in Sec-

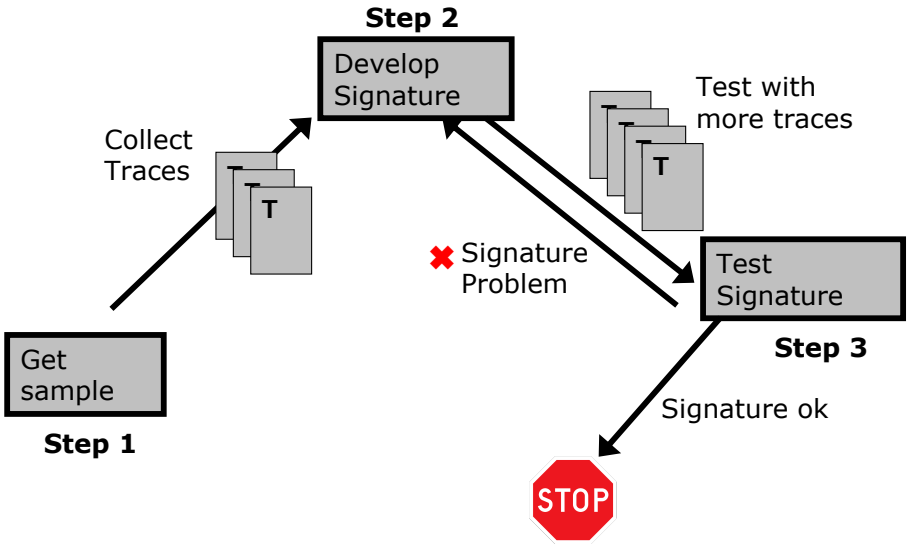


Figure 5.1: Developing a Signature. Step 1 involves getting a sample and getting the sample to execute its malicious payload. In step 2, we use the traces from step 1 to develop a signature based on these traces. After forming an initial signature, we take traces from step 1 and additional traces from other malware variants to test for false negatives. In step 3, we use disk-level activity recorded from benign sources to test for false positives. If the false positive rate is acceptable, the signature is ready. Otherwise, step 2 is revisited by redeveloping the signature (possibly using more traces).

tion 5.2.4). Each of these samples is a virus, with the exception of W32.Ganda which is a worm.

5.1 Developing Signatures

The process of creating specific signatures is based on analyzing records of a target malicious program's repeated execution. Figure 5.1 illustrates the development process. Initially, the sample may be examined using the same custom-built tools to reverse engineer malware that are currently used for static analysis by AV labs [Szor, 2005a]. Once the general characteristics of the malware are understood, a set of testing environments can be

tailored to its behavior (for example, malware that exhibits different behavior in January and February should be run in systems with their clocks set to January and February). The malware should be run over a wide variety of system configurations with many types of goat files to capture as many different types of behavior as possible.

Recent work has automated the process of finding malware timebombs [Crandall et al., 2006] and exploring multiple viral execution paths [Moser et al., 2007a]. Designed for finding malware timebombs, DACODA is a symbolic execution engine that can discover predicates on a system timer [Crandall et al., 2006]. Malware often will only exhibit certain behavior when certain time criteria are met, and DACODA's goal is to find this criteria. Knowing some time-related conditions that must be met to trigger an infection for a malware specimen is important to quickly generate a signature.

Similar to DACODA, Moser et al.'s tool identifies conditional code segments and forces the malware to follow these execution paths, increasing the amount of code that can be dynamically observed at once [Moser et al., 2007a]. Their tool relies on creating snapshots to remember state to later explore other paths. One of their problems is their inability to replay state in some situations (e.g., a program's application of non-linear transformations or insert branches with many paths to force state explosion). Our disk detector would benefit from behavioral analysis tools that can generate different kinds of traces especially for unknown malware. Good tracing tools can reduce the effort required from an analyst to understand a new virus.

There are times where a host-based tracing tool will not be sufficient, such as if the malware purposefully attacks or deceives the tracing infrastructure. For these cases, the malware's executions can be recorded using a disk-level "detector" that commits all of the file requests to disk for later examination. Although a disk-level tracing tool cannot collect all the information visible to a host-level tracing tool, it would be more difficult to detect. Since the recording mechanism is below the operating system and the operating system is unaltered, the malware would have a more difficult problem to detect that it is being studied. This avoids some of the widely exploited pitfalls of anti-emulation techniques

[Szor and Ferrie, 2003], such as the inability to perfectly recreate a natural environment [Garfinkel et al., 2007].

For some instances of malware, we read available writeups about the malware to begin to understand their behavior (of course, an analyst encountering a new malware sample would not have this option). Other malware samples had no writeups. To gain a further understanding of each sample's behavior, we studied our tracing tool's output to find interesting disk-level activity. For one experiment, we believed we had a specific malware instance, and it turned out to be two separate instances of malware blended together (see Section 5.2.3 on Sality and Linkbot).

Once traces have been obtained, they can be examined to find areas of similarities between instances of the malware. These commonalities become the initial signature, which then needs to be tested against different executions of the malware, as well as against different generations of the same malware and its variants. Parts of the signature may be removed or made more general to ensure the signature maintains a high accuracy rate. The desire for generality needs to be balanced with the need for precision to avoid false positives, so the signature should not be overly generalized. It may be preferable to create different signatures for different variants that are effectively a different virus altogether.

After three months of experience with the record collection system and without automation, it was possible for one student to create a good signature for a sophisticated sample of malware with one day's effort. In Section 7.3.1, we consider future work that seeks to automate the generation of behavior-specific signatures.

5.2 Example Signatures

To demonstrate behavior-specific signatures, we selected one older piece of malware and three currently popular and complex pieces of malware: W32.Efish (an older virus), W32.Ganda, W32.Sality.L, and W32.Parite. Many malicious programs today share behaviors with different families of malware. Efish is a file-infecting virus that uses a stand-alone

unencrypted version of itself to find and infect other files. Ganda is a memory resident mass-mailing worm that alters its data on the hard disk. Sality is a keylogging, polymorphic, file-infecting virus. Parite is a file infector that creates a companion file to enable its polymorphism.

For each sample, we initially observed file system events in real time using Sysinternals' Filemon [Russinovich and Cogswell, 2006]. Then, we ran each virus (or worm) in VMWare and recorded its disk activity using an operating system-level driver. From this, we compared the records and generated disk-level signatures and then checked them against variants and generations.

In our behavior-specific signatures, we use some events not included in our general behavioral rules. For instance, the signatures use file creations, comparisons of data inside requests, and locations of data accesses inside files. To recognize create events, the detector needs additional semantic information to infer when a file is created. This could be done by checking the data of each request for a file system create (Section 3.3.3). This is not done in our prototype, but previous research has shown the task of inferring file system semantics is feasible [Sivathanu et al., 2004a].

5.2.1 A Look Back at Our First Attempt

As a first attempt, we began our first detailed examination of malware behavior with W32.Efish.A (Efish), a virus that is representative of many of the viruses we studied. Our analysis of EFish is based on Ferrie and Perriot's analysis [Ferrie and Perriot, 2004], our own runtime observations, and the source code. W32.Efish.A is a multi-threaded, slow polymorphic, parasitic infector. The polymorphism is its primary defense against AV engines, allowing it to evade detection by string scanning. Slow polymorphism makes it particularly tricky; new polymorphic decryptors are generated infrequently, so detection methods need to be thoroughly checked against a wide range of generations of the virus [Ferrie and Perriot, 2004]. We chose to study Efish because several AV companies label Efish as an infecting virus that is also a dropper (a program that writes other ma-

licious programs to the disk). We expected Efish to generate a lot of I/O, and it did not disappoint.

In the following Efish analysis, we highlight some of the challenges and difficulties we encountered generating useful disk-level behavioral traits. Through our study of Efish, we were able to gain a better understanding of typical and atypical disk-level virus activity. This understanding was useful as we moved forward to creating more general behavioral rules (Section 4.1) and more specific disk-level signatures. We were able to develop our signature for Efish in three months while developing our tracing and analysis infrastructure. Once we gained enough experience making these disk-level signatures, we could generate a signature for a previously unstudied malware in about one day.

An Efish infection begins when a host application is executed. We found that the disk events generated by the initial execution of most files (both benign and malicious) is indistinguishable and, importantly, predictable. Execution of a new executable generates a read request at offset 0 (the start of the file). This block data (from read at offset 0) begins with the ASCII characters `ZM` or `MZ` (which Windows uses to distinguish executable files). Following the initial read are accesses to the files `apphelp.dll` and `sysmain.sdb`, which are visible as close requests with the associated filename. The close events do not identify a virus, since many applications access and close these files. This highlights the problem of finding file system events that can be used to characterize viruses to understand and then later use to detect a virus at the disk. These close events are useful, however, for recognizing application startup and ultimately Efish's disk-level behavior, but we do not use them as part of our disk-level detector.

Efish takes control of the flow of execution by replacing the first 32 bytes of a function prologue in the host file with a decryptor. Many polymorphic viruses using a decryptor access the Windows API via the registry to initialize a pseudo-random number generator. We did not use the registry access behavior in our rules or signatures. We may be able to use these types of registry accesses in other signatures, but we leave that for future work (Section 7.3).

The Efish decryptor decrypts the body onto the stack and then executes it. Once Efish has begun executing, its behavior can be divided into two distinct phases: (1) dropping a new file into the `C:\Windows\` directory; and (2) searching the hard drive for files to infect and infecting them. The first phase is typical of viruses that download or carry other malware (such as keyloggers) or make use of a companion file. The second phase is the file-infecting behavior characteristic of a virus.

The file that Efish drops is its own unencrypted virus body. It names the file "`ExpIorer.exe`" and stores it in the `C:\Windows\` directory. This simple naming trick is highly effective because the Windows Task Manager uses a sans serif font to display processes, making "`ExpIorer.exe`" look like "Explorer.exe". The work of the host application is done once `ExpIorer.exe` has been created and executed. An executable by the name of `ExpIorer.exe` is not necessarily indicative of a virus, but it is very unusual. Disk-level modifications like this may be able to be incorporated into some type of heuristic analyzer [Symantec, 1997a].

The writing and executing of `ExpIorer.exe` triggers the following three requests (see Section 4.1 for the syntax of rules) listed in Figure 5.2. The read corresponds to the file being executed and is similar to the R_oW_o rule.

```
create   [ExpIorer.exe];
read     [ExpIorer.exe@offset:0]
write    [ExpIorer.exe@offset:0];
```

Figure 5.2: Efish Dropping Rule

Next, `ExpIorer.exe` begins aggressively searching for and infecting files on the hard drive and through network shares. It has three threads dedicated to this task: the first looks through fixed drives; the second looks for network shares every ten minutes; and the third targets random IP addresses. We focus on detecting the behavior of the first thread because it is the thread that generates the most interesting disk traffic. We briefly studied using Efish's thread-level characteristics and disk behavior from multiple threads,

because the disk-level behavior from multiple threads was abnormal (e.g., one thread kept failing in its attempt to connect to multiple random IP addresses). Deriving patterns from this data, however, was more difficult than using just a single thread of execution, and we picked the thread that infected files on the local disk.

Efish begins the individual file infection sequence by checking to see if the Windows File Protection (WFP) feature is protecting the file (see Section 3.7.1). If the file is unprotected, Efish checks several other attributes, including its character set and file type. The body of the virus is placed in the last section of the file. These steps generate disk activity that matches the R_oRW_oW rule.

Many viruses attempt to infect files that are protected by WFP. Some are smarter and first try to detect if a file is protected by WFP like Efish. Detecting failed writes of WFP-protected files would be useful information to include as a heuristic, but a program trying to replace or modify a protected file is not necessarily a virus. However, it may be a useful indicator that a program is likely to be malicious, since most users will not want a third party replacing or modifying a protected system file. Although not included in our signatures or rules, a disk could enforce a policy of denying all writes to system-protected files. These types of disk-level policies have been studied before by others [Pennington et al., 2003; Griffin et al., 2003] (see Section 2.4).

We checked all of these Efish events against Efish generations 1, 50, 100, 150, 200, and 250 and found that the execution, file dropping, and infection behaviors were always detected correctly by our execution rule, the Efish Dropping Rule, and the R_oRW_oW infection rule respectively.

We end this discussion with an amusing anecdote we experienced. Shortly after our initial work with Efish, someone contacted us after finding our work through an online search engine to thank us for our detailed information available online. We were the first to attribute certain disk-level behavior to malware (e.g., certain errors were being generated in our traces), and this helped them figure out the specific problem of their machine —

it was infected with three viruses! Furthermore, the file system behavior the user found beneficial did not make it into any of our rules or signatures showing that there is semantic information that we should be careful to note even if we do not use it in the signature. This type of disk activity could help us understand malware and modify existing signatures faster if we have extra information like this available.

5.2.2 W32.Ganda

Ganda, which has its own SMTP engine, sends mass quantities of e-mails on politically sensitive topics in English and Swedish that entice the recipient to open an attached screensaver [Hypponen et al., 2003]. We developed a signature for Ganda that captures the execution of the screensaver and the dropping of two files (with identical content) in the `c:\Windows\` directory. To do this, the virus reads itself at four specific offsets and then creates two new executable files (one named `scandisk.exe`, the other randomly named). The signature in Figure 5.3 was successfully tested against two wild Ganda samples which contained slightly different code but displayed identical behavior.

read*4 [`ganda.scr`@offsets: 78, 3, 27, 59];
create [`scandisk.exe`];
read [`ganda.scr`@offset:0];
write [`scandisk.exe`@offset:0];
create [*random.exe*@offset:0];
write [*random.exe*@offset:0]
where *random.exe* and `scandisk.exe` have the same contents and reside in `C:\Windows\`

Figure 5.3: Ganda Signature

Ganda begins searching for executable files during the last few actions described above, but does not begin writing to files until `random.exe` is written. Ganda puts a twist on file infections: instead of infecting files with a copy of itself, it leaves code that points to one of the dropped files in the `c:\Windows` directory. The insertion of this code still requires the same actions as a regular file infection, so our general file infection rules

still detect Ganda. After infecting some files in this fashion, the mass mailing occurs. Since the characteristic behavior described here happens before the file infection and mass mailing, these other malicious components could be prevented with use of a Ganda-specific signature.

Upon execution, Ganda decrypts its encrypted sections and drops exact copies of itself into the `c:\Windows\` directory as `scandisk.exe` and a second, randomly-named executable file. The execution disk-level sequence depicts the execution of the original `.scr` file (`.scr` is a file extension for a Windows screensaver file).

The offsets in the signature were always constant in traces, but these offsets are file system information not directly visible to the disk since by the time the requests reach the disk they may be merged. A disk cannot expect to see the same offsets that our tracing tool does. These offsets are at the file system level, and are offsets into a specific file. By the time a disk request reaches the disk, much of the semantic information has been lost. Many times, an OS will merge disk requests that are physically contiguous even if the requests came out of order. Part of this merging depends on scheduling (do not want to delay requests overly long for the chance to merge other requests). When the detector checks the block addresses of disk accesses, it will match any read that includes these offsets.

A disk-level detector may still gain some benefit in using the file offsets. For example, if a disk sees a read for block 0 through block 64 of `ganda.scr`, then that matches three-fourths of the offset in this signature, but it does not match block 78. A generalized version of the Ganda signature (without requiring specific file offsets) triggered no false positives on the user data. The main reason it is not likely to exhibit false positives is that both created files' data must match and the created files must be in `C:\Windows\` to recognize Ganda.

An interesting attribute of this signature is our inability to properly describe all of the signature's attributes. In particular, Ganda drops two executable files that are exactly the same, `scandisk.exe` and the randomly named file, *random*`.exe`. Many AV engines often have to update their engine along with their signatures to match particular viruses.

If we can generalize types of events that force an update to the disk detector's engine, then future updates with similar events may already be handled. For example, in Ganda's case, we could augment our signature descriptions by stipulating that certain parts of a read or write should be remembered (similar to the Python regular expression module's support for backreferences). Using a backreference, we could then stipulate that the randomly created file was the same as the `scandisk.exe` file. In this case, we changed the detector instead of changing the signature.

We tested this signature without offsets (Figure 5.3) against three separate instances of Ganda (all were obtained from user-uploaded samples [Offensive Computing, 2007]). Two were identical, and one was a variant with code and file size changes but the same behavior. Human-tailored alterations seem to have been made to one of the files in an attempt to evade string scanning. Our signature successfully detected all three Ganda samples with no false positives.

5.2.3 W32.Sality.L

W32.Sality.L is a keylogging, polymorphic, file-infecting virus that also has worm behavior (e.g., including having its own SMTP engine). Variants of the widely known W32.Bagle worm have downloaded Sality for use in a machine's compromise [ca, 2006]. We acquired

create	[\system32\wmimgr32.dll];
write	[\system32\wmimgr32.dll@offset:*any*];
read*4	[\system32\wmimgr32.dll@offset:*any*];
read	[Sality.exe@offset:0];
read*2	[Sality.exe@offset:64, 128];
create	[\system32\lssas.exe];
write	[\system32\lssas.exe@offset:*any*];
write	[Sality.exe@offset:0];
write	[\system32\kfxijtft.bat@offset:0];
read	[Sality.exe@offset:0];
write	[\system32\system.ini@offset:0];

Figure 5.4: Observed Disk-level Actions of Sality.L Instance.

Sality.L from `vx.netlux.org` [VXHeavens, 2006], an online virus repository. We ran the virus and observed the actions shown in Figure 5.4. Subsequent runs yielded the same general form, with `lssas.exe` replaced by other names like `firewall.exe` and `explorer.exe`. The batch file contained a script that deletes itself and the original file, and its name was a randomly-generated string.

This was a confusing result, since this behavior did not match the Symantec Security Report for the Sality family (Symantec employees write these reports as part of a writeup on the information of commonly known malware) [Florio, 2003]. According to the Symantec report, Sality drops a malicious DLL in the `c:\windows\system` folder, calls it, and then injects the DLL into a running process. It then writes its configuration information to `system.ini`.

The Sality.L signature that we developed from both traces and the writeups (verified with other samples), shown in Figure 5.5, accounts for part, but not all, of the behavior we had been viewing.

```
  write   [drop.dll@offset:0];
read*4    [drop.dll];
   read   [\system32\system.ini@offset:0];
  write   [\system32\system.ini@offset:0|data:"TFTempCache" or "MCIDRV_VER"];
```

Figure 5.5: Sality.L Signature

The name *drop* is a variable name for `.dll` files, and drop is always left in the `system32` or `Temp` folder. A significant amount of the observed activity is omitted from this signature and Symantec's description. To look for the source of the discrepancy, we uploaded the sample to `virusscan.jotti.org` [Bosveld, 2007], a site that checks uploaded files against fifteen virus scanners and returns the results. Ten of the fifteen virus scanners agreed that the file was infected with Sality; the other five identified the file as also being infected by W32.Linkbot.M, a backdoor-opening worm. The Symantec Security Response for Linkbot.M [Shinotsuka, 2005] confirmed that the worm's expected behav-

ior matches the parts of the signature not attributed to Sality. We knew previously that variants of the W32.Beagle worm download Sality.L [ca, 2006], but finding Linkbot and Sality functionally packaged together was surprising. Being able to identify the malicious behavior without expecting it was an encouraging mistake.

We arrived at our discovery through a manual process. A disk-level detector would recognize this malware either by its file infection behavior or through a behavior-specific disk-level signature. Building on this discovery of the packaged malware, this analysis could be used as a malware classifier by using the observed disk-level behavior to classify an arbitrary program with some indicated level of confidence. This can help someone making a signature understand the program better. For instance, if the program is Sality with 50% certainty and Linkbot with 50% certainty, it is likely the program contains both.

The original Sality signature was designed using only the information from our Sality.L sample and had a specific file name for the dropped DLL file (`wcmlogon.dll`); once we replaced that with the variable name drop (as is shown in Figure 5.5), the signature also detected Sality.M, Sality.O, and Sality.Q. We were unable to obtain functional copies of Sality.N, Sality.P or other later variants for testing. Earlier variants exhibited slightly different behavior such as dropping randomly-named executables. The signature we developed for Sality.L did not detect Sality.K or other earlier variants we tried.

The signature was tested for false positives against all our trace data and none were found. The `system.ini` file is not normally modified, and it is even more rare for the modification to have the same data used in our signature. Additionally, our general infection signatures catch Sality's file infection sequence (although not all of the Sality variants include a file infector component) even though our Sality-specific signature would capture the virus before file infections began.

5.2.4 W32.Parite

Parite is similar to Efish (Section 5.2) in that it is a memory-resident polymorphic file infector that can spread across network shares and mapped drives. The description of

Parite is based on Landesman's [Lan05], which was also confirmed by our independent behavioral observations. Parite.A self-decrypts and drops a sample of Parite.B, in the form of a Borland-compiled executable file, into the `C:\Windows\Temp\` directory as a `.tmp` file. The `.tmp` file is named with three random letters followed by a number. Once the file is created, Parite.A executes Parite.B and exits. Parite.B then begins infecting files and writes a reference to itself into the registry, using the string "PINF," which is reflected in the `ntuser.dat.LOG` file that backs up the part of the registry related to HKEY_CURRENT_USER [Microsoft, 2007d]. Figure 5.6 shows a possible signature for Parite.

```
create    [name.tmp];
write     [name.tmp@offset:0];
write*3   [name.tmp];
read*7    [name.tmp@offsets:336, 274, 2, 66, 130, 194, 258];
write     [ntuser.dat.log|data:"PINF"];
```

Figure 5.6: Parite Signature

The number of writes and reads remain constant across generations since Parite's polymorphism only changes the encryptor and encryption of the main virus body between infections. The Parite.B sample is unencrypted when it is written and read. The structure of the unencrypted file is therefore the same leading to a regular pattern of file offset accesses in the reads and writes. Although these same disk-level patterns are repeatedly seen across different traces, the first event, read [*file*.exe@offset:0], may not be a good candidate for the first part of the signature. Every time a read at offset 0 of an executable is issued to the disk, a new state machine will be created for matching Parite. Instead, this event can be dropped from the signature.

Another characteristic behavior is Parite's registry updating behavior. The last part of its rule stipulates a write with data containing the string "PINF". This write is part of a registry update that points back to the temp file that was created earlier. Registry

updates that point back to infected or dropped files are common. If we understand the data in a registry update, we can use this in a disk-level signature. Similar to our semantic gap solutions (Section 3.3), this would require reverse engineering registry updates or cooperation with Microsoft.

Parite's signature describes three major events: the execution of an executable file (the virus), the creation and use of a `.tmp` file that contains the decrypted version of the virus, and the infection of a registry file. The `.tmp` section of the signature contains the most specific information; it is identifiable as a creation, a write to the header of the file followed by three writes to any part of the file, and seven reads at known offsets.

We checked the Parite signature (without file offsets) against seven generations of the virus, and all were detected correctly. The signature was also tested (again without file offsets) for false positives against all the collected traces described in Section 5.4. Since the signature is so specific, we were not surprised it generated no false positive reports. For instance, the data string "PINF" would not be expected to appear frequently in other traces. However, this may be an easy attribute to change in the malware. If we add the registry update to the signature (assumes an understanding of the disk-level registry update), then this could aid in detection. The Parite signature can be used to detect Parite prior to its file infection sequence; even without this early-warning shield, however, the file infection rules described in Section 4.1 catch Parite's later file infection behavior.

5.3 Costs

Our behavior-specific signatures specify a sequence of disk-level events. Each event is associated with specified files. For example, a read from `system.ini` at offset 64 followed by the creation of a `.bat` file (any file that ends in `.bat`) can be part of a behavior-specific signature.

In our behavior-specified signatures, filenames can be dynamically bound. For example, the event [read *name*.tmp@offset:64] will be matched when a read is issued for any file

with a `.tmp` file extension at offset 64. After this event is matched, any reference to the variable *name*.tmp later in the signature is bound to the file name in the matched event.

For each signature, there is a corresponding state machine, which can include transitions with variables. For each state machine, there is a set of possible states, each of which includes a variable binding table. We refer to the machine's state and the table of bound variables as its *configuration*. When an event is matched, the state machine may enter a new possible state, with a new associated variable binding table. To compute the overhead of these signatures, we first analyze the number of possible state machine configurations for a single signature (storage cost), then we determine how many state machines will need to be transitioned given an event or disk request (computational cost).

Since we do not have a sufficiently large collection of behavior-specific signatures to estimate the expected cost in practice, we instead compute an upper bound on the cost as a function of the number of signatures and the number of variables in those signatures. For a FSM with no variables, the configuration is the FSM state (no binding information). A FSM with no variables could be in any state at any point in time. For instance, when a transition occurs from the start state to the next state, the same FSM could match another request in its initial start state. The total number of possible configurations from a single FSM with no variables is the number of states, s, in the FSM.

Signatures with variables will have a larger number of possible configurations. The main challenge is that each variable requires maintaining a new configuration when it is bound, because a new disk request could be issued that would match the same variable with a different value. In our example, a read of foo.tmp at offset 64 and a read of bar.tmp at offset 64 would both match the event [read *file*.tmp@offset:64]. The number of configurations that could potentially be spawned by this event is the number of values (or file names) to which the variable, *file*.tmp, could bind. The upper bound on the possible binding values is the number of possible files that have the extension `.tmp` and are 64 or more blocks in size.

We now introduce some notation with an example signature (See Figure 5.7). We label

```
 read   [foo.tmp@offset:64];
write   [name.tmp@offset:0];     // name.tmp is variable v1
write   [done.dat@offset128];
 read   [name2.txt@offset:32];   // name2.txt is variable v2
write   [name2.txt@offset:0];    // name2.txt already labeled v2
write   system.ini@offset:0;
```

Figure 5.7: Example Behavior-Specific Signature. The variables in this signature are *name*.tmp and *name2*.txt. Each variable's first occurrence receives a label in the signature binding each variable in order.

the first occurrence of the ith variable from left-to-right, top-to-bottom, v_i. In this example, *name*.tmp is labeled as v_1, and the first *name*.txt (in the read state) is v_2. The write to *name*.txt at offset 0 is not labeled (v_2, the previous *name*.txt, will determine the value to which it binds).

These variables will determine how many configurations are possible at any given time. The set, V_i, is the set of all possible values that could be bound to the ith variable v_i; $|V_i|$ is the size of the set V_i. The total number of variables for a behavior-specific signature is v. Given these parameters, an upper bound on the number of configurations for a given signature is:

$$\text{number of states} \times \prod_{i=1}^{v} |V_i|$$

For a system of n signatures, each with a maximum of s states and v variables with up to w possible values for each variable, the upper bound on the total number of configurations is nsw^v.

Although we did not use other types of variables in the few signatures we developed, they would likely come in useful. This includes variables for data to be matched, different file sizes, varying offsets, etc. We risk state explosion with the ability to match a variable in a specific type (e.g., a file name), but this avoids the problem of an attacker thwarting a signature with simple obfuscations.

We expect the number of active configurations and the number of states to transition per event to be much lower than our upper bound. The number of actual variable bindings is limited by the disk requests; although the disk may contain a large number of files, only those files which are actually requested are ever instantiated in binding tables.

There are other ways to decrease the cost of using variables. Typically, there will be 16 or less states and one to two variables in a signature. One easy way to avoid problems with state explosion is avoiding variables that will obviously lead to a large number of state machine configurations. For example, a read from *file*.exe at offset 0 will generate too many configurations for the state machine. New signatures should be designed with these constraints in mind. Another possibility is to eliminate state machine configurations that are the same, but this does require making a comparison between two of the same type of FSMs to see if their current states are the same.

State Transitions. Given a signature, a copy of a specification of that signature will be stored in memory to know which state transition to take on a match of a state within that signature. The main problem is managing the potential state explosion. Upon receiving a disk request, it could match more than one state in different state machines. In our implementation, we simply stored all the states and performed linear searches through the potentially matching signatures. This of course will not scale, and we now present an alternative design.

To quickly match a disk event, we can use tables to lookup states (e.g., a hash table). With a potential number of exponential states, we cannot store all possible states and compare against a given disk request. Instead, we can make use of the structure of the states to implement a more succinct representation. Our signatures used three events, multiple files (file ids), seven file extensions, and multiple file offsets. These values can be used to index states.

5.3.1 Summary

The malware-specific signatures for Efish, Ganda, Sality, and Parite are straightforward representations of the disk-level effects of their characteristic behavior. From Efish we gained insight into creating other disk-level signatures. The signatures for Ganda and Sality remained effective against variants that were not used in developing the rules, and Parite's signature worked for morphed generations of the virus. Efish, Ganda, Sality, and Parite triggered no false positives on our collected user data. These are encouraging results. Parite is a highly used virus in the wild today [Braverman et al., 2006] even though it was developed in 2001.

Much of our work was a manually intensive process of collecting a sample and its variants, generating traces from that sample, studying the traces for interesting disk-level behavior, creating a signature from the disk-level behavior, and then testing the signature. If the signature did not match the sample's traces or its variant traces (i.e., a false negative), then we had to repeat this whole process starting with more analysis of the traced disk-level activity. If the signature matched too much disk-level activity as belonging to the malware sample or to one of its variants (i.e., a false positive), then we repeated the steps to generate another signature. After gaining a better understanding of malware disk-level behavior, we could perform many of these steps more quickly, and we discuss the possibility of automating this manual process in Section 7.3.1.

Chapter 6

Security Analysis

The previous two chapters demonstrated that our approach can effectively detect current viruses; we now consider potential attacks targeted against our disk-level defense. We examine viruses engineered to evade our detection mechanisms in Section 6.1. We consider circumvention in Section 6.2, and resource exhaustion in Section 6.3. Section 6.4 examines how the mechanisms we described for handling false positives in Section 4.5 also provide opportunities for malware authors.

6.1 Behavior Obfuscators

Evasive malware can be purposefully designed to access data in a way that does not match a general behavioral rule or a behavior-specific signature. We have constructed our general behavioral rules based on certain properties of virus behavior. A virus must read a program to know how and where to insert itself. The modified program must be written to disk to become infected. A virus could infect a file by obfuscating its disk-level activity to evade our rules. Similarly, our behavior-specific signatures match other specific malicious disk activity. If a virus can obfuscate its malicious activity, it can evade a behavior-specific signature.

A myriad of possible obfuscation scenarios could be envisioned, but the key advantage of disk-level malware detection is always being able to see requests to and from the disk.

We present one example that has been used in past viruses, and another example that could be used against us in the future.

ZMist is an example of a virus that creates a new data file and then copies it over an existing executable [wizard, 2002]. This activity does not match our generic rules, but it does require specific disk activity (overwriting). An additional rule could be developed to identify it, but an overly general rule that detects copying over executables would have a high false positive rate due to matching program updates. A more sophisticated rule might recognize both the creation and copy events. More deceptive and complex copying and moving strategies can be conceived. For example, a virus could read in a target executable and write out information to a temporary non-executable file. After the next reboot, it could read the temporary file for information about the file that was initially read. Then, the virus could simply replace the targeted executable with an infected copy. If such viruses become common, more sophisticated detection rules could be developed that track information flow through the disk across copies, moves, and renames (see Section 7.3.2).

Malware could be constructed to perform malicious disk actions in a randomized way in order to evade behavior-specific signatures. Similar to current code-morphing viruses, behavior-morphing viruses could change their disk behavior as they propagate to make it difficult to find effective virus-specific signatures. Some limited forms of behavior-morphing exist. For instance, some malware uses filenames that are picked randomly including Ganda, Sality, and Parite. This is more difficult to match compared to a static unchanging filename. To deal with this type of behavior, we designed our signatures to use variable file names and match these obfuscated file names. We plan to investigate these issues in future work (see Section 7.3), but note that virus authors will only be motivated to create disk-level behavior-morphing viruses if behavioral defenses such as this one become widespread.

Similarly, malware could attempt to modify disk accesses to have completely different behavior. For instance, whenever a disk access happens, the malware could alter the request to read as much data from the disk as possible causing the disk to match a state in

any rule that is waiting for a read from a specific block. This could cause some behavior-specific signatures to match a state monitoring reads from specific files, leading to false positives. Although this behavior would likely cause a large slowdown, it may not be noticed by typical users.[1]

Radically changing disk accesses could likely be detected by a disk-level detector or even a host-level defense. This is equivalent to asking for all the executable files on the disk whenever needing to read certain bytes from a particular file. This type of slowdown would make the machine too inefficient, and it may cause a match on a rule by forcing states to be matched for the extra disk activity that would not have normally been matched in normal disk usage. If malware is responsible for the alarm, then the detector has done its job.

Our signatures may be circumvented when a specific monitored behavior is not performed while the intended action of that behavior is accomplished. For example, a rule that monitors disk-level activity related to a specific `.bat` file (e.g., Sality) could be evaded if a `.com` file is used to accomplish the same action as that `.bat` file. These types of behaviors may be used to avoid specific signatures, but tailored signatures could be extended to handle these types of behavior obfuscations.

6.2 Circumvention

Because the detector is running on the disk processor, which sees all disk requests, circumvention in the form of not seeing disk requests is not possible: there is no way for malware to access the disk contents without going through the disk processor.

One circumvention strategy is to hide file manipulations from the disk. Malware can map files into memory before modifying them. This makes identifying malicious disk modifications more difficult. Some observable malicious disk activity could later be hid-

[1] The author has seen several real-world instances of malware that has been left running on machines for months or longer (users often notice their machine is slower from the malware's presence but are unsure of the cause and how to troubleshoot the problem).

den by malware. Hidden, in this sense, means the targeted write is part of a larger write that occurs later in the memory-mapped write. While this is a possibility, most malicious disk-level modifications are done by the system process writing the memory-mapped files to the disk, because AV kernel modules attempt to intercept all data going to and from the disk. Our general behavioral rules and behavior-specific signatures can capture even memory-mapped disk accesses as demonstrated in our experiments (much of the malware we studied used memory-mapped disk accesses). If a write to the disk crosses file boundaries, the disk will use the semantic map to associate the disk accesses to multiple files. For those writes that are memory-mapped, they eventually must reach the disk, and we have a chance to monitor and respond to them before they access the disk.

An attacker could also try to compromise the disk detector program itself, which is stored in the disk's firmware. Malware that corrupts firmware of other hardware devices has been suggested in the past [King and Chen, 2005; Heasman, 2007]. A disk's firmware can be upgraded, but doing this is a rare and restricted operation where the host processor updates the firmware through special commands while the disk processor runs off code copied to its RAM [Wells et al., 1999].

An attacker can modify or stop a firmware update using this type of firmware update design. For instance, the attacker could intercept sectors being written to the disk during a firmware update and change those sectors being written. Actual modification of an issued update would take place after the host checks the update's signature. Because of the potential attacks, we must change this update process. A disk providing disk-level malware detection would need to update its firmware differently than traditional designs.

We can simply change this update procedure to have the disk processor perform the update itself and check signatures on proposed updates that are signed with a secretly stored vendor key. The host is used as a way to receive the update from the vendor and nothing more.

6.3 Resource Exhaustion

Slow infections can be very difficult to detect. If the detector maintains state that expires over time, then the malware can wait out the expiration period before executing the next event. All of the viruses we tested rapidly infected the system, but viruses could be designed to infect slowly.

Current AV systems suffer from a type of exhaustion attack. When the AV detector must emulate programs to capture their behavior, the performance overhead of emulation is high (some malware takes on the order of millions of operations to emulate [Szor, 2005b; Nachenburg, 1997]). Malware can be constructed to force the emulator to first emulate many garbage instructions before getting to a part of the program that carries out the actual compromise [Drori et al., 2005]. These techniques are well studied and have been used for a long time. For example, some may try making the emulator enter into an infinite loop that would not be executed by the virus [Lord Julus, 1998]. Since we are not doing emulation, those attacks are not a problem for the disk-level detector.

Other types of resource exhaustion attacks, however, may be possible. A resource exhaustion attack on a disk-level detector will either try to overwhelm the disk processor or place a strain on its RAM usage. To exercise the disk processor and possibly its RAM usage, malware can submit a flood of disk requests. For instance, an attacker could submit disk requests that force the detector to create a large number of states from a variable in a behavior-specific signature. We can do two things to mitigate this type of attack: design the detector to recognize these types of attacks (while keeping the false positive rate low) and carefully design the behavior-specific signatures in how variables are used (while keeping the false negative rate low). One hurdle to this attack is that the user may notice heavy disk usage. A possible solution could have the disk buffer incoming requests until necessary.

6.4 Exploiting False Positive Mechanisms

Other approaches could attempt to exploit our false positive solutions. We now examine how malware could change its behavior to exploit these false positive mechanisms.

Legacy Program Updates. One type of attack would be for malware to attempt to emulate a whitelisted behavior for a program update. If the virus could match this trusted behavior, then it could infect the machine. The whitelisted behaviors should be precisely defined to limit the size and content of written data in ways that make it difficult to construct a malicious program that exploits the whitelisted behavior.

A sample of malware that has both trojan and virus characteristics could make benign software look malicious. A trojan-like program could wait until a benign program is updated. At that time, the malicious program could infect other programs as a virus would. Even if the newly downloaded or installed program is legitimate (e.g., through whitelisted behavior), the user may not be able to distinguish the legitimate program's behavior from the malicious program's behavior (the malicious actions are only done at the time of the legitimate program's update). With a signed update, the user would be faced with a verified update of benign software versus a malware alert. One possible defense is detecting the trojan before it executes its malicious actions, and we discuss a disk-level detector for other types of malware in Section 7.3.3. Anything we do to temporarily suspend the detector (e.g., could be done when updating software), opens a possibility for compromise. If malware were to spoof legitimate software and ask that the detector be temporarily suspended, then the machine could be compromised during this time.

Another danger is a malicious program could be constructed to repeatedly trigger virus alerts until a user grows frustrated and disables the detector. This highlights the need for well-designed response and recovery mechanisms that do not frustrate users. If the false positive rate is low, then the malicious program's attempts at repeated virus alerts may prompt the user to try and remove the malware.

System Restoration. System restoration involves restoring the OS and its data to an

earlier (and hopefully clean) state. The ideal solution for system restoration is for the disk to completely handle this operation. This solution is faster in that the disk knows the physical layout of the data, and it can handle all the semantics involved in backups. When a restoration is requested, the disk services the request.

If we maintain the design of the OS managing system restoration points, then we have options at installation time to configure system restoration (assuming the OS is not compromised during installation). The disk recognizes that it is being installed by the creation of a new MFT. If the entire MFT is new, then the entire file system is now being installed, and this should only happen at installation time. The disk can set aside an area on the disk that will be protected from all writes. At the same time, the user could configure a regular period where the disk creates a backup point. These periodic backups can only be scheduled during installation, or at other times via a trusted BIOS and a secure boot option.

6.5 Summary

In this security analysis, we have shown many possible attack avenues on our detector. Malcode authors will only be motivated to develop these attacks if our defense is deployed. Complicated viruses today exhibit identifiable disk-level behavior that can be easily stopped even while other host-level detection methods seem to fail.

Our detector's main advantages over previous approaches are its default approach to denying certain disk-level activity and its ability to offer complete mediation. Rootkits, like Mailbot, running at the same privilege level of the OS can wreak havoc on OS integrity. Although traditional host-level detectors attempt to completely mediate access to important resources like the disk, they are doomed to fail.

Chapter 7

Conclusion

We conclude by summarizing our contributions and by describing how those contributions improve malware detection (Section 7.1). Section 7.2 covers general limitations of the disk-level behavioral approach. In Section 7.3, we discuss interesting avenues of future research including automating disk-level signature generation, incorporating data flow and analysis, and extending the approach to detect other malware that uses the disk.

7.1 Contributions

In this section we summarize our contributions and relate these contributions back to the ***Thesis Statement:*** The disk drive processor can be used to improve malware detection.

Identified general file-infecting virus behavior that is exhibited by a large class of viruses. We identified disk-level behaviors that are common across many file-infecting viruses. Our general behavioral rules are successful at matching viruses, because they are based on the specific disk-level actions a virus typically does when infecting programs. We have shown these behaviors to be consistent with the different types of disk-level activity a virus can generate by recording and comparing disk-level activities for a collection of viruses. Our results show that a disk-level detector can reliably characterize many different viruses that are unknown to the detector.

We developed and tested four rules that capture the fundamental file-infecting behavior of most viruses. These general rules have been shown by our tests on multiple polymorphic and metamorphic viruses to be resilient to the kinds of obfuscations that often cause problems for current virus detectors. Our rules are simple, yet they are able to detect most virus infections in our experiments. Of the four rules we tried, we found the R_oW_o and R_oW_oW rules to have comparable false negative rates with the more general rules. The R_oW_o and W rules detected a greater number of infections in our tests (but with a higher false positive rate than the R_oRW_oW or R_oW_oW rules) while the R_oRW_oW rule detected almost all file-infecting virus infections.

Unlike traditional AV engines, our rules capture the fundamental file-infecting behavior common to many viruses rather than the code fragments identified in known viruses. A detector based on these rules will not need to be updated for each new variant that is released.

Identified causes of false positives and designed methods to deal with these false positives. To test these rules we collected disk activity from eight different users. Testing our rules against this disk activity helped us evaluate their effectiveness. In testing our general behavioral rules and behavior-specific signatures, we identified some sources of false positives. All of these false positives came from a small set of specific activities including program installation, program compilation, and program updates.

We addressed these false positives by designing methods to deal with them and discussed some of the design decisions in how these methods could be implemented in a real detector. System restore is a type of activity that causes false positives. The disk could manage the restoration snapshots which would eliminate these false positives while enhancing the reliability and security of system restorations. Program updates can be handled with signed updates by having program binaries embed a public key. The disk can then verify program updates with the provided install key. Legacy program updates could be facilitated with a direct interface to the disk (e.g., a "trusted" button). When a user communicates using the direct interface, the detector is temporarily suspended to allow a

forbidden action to take place.

Developed a process for developing behavior-specific signatures. While our generic file-infecting rule detects a large class of viruses, it cannot detect all malicious behavior. We also tested three viruses and one worm that possessed other malicious disk activity (in addition to the file infection). From this disk activity, we developed and applied a process to construct specific disk-level signatures that characterized these malicious disk accesses. Each of these malware samples presented different disk-level behaviors that could not be wholly detected with our simple behavioral rules. Instead, some of their behavior required detection with behavior-specific signatures.

Previously, host-level detection could not be reliably enforced with existing malware's demonstrated ability to circumvent monitoring points (W32.Mailbot [King and Chen, 2005; Kasslin, 2006]). After applying our techniques for finding disk-level specific behaviors, we can use the disk to recognize and disallow malicious accesses.

Designed a detector and implemented a proof-of-concept implementation. We implemented a proof-of-concept detector based on our design for a disk-level detector, consisting of a Linux machine (the disk) that contained an exported VMWare disk image over iSCSI to a Windows machine. We bridged the semantic gap by using a mapping between disk blocks and files. Using this implementation shows that the path towards a production-level detector is possible.

We have laid the groundwork for a disk-level malware detector, but there are many interesting problems left unsolved. The next section discusses limitations of our current work. Section 7.3 identifies areas for future research.

7.2 Limitations

Here, we discuss some important limitations of our work, including the challenge of applying our work to other platforms, the remaining problems of the semantic gap, perfor-

mance issues, the impossibility of detecting malware that does not use the disk, and the challenges in deploying a disk-level detector.

7.2.1 Other Platforms

Our work focused on the Microsoft Windows platform because of the plethora of malware available on this platform. However, viruses also exist for both Linux and Mac OSX. Earlier in 2007, a writeup on Virus Bulletin discussed a recent file infector threat on OSX [Ferrie, 2007]. Malware has existed for multiple platforms for many years [Szor, 2005a], and it is currently on the rise for both Linux and Mac OSX [Knight, 2004; Patrizio, 2006].[1] Mac, Linux, and most other operating systems use a disk. There are a few differences including the underlying file system, the ordering of disk-level events, and the formatting of executables, but a disk-level detector should still be effective. Assuming a semantic disk model, changes may be necessary for our inference of file system actions using different file systems. Hence, we should be able to apply what we have learned to these platforms.

7.2.2 Performance

One of the driving assumptions behind this work is that the disk processor is underutilized, but we were not able to test this due to the limits of the prototype. Some analysis has been done on this before in similar work [Griffin et al., 2003; Sivathanu et al., 2004a], but we did not measure the overhead of scanning the data using our signatures and rules. Sivathanu et al.'s results for their FADED application show that maintaining a similar file system map incurs approximately four to seven percent CPU overhead across different file systems. We use a similar semantic map, and the overhead of checking our rules should be low.

A realistic performance analysis is difficult without a more complete implementation where the semantic map is dynamically updated. Because the disk firmware is proprietary,

[1] An office mate that uses a Mac OS on her Apple notebook claimed she had her computer infected recently with some malware.

it would be a significant challenge to put our code on the disk. To measure performance with the prototype detector, we would need to augment its implementation to perform an analysis.

7.2.3 Diskless Malware

The fundamental limitation of a disk-level detector is that it sees only the disk's traffic. Using a disk-level detector is analogous to a defense-in-depth strategy. The disk-level detector is not meant to replace traditional detectors. Rather, it is intended to augment the protection of the disk. Malware that spreads, for example, without using the disk would best be detected through other means (e.g., a network-based defense for malware that spreads via the network).

7.2.4 File System Encryption

File system encryption presents difficulties for disk-level detection. Some of the behavior-specific signatures rely on the disk request's contents. Microsoft Windows recently introduced BitLocker, a whole system drive encryption utility based on a trusted platform module (TPM). A TPM is a chip that stores sensitive information in a machine. Often, this sensitive information includes cryptographic keys, digital signatures, certificates, and sometimes even code. A well-known example of a TPM architecture is Arbaugh et al.'s AEGIS [Arbaugh et al., 1997], and a more recent example is Microsoft's NGSCB (formerly Palladium) [Microsoft, 2007b]. If the user does not have a TPM, a workaround can be done with a USB-stored key.

BitLocker encrypts data at the sector level [Ferguson, 2006], and our behavioral rules would still work (the OS must access data at the block level, but a block typically consists of eight sectors). For the general behavioral rules, encryption should not be a concern, because our state machines do not depend on data content (see Section 4.1). For more specific signatures that could rely on data content (see Section 5.2), this could be an issue.

The BitLocker encryption is only for the most expensive versions of Windows, Vista Ultimate [Microsoft, 2007e]. In other file system encryption implementations [Apple, 2007], only the user's home directory files are encrypted (not application files). Yet another possibility recently introduced in the market is letting the disk do the encryption [Seagate, 2006; Hitachi, 2006]. Drives that do the encryption and decryption offer speed and security advantages over software-based disk encryption. If the disk controls encryption/decryption, then it could always check the rules and signatures just after decryption or just before encryption.

7.2.5 Deployment

In Chapter 3, we presented the semantic gap. This gap must be bridged by a disk-level detector to understand and act upon disk-level behavior. Our prototype implementation used a static view of the data, and we lacked the necessary information to maintain an up-to-date dynamic view of the file system. A full implementation for deployment may require cooperation from Microsoft or reverse engineering other file system data structures that are not currently well understood.

Other issues include our mechanisms in dealing with false positives. Adoption of our false positive mechanisms like providing a key on program installation may be slow. Disk drive vendors will need to be convinced to put a trusted interface on their disk (when updating legacy programs). In the meantime, we could allow the trusted interface to go through the host at the expense of malware being able to intercept or generate interrupts to and from this interface (e.g., a password, or usb stick with key). This would eliminate the need for a direct interface to the disk at the sacrifice of security.

7.3 Future Work

Some areas of future research seem promising including developing techniques to automatically find disk-level signatures, incorporating deeper data-flow analyses to detect

evasive viruses, and extending detection to other types of malware.

7.3.1 Automatically Finding Disk-level Signatures

Trace Collection. When creating a signature it is important to have traces that contain the malicious disk activity while minimizing the recording of the benign disk activity. Creating a signature from traces that contain a lot of benign disk activity can be difficult (e.g., higher potential for false positives). While running our traces, we ran as few programs as possible to ensure data that was generated was coming from the malware. Furthermore, much of the data generated may be difficult to associate to a particular higher-level action even while using process information (which is not available at the disk-level).

To better understand the malware, we traced other high-level information including the parent process id of a disk request. With this increased understanding, we can create better signatures. As another example, the Efish virus uses multiple threads of the same process to find and infect files. We could use thread-level information in our signatures to properly specify each thread's disk activity. The disk does not have access to high-level information like OS thread activity, but we could use the thread-level information to create more effective disk-level signatures by concentrating on the disk-level data that was important in the trace.

While filtering for important data in traces, we also need to add data that may have been missed in the traces. If the malware executions do not trigger some specific malicious action (e.g., malware executes payload on some specific date), we may need to run additional traces that will trigger the missed malicious behavior.

Pruning and Generalizing the Disk Traces. With the collected traces, we can begin pruning the data for interesting disk activity. Our goal is to maximize the amount of data that we can use in our signatures. Techniques for discarding data must be thoroughly tested, but good techniques will save time in signature refinement.

If the signature is too specific, then it can miss variants or other virus generations. We can generalize specific filenames, for example, to capture future variants if the resulting

signature contains a file name that looks randomized. With filenames, we can also analyze the contents of the requests to give them context. If the data is interesting (e.g., a write to an interesting file), then the data contents could also be used in a signature.

One way to help generalize signatures is through heuristics. Heuristic signatures are used to identify specific properties of viruses. *Static heuristics* rely on non-changing attributes of the virus such as recognizing suspicious file structures, and *dynamic heuristics* use emulation [Szor, 2005a]. We would like to use heuristics in a different sense - to find better signatures in an automated fashion. These heuristic signatures could provide attributes of disk-level behavior that may not be immediately added to a final signature. If a trend or common heuristic is identified, then this heuristic could eventually be put into a real behavior-specific signature.

Some experiments revealed interesting techniques that may be added as a heuristic. When we ran traces on actual viruses, we found many viruses that used the Windows cryptographic API to seed their pseudorandom number generator. Disk accesses of this API could be construed as part of a heuristic signature (legitimate programs needing a pseudorandom seed will often use this API as well). We also observed attempts at modifying system files that Windows would respond to with Windows File Protection enabled [Microsoft Help and Support, 2007]. This could also become part of a heuristic disk-level signature.

For instance, once a certain executable is read, we may notice a trend that certain files on the disk are read. We can build correlations of disk-level activity based on other disk-level activity. This has actually been done in the past by finding correlations between block-level events [Li et al., 2004]. They exploited block correlations to prefetch data and discuss how block correlations can help in physical block locations on the disk. Observed heuristics could be used to identify other malware that share similar heuristics, but may not have the observed activity in their signatures. This could help a malware analyst quickly identify an unknown malware sample by showing similarities in disk-level be-

havior. Using these heuristics in an actual signature would require a lot of testing to make sure matched events would not raise a false alarm.

Automated Signature Learning. In our initial work, we found our signatures through both experimentation and gaining contextual information about disk-level traces through reading detailed analyses on specific viruses. An automated method may be more reliable and efficient.

One of our main challenges in understanding disk-level behavior was sifting through so much data in order to generate a useful disk-level signature. As we became better at reading disk-level traces, we were able to develop signatures more quickly. Finding disk-level patterns from recorded disk activity can be challenging. Automating this process would enable us to create better signatures quickly and cheaply. This automation requires performing a pre-analysis of the malware to make sure the malware exhibits its malicious disk-level behavior, collecting the disk-level traces of the malware executions, pruning the traces for possible interesting disk-level activity, analyzing the pruned data to generate the candidate behavior-specific signatures, testing the candidate signatures effectiveness, and refining the signatures if necessary (see Figure 5.1 from Section 5.1).

The machine learning community has developed techniques that find minimal deterministic finite state machines using examples of accepted and non-accepted input [Angluin, 1987]. Angluin's L^* algorithm is an *online* algorithm that can query a teacher to find a finite-state machine (FSM). Applying the L^* algorithm to the problem of learning general behavioral rules does pose some challenges including having a teacher that could answer if an input was accepted. In the beginning, we could not have programmed an adequate teacher as required by this algorithm until later finding the actual finite state machines. Only by deeper inspection were we able to determine if specific disk requests were relevant for inclusion in a state machine. Moreover, these behavioral rules will not change often making it less profitable to spend effort on algorithms that could learn these rules. However, insight from making the rules may benefit in applying an automated algorithm to behavior-specific signatures.

Another problem in applying an automated algorithm is our inability to provide input (or traces) that will be useful in learning the state machines. Deciding which parts of the training data are important can be difficult even for a human. Our goal is to find a FSM that matches the disk requests of malicious programs (low false negative rate) and not the disk requests of benign programs (low false positive rate) with little or no aid from a teacher. This suggests another approach may be of use like an *offline* algorithm (computes a FSM without queries from a set of finite examples) or some combination of an offline and online algorithm that computes the signatures (e.g., a modified version of Biermann's learning algorithm by Grinchtein and Leucker [Biermann and Feldman, 1972; Grinchtein and Leucker, 2006]). Better solutions will likely involve a partially automated solution that may occasionally require human intervention. Any improvement over the manual process of generating behavior-specific signatures would be useful.

Testing and Refining Behavior-Specific Signatures. Another aspect of this work is testing and refining the disk-level behavior-specific signatures. When we find interesting disk activity in our trace analysis, we must verify if this disk activity occurs in multiple traces or is an anomaly. This may require generating additional traces to verify the presence of the data in those traces.

An additional step in the signature testing is the signature refinement. If, after the initial signature, we find that the signature is ineffective, we will want to refine the signature by adding more information. We currently evaluate if a signature is likely to have too many false positives by running the signature over a large number of disk-level traces. This evaluation can take prohibitively long, and there may be helpful speedups including adding more memory to deal with large traces, parallelizing the false positive trace checking, and perhaps better algorithms at scanning the traces. We envision a complete solution for testing and refining signatures to be an automated algorithm that can iteratively find a signature based on certain parameters including the maximum size of a signature, the acceptable number of false positives (very low), and the acceptable number (and possibly type) of false negatives.

7.3.2 Data Flow and Analysis

Section 6.1 discussed ways to evade certain signatures by using the disk in ways that do not match the general infection rules but still accomplish the needed task. For example, the virus could read an executable, delete the executable, and then replace the executable with an infected version (an overwriting virus). Through a deeper analysis of the data flow, the detector should be able to capture evasive malware. If the disk compares data from a newly created executable with a previously deleted executable, it could perhaps recognize the malicious behavior. The detector may have to inspect the data more to avoid false positives (e.g., legacy patches), but this gives an idea of how the detector can perform a deeper analysis to capture different behaviors that may circumvent monitored behaviors.

Another similar type of analysis can be done across reboots. Suppose a virus reads an executable and then writes out a file as a simple text file. Later, after a reboot, the virus renames the text file to an executable (after getting the user to run the virus again). Among our current rules and signatures, we do not carry our state machines across reboots, but this type of data flow analysis may become necessary.

Other types of disk-level behavioral morphing viruses can be concocted. Some of our signatures depend on certain ordering of events. Malware could take advantage of this by reordering the way it accesses a disk (assuming its accesses can be reordered), and we may need to monitor these behaviors and allow for certain event reorderings.

7.3.3 Application to Different Types of Malware

This thesis focuses on viruses. In this section we consider applying a disk-level detector in detecting other types of malware that use the disk. For much of this malware, behavior-specific signatures will be needed. We now survey different types of malware and describe their defining characteristics.

Worms. Worms are malicious programs whose primary means of replication is the network. For example, many current worms use known vulnerabilities like buffer overflows

or email to spread (e.g., W32.Beagle.A@mm [Hindocha, 2004], W32.Slammer [Internet Security Systems, 2003], and W32.Blaster [Dougherty et al., 2003]). Worms may also use the disk in other interesting ways. For example, W32.Blaster creates registry entries to flood the windows update service [Dougherty et al., 2003]. As worms use the disk, we can use disk-level detection to specifically detect disk-based worms (see Section 5.2.2). Since some worms do not need to generate any disk traffic to propagate (some can be removed by a simple reboot [Internet Security Systems, 2003]), disk-level techniques are not effective for network-based worms.

Spyware. Spyware is often correlated with stealing or tracking information on the infected machine. In addition to the privacy issues of spyware, another major problem is its burdening of system resources like the processor, memory, and disk [McDermott, 2005; Purdue University, 2005]. Although this thesis does not address spyware directly, most spyware manifests malicious disk activity, because many are in the form of Browser Helper Objects (BHOs) installed as a dynamically linked library (DLL). Moshchuk et al. reported that 85% of spyware-infected executables found on a crawl of the Internet are due to browser hijacks like BHOs, and much of the spyware they found installs at least one executable [Moshchuk et al., 2006]. Some spyware used the disk even more (five percent of the spyware-infected executables in their study installed five or more programs). The important point is that spyware needs the disk to operate, and a disk-level defense should be useful to defend against these types of attacks.

Trojan Horses. From the Greek story of the Trojan horse, a software Trojan horse is a program that pretends to be a harmless application while actually performing some malicious action unbeknownst to the user. For example, it can download and install other malware in the background while allowing the user to play a game. Because many trojans do use the disk to drop, download, or install other malware, a disk-level defense should be possible.

Trojan malware has significantly increased with the focus on financial gains using malware [Braverman et al., 2006; Gullotto et al., 2007]. There are various kinds of trojans

demonstrating different malicious behaviors. Some download other malware onto a compromised machine (*downloaders*) while others will create or drop files that will be later used (*droppers*). Other trojans will steal passwords while others may log keystrokes (*keyloggers* often store a log of keystrokes to a file before sending it to a remote site), and some are used in a botnet to create a *backdoor*, an unauthorized way of remotely controlling a system. Downloaders, droppers, keyloggers, and backdoors should be detectable, because they all will eventually use the disk.

A story in late 2005 shows the scale and use of a Trojan [Eckelberry, 2005]. What was originally thought to be a variant of a known piece of malware called CoolWebSearch turned out to be a keylogger. When this was realized, the destination of the data lead to a large identity theft ring where thousands of users' data had been compromised. Because keyloggers use the disk both to install and store logs, a disk-level detector may be configured to monitor for the keyloggers or the sensitive data they log.

Rootkits. Like trojans, rootkit usage has been on the rise [Braverman et al., 2006; Gullotto et al., 2007]. Rootkits are often installed by an attacker to maintain stealth access to a compromised machine. Rootkits can be especially hard to detect given that they can modify parts of the OS that the host-level detectors depend upon to operate [Kasslin, 2006]. To remain persistent across reboots, rootkits need to change the disk state while maintaining stealth.

A disk detector could detect a rootkit based on its specific usage of the disk. While current rootkit signature creation approaches suffer from the rootkit actively changing its behavior on detecting a virtual machine (e.g., W32/Polip [Ferrie, 2006b]), a disk-level detector can reliably see all the rootkit's disk traffic. This makes it easier to obtain signatures, and the detector's location ensures the rootkit's disk-level behavior will be seen.

Many rootkit detectors including Strider Ghostbuster [Wang et al., 2005], RootkitRevealer [Cogswell and Russinovich, 2006], and Blacklight [F-Secure, 2006] will compare a high-level view of data to a low-level view, because the rootkit will attempt to hide low-level data from high-level reads of that data. For example, both a call to `ReadFile`() to

a certain file and typing `dir` with the proper arguments at the command prompt are examples of a high-level scan of the OS. If a rootkit is hiding data, the high-level view of the data should mismatch the low-level view. The assumption is that the rootkit has not compromised the code that generates the low-level view of the data. If the rootkit has compromised the integrity of the OS to hide certain data, the high-level view will not show anything amiss. Rutkowska observes that these host-level techniques could be circumvented by not issuing disk traffic through the normal detection monitoring points [Rutkowska, 2005] (e.g. W32.Mailbot [Kasslin, 2006]).

Comparing the scans of data provides an interesting opportunity for a disk-level detector. If we use a disk-level detector for a low-level read of the data, then its view will not be compromised by a rootkit. However, the challenge is comparing the two views of the OS, because a dependable channel between the disk and the high-level scan data does not exist. Malware can intercept and change any transmission of either of the scans.

Other methods of detection may exist. Because the detector can see the rootkit's usage of the disk, it should be able to silently monitor disk usage. Discrepancies in the way data is handled may become apparent. For instance, the disk may notice a correlation between a commonly used program beginning execution (e.g., Internet Explorer) and certain executable files being accessed. If anomalies are noticed in the way data is accessed, they may indicate the presence of malware. Better rules about the disk usage may more easily detect rootkits.

When malware digs deeper into the OS to remain hidden, we need a low-level detector to actually detect it. When the core parts of the OS change beneath the detector, it becomes ineffective at detecting malware. A disk-level detector is invulnerable to these types of attack.

7.4 Final Thoughts

Both industry experience and recent research have demonstrated that effective malware detection is difficult. Current approaches suffer from fundamental flaws including dependence on host-level defenses that can be circumvented, use of blacklisting, and over-reliance on static analysis. Using the disk processor to improve malware detection is desirable since nearly every system already has a disk processor and it is privy to all disk accesses. Our approach can detect most file infecting viruses with simple generic rules, and it can detect other malicious disk activity (e.g., worms) through behavior-specific signatures. Disk-level behavioral malware detection offers a promising approach over traditional malware detectors by monitoring disk-level behavior to detect malware.

References

[Acharya et al., 1998] Acharya, A., Uysal, M., and Saltz, J. (1998). Active disks: Programming model, algorithms, and evaluation. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages (ASPLOS)*.

[Acohido and Swartz, 2004] Acohido, B. and Swartz, J. (2004). Going price for network of zombie pcs: $2,000-3,000$. *USA Today*. `http://www.usatoday.com/tech/news/computersecurity/2004-09-08-zombieprice_x.htm`.

[Agrawal et al., 2007] Agrawal, N., Bolosky, W. J., Douceur, J. R., and Lorch, J. R. (2007). A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. USENIX.

[aimtrading, 2006] aimtrading (2006). Hddguru forums. `http://forum.hddguru.com/newbie-info.-from-and-for-newbies-..-about-firmware.-sa.-etc-vp45284.html`.

[Altaparmakov et al., 2007] Altaparmakov, A., Russon, R., Szakacsits, S., Sornes, E., and Pakhuchiy, Y. (2007). mkntfs source code - part of the linux-ntfs project. `http://www.linux-ntfs.org/doku.php?id=downloads`.

[Anderson, 2005] Anderson, D. (2005). Disc drive technology update. `http://storageconference.org/2005/emerging-technology-panel/et-anderson.pdf`.

[Angluin, 1987] Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Proceedings of the Journal of Information and Computation*, 75(2):87–106.

[Apple, 2007] Apple (2007). Filevault. http://www.apple.com/macosx/features/filevault/.

[Arbaugh et al., 1997] Arbaugh, W. A., Farber, D. J., and Smith, J. M. (1997). A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE.

[ARM Enterprise, 2007] ARM Enterprise (2007). Seagate cheetah family of disk drives. http://www.arm.com/markets/enterprise_solutions/armpp/462.html.

[ARM9E Family, 2007] ARM9E Family (2007). Arm966e-s. http://www.arm.com/products/CPUs/ARM966ES.html.

[Arpaci-Dusseau et al., 2006] Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Bairavasundaram, L. N., Denehy, T. E., Popovici, F. I., Prabhakaran, V., and Sivathanu, M. (2006). Semantically-smart disk systems: Past, present, and future. *Sigmetrics Performance Evaluation Review (PER)*, 33(4):29–35.

[Balakrishnan and Reps, 2004] Balakrishnan, G. and Reps, T. (2004). Analyzing memory accesses in x86 executables. In *International Conference on Compiler Construction*.

[Bellissimo et al., 2006] Bellissimo, A., Burgess, J., and Fu, K. (2006). Secure software updates: Disappointments and new challenges. In *In the Proceedings of USENIX Hot Topics in Security (HotSec)*.

[Bender et al., 2006] Bender, M. A., Farach-Colton, M., and Mosteiro, M. A. (2006). Insertion sort is o(n log n). *Theory of Computing Systems*, 39(3):391–397. Special Issue on *SPAA '00*.

[Biermann and Feldman, 1972] Biermann, A. W. and Feldman, J. A. (1972). On the synthesis of finite-state machines from samples of their behavior. In *In IEEE Transactions on Computers*, volume C-21, pages 592–597. IEEE.

[Black Jack, 2001] Black Jack (2001). Anti heuristic techniques. http://vx.netlux.org/lib/vbj01.html.

[Bosveld, 2007] Bosveld, J. (2007). Jotti's malware scan 2.99. http://virusscan.jotti.org.

[Braverman et al., 2006] Braverman, M., Williams, J., and Mador, Z. (2006). Microsoft security intelligence report. http://www.microsoft.com/downloads/details.aspx?FamilyId=1C443104-5B3F-4C3A-868E-36A553FE2A02.

[ca, 2006] ca (2006). ca writeup on win32.sality.l. http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?ID=53100.

[Center, 2001] Center, I. A. R. (2001). Storage systems - storage tank. http://www.almaden.ibm.com/StorageSystems/Past_Projects/Storage_Tank/.

[Christodorescu and Jha, 2003] Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*.

[Christodorescu and Jha, 2004] Christodorescu, M. and Jha, S. (2004). Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 34–44.

[Christodorescu et al., 2005] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E. (2005). Semantics-aware malware detection. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*.

[Ciubotariu, 2006] Ciubotariu, M. (2006). What next? trojan.linkoptimizer. *Virus Bulletin*, pages 6–10. http://www.virusbtn.com/virusbulletin/archive/2006/12/vb200612-trojan-linkoptimizer.

[Clark, 2005] Clark, S. (2005). U3 - 'official' portable usb apps platform. http://www.everythingusb.com/u3.html.

[Cluster File Systems, Inc., 2007] Cluster File Systems, Inc. (2007). About the lustre architecture. http://web.archive.org/web/20060518225423/http://lustre.org/architecture.html.

[Cogswell and Russinovich, 2006] Cogswell, B. and Russinovich, M. (2006). Sysinternals rootkitrevealer. http://www.sysinternals.com/utilities/rootkitrevealer.html.

[Cohen, 1987] Cohen, F. (1987). Computer viruses: Theory and experiments. *Computers and Security*, 6(1):22–35.

[Collake, 2000] Collake, J. (2000). W2k undocumented registry setting fully disables windows file protection. http://www.ntbugtraq.com/default.aspx?pid=36&sid=1&A2=ind0006&L=ntbugtraq&P=9861.

[Collake, 2006] Collake, J. (2006). Hacking windows file protection. http://www.bitsum.com/aboutwfp.asp.

[Cornell et al., 2004] Cornell, B., Dinda, P. A., and Bustamante, F. E. (2004). Wayback: A user-level versioning file system for linux. In *Proceedings of the 2004 USENIX Technical Conference*. USENIX.

[Crandall et al., 2006] Crandall, J. R., Wassermann, G., de Oliveira, D. A. S., Su, Z., Wu, S. F., and Chong, F. T. (2006). Temporal search: Detectin hidden malware timebombs with virtual machines. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

[Dougherty et al., 2003] Dougherty, C., Havrilla, J., Hernan, S., and Lindner, M. (2003). Cert advisory ca-2003-20 w32/blaster worm. http://www.cert.org/advisories/CA-2003-20.html.

[Drori et al., 2005] Drori, O., Pappo, N., and Yachan, D. (2005). New malware distribution methods threaten signature-based av. *Virus Bulletin*, pages 9–

11. http://www.virusbtn.com/virusbulletin/archive/2005/09/vb200509-new-malware-distribution-methods.

[Eckelberry, 2005] Eckelberry, A. (2005). More on the identity theft ring. http://sunbeltblog.blogspot.com/2005/08/more-on-identity-theft-ring.html.

[Eckmann et al., 2000] Eckmann, S. T., Vigna, G., and Kemmerer, R. A. (2000). An attack language for state-based intrusion detection. In *Proceedings of the 2000 ACM Workshop on Intrusion Detection*. ACM.

[Egele et al., 2007] Egele, M., Kruegel, C., Kirda, E., Yin, H., and Song, D. (2007). Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference*.

[EMC, 2007] EMC (2007). Emc centera. http://www.emc.com/products/systems/centera.jsp.

[Emulex, 2005] Emulex (2005). Seagate, emulex and ibm team up to demonstrate industry's first standards-compliant object-based storage devices. http://www.emulex.com/press/2005/0412-01.jsp.

[Exegy, 2007] Exegy (2007). The exegy text miner 2.0. http://www.exegy.com/productsheets.html.

[F-Secure, 2006] F-Secure (2006). Blacklight. http://www.f-secure.com/blacklight/.

[Ferguson, 2006] Ferguson, N. (2006). Aes-cbc + elephant diffuser. http://www.microsoft.com/downloads/details.aspx?FamilyID=131dae03-39ae-48be-a8d6-8b0034c92555.

[Ferrie, 2002a] Ferrie, P. (2002a). Attack of the clones. *Virus Bulletin*, pages 4–5. http://pferrie.tripod.com/papers/gemini.pdf.

[Ferrie, 2002b] Ferrie, P. (2002b). Unexpected results [sic]. *Virus Bulletin*, pages 4–5. http://pferrie.tripod.com/papers/chiton.pdf.

[Ferrie, 2003] Ferrie, P. (2003). You've got more m(1**)a(d)i(l+k)l. *Virus Bulletin*, pages 6–7. http://pferrie.tripod.com/papers/junkhtml.pdf.

[Ferrie, 2006a] Ferrie, P. (2006a). Attacks on virtual machine emulators. In *Anti Virus Asia Researcher's conference (AVAR)*. http://pferrie.tripod.com/papers/attacks.pdf.

[Ferrie, 2006b] Ferrie, P. (2006b). Tumours and polips. *Virus Bulletin*. http://pferrie.tripod.com/papers/polip.pdf.

[Ferrie, 2007] Ferrie, P. (2007). Do the macarena. http://pferrie.tripod.com/papers/macarena.pdf.

[Ferrie and Perriot, 2004] Ferrie, P. and Perriot, F. (2004). To catch efish. *Virus Bulletin*, pages 4–6. http://pferrie.tripod.com/papers/efish.pdf.

[Florio, 2003] Florio, E. (2003). W32.hllp.sality. http://www.symantec.com/security_response/writeup.jsp?docid=2006-011714-3948-99.

[Forrest et al., 1996] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE Computer Society Press.

[Garfinkel et al., 2007] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. (2007). Compatibility is not transparency: Vmm detection myths and realities. In *11th Workshop on Hot Topics in Operating Systems (HOTOS)*. USENIX.

[Garfinkel and Rosenblum, 2003] Garfinkel, T. and Rosenblum, M. (2003). A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*.

[Gordon, 1994] Gordon, S. (1994). The generic virus writer. *Proceedings of the 4th International Virus Bulletin Conference*. http://vx.netlux.org/lib/asg03.html.

[Gordon, 1995] Gordon, S. (1995). Technologically enabled crime: Shifting paradigms for the year 200. *Virus Bulletin*, 14(5):417.

[Goudey, 2004] Goudey, H. (2004). Watch the money-go-round, watch the malware-go-round. *Proceedings of the 2004 Virus Bulletin Conference*. `http://ca.com/files/VirusInformationAndPrevention/hgoudeyvb2004.pdf`.

[Griffin et al., 2003] Griffin, J., Pennington, A., Bucy, J., Choundappan, D., Muralidharan, N., and Ganger, G. (2003). On the feasibility of intrusion detection inside workstation disks. Technical Report CMU-PDL-03-106, Department of Computer Science, Carnegie Mellon University.

[Grinchtein and Leucker, 2006] Grinchtein, O. and Leucker, M. (2006). Learning finite-state machines from inexperienced teachers. In *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006*, volume 4201 of *Lecture Notes in Computer Science*.

[Gruener, 2007] Gruener, W. (2007). Vista to go: Sandisk improves u3 flash drives with microsoft's help. `http://www.tgdaily.com/content/view/32005/140/`.

[Gullotto et al., 2007] Gullotto, V., Jones, J., and Landesman, M. (2007). Microsoft security intelligence report. `http://www.microsoft.com/downloads/details.aspx?FamilyId=AF816E28-533F-4970-9A49-E35DC3F26CFE`.

[Gutmann, 2007] Gutmann, P. (2007). The commercial malware industry. `http://www.cs.auckland.ac.nz/~pgut001/pubs/malware_biz.pdf`.

[Halderman and Felten, 2006] Halderman, J. A. and Felten, E. W. (2006). Lessons from the sony cd drm episode. In *Proceedings of the 15th USENIX Security Symposium*. USENIX.

[Hayes, 2002] Hayes, B. (2002). Who goes there? an introduction to on-access scanning, part two. `http://www.securityfocus.com/infocus/1626`.

[Heasman, 2007] Heasman, J. (2007). Firmware rootkits and the threat to the enterprise. *Blackhat DC 2007*. https://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf.

[Hile et al., 1994] Hile, J., Gray, M., and Wakelin, D. (1994). In transit detection of computer virus with safeguard.

[Hindocha, 2004] Hindocha, N. (2004). Technical details of w32.beagle.a@mm. http://www.symantec.com/security_response/writeup.jsp?docid=2004-011815-3332-99&tabid=2.

[Hitachi, 2006] Hitachi (2006). Hitachi to anchor 2.5-inch hard drive leadership with quarter-terabyte product. http://www.hitachi.us/Apps/hitachicom/content.jsp?page=PressReleases/details/11012006.html&path=jsp/hitachi/aboutus/Press-Media/.

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

[Hutchinson et al., 1999] Hutchinson, N. C., Manley, S., Federwisch, M., Harris, G., Hitz, D., Kleiman, S., and O'Malley, S. (1999). Logical vs. physical file system backup. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. USENIX.

[Hypponen et al., 2003] Hypponen, M., Tocheva, K., Erdelyi, G., and Podrezov, A. (2003). F-secure ganda virus description. http://www.f-secure.com/v-descs/ganda.shtml.

[IBM, 2007] IBM (2007). Ibm 2.5-inch 15k rpm 73 gb sff sas hdds provide highest spindle speed performance for storage in system x, bladecenter, and storage, and i/o blades. http://www-01.ibm.com/common/ssi/rep_ca/6/877/ENUSZG07-0366/ENUSZG07-0366.PDF.

[Ilet, 2004] Ilet, D. (2004). Most spam generated by botnets, says expert. http://news.zdnet.co.uk/security/0,1000000189,39167561,00.htm.

[Ilgun et al., 1995] Ilgun, K., Kimmerer, R. A., and Porras, P. A. (1995). State transition analysis: A rule-based intrusion detection approach. In *Proceedings of the IEEE Transactions on Software Engineering*, volume 21. IEEE.

[Intel Corporation and Seagate Technology, 2003] Intel Corporation and Seagate Technology (2003). Serial ata native command queuing. http://www.seagate.com/content/docs/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf.

[Internet Security Systems, 2003] Internet Security Systems (2003). Microsoft sql slammer worm propagation. http://xforce.iss.net/xforce/alerts/id/advise140.

[Jiang et al., 2007] Jiang, X., Wang, X., and Xu, D. (2007). Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM.

[Kaspersky Lab, 2005] Kaspersky Lab (2005). Kaspersky anti-virus engine technology. http://www.opsec.com/solutions/partners/downloads/Kaspersky_EngineTech_WP.pdf.

[Kasslin, 2006] Kasslin, K. (2006). Kernel malware: The attack from within. In *Proceedings of the 2006 AVAR Conference*. http://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf.

[Keeton et al., 1998] Keeton, K., Patterson, D. A., and Hellerstein, J. M. (1998). A case for intelligent disks (idisks). In *Proceedings of the 1998 Special Interest Group on Management of Data (SIGMOD)*, volume 27. ACM.

[Keizer, 2006] Keizer, G. (2006). Anti-spyware vendors mad about consumer reports test methods. http://www.informationweek.com/internet/showArticle.jhtml?articleID=192300458.

[Keizer, 2007] Keizer, G. (2007). Symantec false positive cripples thousands of chinese pcs. http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9019958&intsrc=hm_list.

[Kim and Spafford, 1994] Kim, G. and Spafford, E. (1994). The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM.

[King and Chen, 2005] King, S. and Chen, P. (2005). Subvirt: Implementing malware with virtual machines. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*.

[Kirda et al., 2006] Kirda, E., Kruegel, C., Banks, G., Vigna, G., and Kemmerer, R. A. (2006). Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*.

[Knight, 2004] Knight, W. (2004). Pundits predict malware may target linux. http://software.newsforge.com/article.pl?sid=04/11/10/2014240&tid=78.

[Lakhotia and Kumar, 2004] Lakhotia, A. and Kumar, E. U. (2004). Abstract stack graph to detect obfuscated calls in binaries. In *4th IEEE International Workshop on Source Code Analysis and Manipulation*.

[Lakhotia and Mohammed, 2004] Lakhotia, A. and Mohammed, M. (2004). Imposing order on program statements to assist anti-virus scanners. In *11th Working Conference on Reverse Engineering*, pages 161–170.

[Li et al., 2004] Li, Z., Chen, Z., Srinivasan, S. M., and Zhou, Y. (2004). C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*. USENIX.

[Lord Julus, 1998] Lord Julus (1998). Anti-debugger & anti-emulator lair. http://vx.netlux.org/lib/vlj03.html.

[Marx, 2004] Marx, A. (2004). Anti-virus outbreak response testing and impact presentation. http://www.av-test.org/down/papers/2004-09_vb_2004.zip.

[Matasano Security, 2006] Matasano Security (2006). Ignore igor muttik's "retrospective" antivirus testing method. http://www.matasano.com/log/434/ignore-igor-muttiks-retrospective-antivirus-testing-method/.

[McAfee, 2002] McAfee (2002). W32/hll.ow.elitiamo. http://vil.nai.com/vil/content/v_99744.htm.

[McDermott, 2005] McDermott, J. (2005). Spyware clogging network arteries. http://www.windowsecurity.com/whitepapers/Spyware-Network.html.

[Mesnier et al., 2005] Mesnier, M., Ganger, G., and Riedel, E. (2005). Object-based storage. 24:31–34. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?tp=&arnumber=1462464&isnumber=31453.

[Microsoft, 2001] Microsoft (2001). Use system restore to undo changes if problems occur. http://www.microsoft.com/windowsxp/using/helpandsupport/learnmore/systemrestore.mspx.

[Microsoft, 2003a] Microsoft (2003a). Microsoft to acquire antivirus technology from gecad software. http://www.microsoft.com/presspass/press/2003/jun03/06-10GeCadPR.mspx.

[Microsoft, 2003b] Microsoft (2003b). Microsoft to acquire antivirus technology from gecad software. http://www.microsoft.com/presspass/press/2003/jun03/06-10gecadpr.mspx.

[Microsoft, 2003c] Microsoft (2003c). Ntfs technical reference. http://technet2.microsoft.com/windowsserver/en/library/81cc8a8a-bd32-4786-a849-032-45d68d8e41033.mspx.

[Microsoft, 2006] Microsoft (2006). Microsoft portable executable and common object file format specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[Microsoft, 2007a] Microsoft (2007a). Ifs kit - installable file system kit. http://www.microsoft.com/whdc/devtools/ifskit/default.mspx.

[Microsoft, 2007b] Microsoft (2007b). Microsoft next-generation secure computing base. http://www.microsoft.com/resources/ngscb/default.mspx.

[Microsoft, 2007c] Microsoft (2007c). Microsoft windows installer. http://msdn2.microsoft.com/en-us/library/aa372866.aspx.

[Microsoft, 2007d] Microsoft (2007d). Registry hives. http://msdn2.microsoft.com/en-us/library/ms724877.aspx.

[Microsoft, 2007e] Microsoft (2007e). Windows vista: Compare editions. http://www.microsoft.com/windows/products/windowsvista/editions/choose.mspx.

[Microsoft, 2008] Microsoft (2008). Windows Vista: Features Explained: Performance. http://www.microsoft.com/windows/products/windowsvista/features/details/performance.mspx.

[Microsoft Help and Support, 2007] Microsoft Help and Support (2007). Description of the windows file protection feature. http://support.microsoft.com/kb/222193.

[Mohammed, 2003] Mohammed, M. (2003). *Zeroing in on Metamorphic Computer Viruses*. PhD thesis, University of Louisiana at Lafayette, The Center for Advanced Computer Studies.

[Moser et al., 2007a] Moser, A., Kruegel, C., and Kirda, E. (2007a). Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. IEEE.

[Moser et al., 2007b] Moser, A., Kruegel, C., and Kirda, E. (2007b). Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*.

[Moshchuk et al., 2006] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M. (2006). A crawler-based study of spyware on the web. In *Proceedings of the 13th Network and Distributed System Security Symposium*.

[Nachenburg, 1997] Nachenburg, C. (1997). Polymorphic virus detection module. `http://www.patentstorm.us/patents/5696822-description.html`.

[Nagar, 2003] Nagar, R. (2003). Filter manager. `http://download.microsoft.com/download/f/0/5/f05a42ce-575b-4c60-82d6-208d3754b2d6/Filter_Manager.ppt`.

[Netezza, 2007] Netezza (2007). Product overview: The netezza performance server data warehouse appliance. `http://www.netezza.com/products/data-warehouse-appliance-products.cfm`.

[Nullsoft, 2007] Nullsoft (2007). Nullsoft scriptable install system (nsis). `http://nsis.sourceforge.net/Main_Page`.

[Offensive Computing, 2007] Offensive Computing (2007). Community malicious code research and analysis. `http://www.offensivecomputing.net`.

[Patrizio, 2006] Patrizio, A. (2006). Linux malware on the rise. http://www.internetnews.com/dev-news/article.php/3601946.

[PC World, 2007] PC World (2007). Top 10 internal hard drives. http://www.pcworld.com/printable/article/id,123680/printable.html.

[Pennington et al., 2003] Pennington, A., Strunk, J., Griffin, J., Soules, C., Goodson, G., and Ganger, G. (2003). Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the 12th USENIX Security Symposium*. USENIX.

[Perriot and Ferrie, 2004] Perriot, F. and Ferrie, P. (2004). Principles and practise of x-raying. *Virus Bulletin*, pages 51–66. http://pferrie.tripod.com/papers/x-raying.pdf.

[Perriot et al., 2003] Perriot, F., Szor, P., and Ferrie, P. (2003). Striking similarities: Win32/simile and metamorphic virus code. http://www.symantec.com/avcenter/reference/striking.similarities.pdf.

[Peterson and Burns, 2005] Peterson, Z. N. J. and Burns, R. (2005). Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212.

[Petroni and Hicks, 2007] Petroni, N. L. and Hicks, M. (2007). Automated detection of persistent kernel and control-flow attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM.

[Petroni Jr. et al., 2004] Petroni Jr., N. L., Fraser, T., Monlina, J., and Arbaugh, W. A. (2004). Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th USENIX Security Symposium*. USENIX.

[Preda et al., 2007] Preda, M. D., Christodorescu, M., Jha, S., Seshia, S. A., and Debray, S. (2007). A semantics-based approach to malware detection. In *Proceedings of the 34th Symposium on Principles of Programming Languages (POPL)*.

[Purdue University, 2005] Purdue University (2005). Spyware. http://www.purdue.edu/securepurdue/bestPractices/spyware.cfm.

[QLogic, 2007] QLogic (2007). Resources for qlogic 4050c. http://support.qlogic.com/support/product_resources.asp?id=962.

[Radkov et al., 2004] Radkov, P., Yin, L., Goyal, P., Sarkar, P., and Shenoy, P. (2004). A performance comparison of nfs and iscsi for ip-networked storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX.

[Reliable Antivirus, 2006] Reliable Antivirus (2006). Rav antivirus updates. http://web.archive.org/web/*/http://www.ravantivirus.com/pages/dldupdate.php?type=Engine.

[Riedel et al., 1998] Riedel, E., Gibson, G., and Faloutsos, C. (1998). Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th Conference on Very Large Databases (VLDB)*.

[Russinovich and Cogswell, 2006] Russinovich, M. and Cogswell, B. (2006). Filemon for windows 7.04. http://www.microsoft.com/technet/sysinternals/utilities/filemon.mspx.

[Russinovich and Solomon, 2001] Russinovich, M. and Solomon, D. (2001). Windows xp: Kernel improvements create a more robust, powerful, and scalable os. http://msdn.microsoft.com/msdnmag/issues/01/12/XPKernel/.

[Russinovich and Solomon, 2004] Russinovich, M. E. and Solomon, D. A. (2004). *Microsoft Windows Internals*. Microsoft Press, 4 edition.

[Rutkowska, 2005] Rutkowska, J. (2005). Thoughts about cross-view based rootkit detection. http://invisiblethings.org/papers/crossview_detection_thoughts.pdf.

[Rutkowska, 2006] Rutkowska, J. (2006). Rootkit hunting vs. compromise detection. http://invisiblethings.org/papers/rutkowska_bhfederal2006.ppt.

[SanDisk, 2007] SanDisk (2007). Frequently asked questions about u3. http://www.sandisk.com/Retail/Default.aspx?CatID=1450.

[Santry et al., 1999] Santry, D. S., Feeley, M. J., Hutchinson, N. C., Veitch, A. C., Carton, R. W., and Ofir, J. (1999). Deciding when to forget in the elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*. ACM.

[Satran et al., 2004] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M., and Zeidner, E. (2004). Internet small computer systems interface (iscsi). http://www.ietf.org/rfc/rfc3720.txt.

[Seagate, 2006] Seagate (2006). Drivetrust technology: A technical overview. http://www.seagate.com/docs/pdf/whitepaper/TP564_DriveTrust_Oct06.pdf.

[Seagate Press Release, 2005] Seagate Press Release (2005). Hardware-based full disc encryption security. http://web.archive.org/web/20050613011545/http://www.seagate.com/cda/newsinfo/newsroom/releases/article/0,,2732,00.html.

[Seagate Technology, 2007] Seagate Technology (2007). Speed considerations. http://www.seagate.com/www/en-us/support/before_you_buy/speed_considerations/.

[Securitas Technologies, 2006] Securitas Technologies, I. (2006). Security threats. http://www.securitastech.com/marketplace_st.htm.

[Shin et al., 2006] Shin, S., Jung, J., and Balakrishnan, H. (2006). Malware prevalence in the kazaa file-sharing network. In *Proceedings of the 2006 Internet Measurement Conference (IMC)*. http://nms.lcs.mit.edu/papers/imc145s-shin.pdf.

[Shinotsuka, 2005] Shinotsuka, H. (2005). Symantec writeup on w32.linkbot.m. `http://www.symantec.com/security_response/writeup.jsp?docid=2005-052109-2651-99`.

[Silberstein, 2004] Silberstein, M. (2004). Designing a cam-based coprocessor for boosting performance of antivirus software. `http://www.technion.ac.il/~marks/docs/AntivirusReport_revised_version.pdf`.

[Singh and Lakhotia, 2002] Singh, P. K. and Lakhotia, A. (2002). Analysis and detection of computer viruses and worms: An annotated bibliography. In *ACM SIGPLAN Notices*, volume 37, pages 29–35, New York, NY, USA. ACM Press.

[Sivathanu et al., 2006] Sivathanu, G., Sundararaman, S., and Zadok, E. (2006). Type-safe disks. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

[Sivathanu, 2005] Sivathanu, M. (2005). *Semantically-Smart Disk Systems*. PhD thesis, University of Wisconsin - Madison.

[Sivathanu et al., 2004a] Sivathanu, M., Bairavasundaram, L. N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2004a). Life or death at block-level. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation (OSDI)*. USENIX.

[Sivathanu et al., 2004b] Sivathanu, M., Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2004b). Improving storage system availability with D-GRAID. In *Proceedings of the USENIX 2004 Conference on File and Storage Technologies (FAST)*, pages 15–30, San Francisco, CA. University of Wisconsin, Madison, USENIX Association.

[Sivathanu et al., 2003] Sivathanu, M., Prabhakaran, V., Popovici, F. I., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2003). Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*. USENIX.

[Song et al., 2007] Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., and Stolfo, S. J. (2007). On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 541–551, New York, NY, USA. ACM.

[Stepan, 2006] Stepan, A. (2006). Improving proactive detection of packed malware. *Virus Bulletin*, pages 11–13. `http://www.virusbtn.com/pdf/magazine/2006/200603.pdf`.

[Strunk et al., 2003] Strunk, J. D., Goodson, G. R., Scheinholtz, M. L., Soules, C. A. N., and Ganger, G. R. (2003). Self-Securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Syposium on Operating Systems Design and Implementation (OSDI)*, pages 165–180. USENIX.

[Symantec, 1997a] Symantec (1997a). Understanding heuristics: Symantec's bloodhound technology. `http://www.symantec.com/avcenter/reference/heuristc.pdf`.

[Symantec, 1997b] Symantec (1997b). W97.melissa.a. `http://www.symantec.com/avcenter/venc/data/mailissa.html`.

[Symantec, 2003] Symantec (2003). W32.velost. `http://www.symantec.com/avcenter/venc/data/w32.velost.html`.

[Symantec, 2006] Symantec (2006). Norton antivirus updates for windows. `http://securityresponse.symantec.com/avcenter/download/pages/US-N95.html`.

[Szor, 1996] Szor, P. (1996). Nexiv_der: Tracing the vixen. *Virus Bulletin*, pages 11–12. `http://www.virusbtn.com/pdf/magazine/1996/199604.pdf`

[Szor, 2001] Szor, P. (2001). Drill seeker. *Virus Bulletin*, pages 8–9. `http://www.virusbtn.com/pdf/magazine/2001/200101.pdf`.

[Szor, 2005a] Szor, P. (2005a). *The Art of Computer Virus Research and Defense*. Addison-Wesley, 1st edition.

[Szor, 2005b] Szor, P. (2005b). Metamorphic computer virus detection. http://www.freepatentsonline.com/EP1522163.html.

[Szor and Ferrie, 2003] Szor, P. and Ferrie, P. (2003). Hunting for metamorphic. http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf.

[T10 Technical Committee, 2007] T10 Technical Committee (2007). Object-based storage devices - 2 (osd-2). http://www.t10.org/ftp/t10/drafts/osd2/osd2r01.pdf.

[Tarari, 2006] Tarari (2006). Anti-virus content processor. http://www.tarari.com/antivirus/index.html.

[The Linux-NTFS Project, 2007] The Linux-NTFS Project (2007). The linux-ntfs project documentation. http://www.linux-ntfs.org/content/view/105/44/.

[Toshiba America Information Systems, Inc., 2007] Toshiba America Information Systems, Inc. (2007). Mk3253gsx specifications. http://www.toshibastorage.com/main.aspx?Path=HardDrivesOpticalDrives/2.5-inchHardDiskDrives/MK3253GSX/MK3253GSXSpecifications.

[TrendMicro, 2007] TrendMicro (2007). Trendmicro virus pattern files. http://www.trendmicro.com/download/viruspattern.asp.

[Turner, 2006a] Turner, D. (2006a). Semantic internet security threat report: Trends for january 06 - june 06. X. http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf.

[Turner, 2007] Turner, D. (2007). Semantic internet security threat report: Trends for july 06 - december 06. XI. http://eval.symantec.com/mktginfo/enterprise/

white_papers/ent-whitepaper_internet_security_threat_report_xi_03_2007.en-us.pdf.

[Turner, 2006b] Turner, S. (2006b). Spamming malware: Parite.b and irc backdoor disable anti-spyware programs. http://blogs.zdnet.com/Spyware/?p=813.

[Virus Bulletin, 2007] Virus Bulletin (2007). The virus bulletin prevalence table. http://www.virusbtn.com/resources/malwareDirectory/prevalence/index.xml?200711.

[VXHeavens, 2006] VXHeavens (2006). Vxheavens online malware repository. http://vx.netlux.org.

[Wagner and Soto, 2002] Wagner, D. and Soto, P. (2002). Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. ACM.

[Wang et al., 2005] Wang, Y.-M., Beck, D., Vo, B., Roussev, R., and Verbowski, C. (2005). Detecting stealth software with strider ghostbuster. Technical Report MSR-TR-2005-25, Microsoft.

[Wells et al., 1999] Wells, S. E., Kynett, V. N., Kendall, T. L., Garner, R., and Brown, D. M. (1999). Method and apparatus for updating flash memory resident firmware through a standard disk drive interface. http://www.patentstorm.us/patents/5835933.html.

[Wildlist Organization, 2007] Wildlist Organization (2007). Pc viruses in-the-wild. http://www.wildlist.org/WildList/200702.htm.

[wizard, 2002] wizard (2002). Zmist: Next generation viruses coming up. http://www.wilderssecurity.com/archive/index.php/t-1693.html.

[Zhang et al., 2002] Zhang, X., van Doorn, L., Jaeger, T., Perez, R., and Sailer, R. (2002). Secure coprocessor-based intrusion detection. In *10th Workshop on ACM Special Interest Group on Operating Systems (SIGOPS)*. ACM.