Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems

A Dissertation
Presented to
The Academic Faculty

by

Julian Bennett Grizzard

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Electrical and Computer Engineering

Georgia Institute of Technology
May 2006

Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems

Approved by:

Dr. Henry L. Owen, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. John A. Copeland
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Linda M. Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: March 31, 2006

To my sister Susan Nadine Grizzard – A truly amazing person

ACKNOWLEDGEMENTS

Most importantly, I would like to deeply thank my advisor Dr. Henry Owen for his unparalleled brilliance and excellent guidance. His continuous encouragement and unwavering support have been of tremendous value to me.

I would also like to thank my committee members Dr. John Copeland, Dr. David Schimmel, Dr. Karsten Schwan, and Dr. Linda Wills. Their support and guidance have greatly impacted the direction of this work.

Next, I would like to add a special thanks Dr. John Levine who worked closely with me during the initial phases of this work.

Additionally, I would like to thank all of my family members including Mom, Dad, Susan, and Nathan for their encouraging support.

Further, I would like to thank Greg Conti for his insightful guidance and unique perspective along the way.

Also, I would like to thank a true friend of mine Charles "Robby" Simpson for his countless exciting and inspiring conversations.

Finally, I would like to thank anyone else who has inspired, encouraged, and guided this work. This list includes (in random order):

Dr. George Riley, David Dagon, Jeff Gribschaw, Dr. Sven Krasser (an $\int f(x)dx$ for you), Herbert Baines, David Creek, Andrew Davenport, Dheeraj Reddy, Fred Stakem, Dr. Wenke Lee, Kulsoom Abdullah, Lance Spitzner, Alfredo Ramos, Mike Nelson-Palmer, Dr. Mustaque Ahamad, Eric Dodson, Willard Dawson, Clay Folk, Tim Jackson, Thorsten Holz, Adam Lackorzyński, Josh Fryman, Dominic DePasquale, Alexander Warg, L4 Fiasco Group, Beverly Scheerer, Chris Clark, Sankalp Gaur, Dr. Amanda Wake, Frank Park, Lawrence Phillips, "Hobbit," Neil Joshi, Brian Rivera, Sarah Tannenbaum, Hayriye Altunbasak, Michael Richardson, Richard Craddock, Jessica Frame, Pamula Halverson, Nidhi Shah, Didier Contis, Ivan Ganev, Jonathan Torian, Dr. Felix Freiling, Chris Lee, and more.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS OR ABBREVIATIONS

ASCII	American standard code for information interchange
BIOS	basic input output system
CFG	control flow graph
CPR	condition policy rules
CPU	central processing unit
DMA	direct memory access
GB	gigabyte
ID	identification
IDS	intrusion detection system
IDT	interrupt descriptor table
I/O	input/output
IPC	interprocess communication
IPS	intrusion prevention system
IRS	intrusion recovery system
ITS	intrusion tolerant system
L4	L4 microkernel
L4Env	L4 environment
L4Linux	Linux port to L4 architecture
MD5	message-digest algorithm 5
MTTF	mean time to failure
MTTR	mean time to recovery
OS	operating system
PCI	peripheral component interconnect
RAID	redundant array of independent disks
RAM	random access memory
ROC	recovery oriented computing
SCT	system call table

SPEC	standard performance evaluation corporation
TCP	transmission control protocol
TCP/IP	transmission control protocol/internet protocol
TIKE	trusted immutable kernel extension
Trojan	Trojan horse
UML	user mode linux
VM	virtual machine
VMM	virtual machine monitor
x86	Intel i386 compatible architectures

SUMMARY

Computer systems on today's Internet are subject to a range of attacks that can compromise their intended operation. Attackers are motivated by economic incentives to perform denial of service (DoS) attacks, distribute unsolicited advertisements, and harvest identity information. In order to defend against these attacks, much work has been done on intrusion detection systems (IDS) that try to detect and stop attacks before damage occurs. IDSs have certain false positive and false negative ratios that determine their effectiveness in practical usage, and they are traditionally classified into signature-based and anomaly-based systems. Signature-based IDSs focus on signatures of known malware, or malicious software. Anomaly-based IDSs focus on detecting deviations from expected behavior based on a learning algorithm applied during normal usage. In this work, we present a third classification of IDSs, which we call integrity-based IDSs. Integrity-based IDSs monitor a system to ensure that it does not deviate from a known good state.

We investigate how integrity-based IDSs can be used to build self-healing computer systems that automatically recover from known and unknown attacks, minimizing the damage that is done by the attack. We believe that integrity-based IDSs can have low false positive and false negative ratios that approach zero. Further, we show that layers in the system can be used to isolate the self-healing monitor from the production system in order to guarantee protection of the monitor without sacrificing significant performance.

Our approach is to build self-healing systems that are equipped with an intrusion recovery system (IRS). The intrusion recovery system consists of an integrity-based intrusion detection system, a *statehold* that contains the known good state, and a recovery mechanism. The goal of the intrusion recovery system is to identify known and unknown attacks and recover from them before significant damage occurs.

CHAPTER 1

INTRODUCTION

Computer systems on the Internet are targeted from a variety of attacks that can compromise their intended operation. Once the system has been compromised, it is difficult to re-establish trust in the system. Computer systems have network, data, storage, and processing resources that are valuable to an attacker. When an attacker gains access to these resources, his or her goal can be to covertly retain access. These resources can be used to distribute malicious network traffic, harvest sensitive information, store illegal or malicious content, and process malicious jobs. The most conventional method to recover from such a compromise is to wipe the system clean and reinstall from known good media. We investigate how an attacker can try to retain covert access, present a more robust system architecture, and investigate alternative methods of recovery based on our architecture. Our methods may be more more efficient, timely, and economical in many situations.

1.1 Motivation

Host computer security has followed an intuitive trend since the dawn of computing. At first, security concerns were not a priority when designing computer systems as there were few users and they could be trusted not to perform malicious activity. With respect to security, physical security was the biggest concern. As computer systems became more networked, more people gained access to various systems, and so trust became more important. During this time, significant malicious activity increased. At first, most of the activity was benign in motivation; however, the activity was inadvertently costly to organizations (e.g. worms). At present, more emphasis has been put on security. Systems have been deployed with more advanced security mechanisms configured with complex security policies in order to prevent unauthorized use of computer resources. However, it is often easily possible to bypass complex security mechanisms and policies, especially if they are not configured properly or

if vulnerabilities exists.

The first viruses and worms sparked a significant increase in computer security awareness and interest. An important research outcome during this time was intrusion detection systems (IDSs), which is a system that can detect when a computer has been compromised [28]. Presently, IDSs have increased to widespread use at both the host and network level. Some work has been done on intrusion prevention systems (IPS) that try to prevent unknown attacks [60, 26]. Most related to our work is intrusion tolerant systems (ITS), which argue that attacks are inevitable and systems should be designed to tolerate attacks [53]. For the foreseeable future, compromised systems will continue to be a problem. Our work focuses on how to recover from an attack once it has occurred. We term our system an intrusion recovery system (IRS) because it is able to detect attacks and recover from them. Although we focus on recovery from rootkits, a specific type of malicious attack, the broader scope of our work entails a model that can recover from many types of attacks.

1.2 Contributions

Our work has studied building self-healing computer systems that can recover from attacks. We have specifically focused our work on recovery from rootkits. With this focus, we have developed a trustworthy computer system architecture suitable for commodity operating systems. We have made many contributions that relate to self-healing, intrusion detection, secure architectures, and rootkits. Below is a list of the major contributions of this work:

- *Self-Healing Systems* – We have investigated systems that are able to detect and recover from attacks. We have shown that it is possible to build systems that can recover from known and unknown attacks.

- *Rootkit Classification* – We have classified rootkits into user-level, kernel-level, device-level, and other-level. Most publicly known rootkits can be classified as either user-level or kernel-level rootkits. We have also developed a classification for rootkits that will target devices and rootkits that target other abstractions in future systems.

- *Integrity* – Most previous IDS work has focused on signature or anomaly-based approaches. We have shown that IDSs can also use an integrity-based approach in which a compromise can be discovered by detecting that the integrity of the system is no longer intact.

- *Virtual Layer Architecture* – We have developed a layered architecture that is suitable for intrusion detection and recovery. We have used this architecture in our design of a self-healing computer system.

- *Dynamic Control Flow Graph* – We have presented a new method for dynamically tracing the control flow graph of a process. This method can be used to detect deviations from accepted paths of execution.

- *Prototype Implementation and Testing* – We have implemented a prototype of the designed system. We have also tested our prototype to verify the effectiveness of our design.

We have tested our methods against a test suite of 10 user-level and 10 kernel-level rootkits. The system is able to recover from all rootkit installations within minutes after installation. The prototype intrusion recovery system incurs a performance penalty of about 30% worse performance than a normal system.

1.3 Outline

Chapter 2 provides background information on the compromise problem including a definition of some important terms. Additionally, an overview of key related research is discussed.

Chapter 3 provides a survey of current rootkits that are known to exist and some discussion of future types of rootkit attacks. This discussion of existing and future rootkits enables an understanding of how to recover from the installation of rootkits.

Chapter 4 provides a discussion of methods for recovery from rootkits. These methods form the basis for our intrusion recovery system.

Chapter 5 introduces a model for an intrusion recovery system. A virtual layered architecture is presented that is more robust than widespread commodity operating system

architectures. We provide a discussion of each layer and present a detailed study of the layer that resides just below the production system.

Chapter 6 presents an evaluation of the model by testing a prototype against a representative suite of attacks. Although the prototype is not optimized, some performance evaluation of the system is investigated, so that a trade-off between system performance loss and increased reliability can be accessed.

Finally, Chapter 7 presents our conclusions and avenues of future work.

CHAPTER 2

BACKGROUND

Computer systems today are very complex and highly accessible to attackers. In the early days of the Internet, it was common for an attacker to take over a computer system in order to achieve fame among his or her peers. However, on today's Internet and for the foreseeable future, attacker's are often motivated by economical, political, or religious motives. Below is a list of possible uses of computer systems based on these motives.

- Information dispersion — An attacker may wish to disperse large volumes of information. The most notable case today is known as *spam* in which unsolicited email is sent out to numerous addresses in order to reach a large audience for some reason such as to advertise a product.

- Illegal content sharing — An attacker may wish to share illegal content. Such content may include copyrighted software, movies, and music. It is desirable to take over another computer for the purpose of sharing this content as the attacker does not want his own identity known.

- Information harvesting — An attacker may wish to obtain information from the user of the system. This information may include banking information, social security numbers, classified information, and so on.

- Denial of service — An attacker may wish to stop service to a computer or group of computers. There are various means to deny service to a computer system. One of the most widely used methods is a distributed denial of service using large clusters of systems to overwhelm a system.

- Selling resources — An attacker may want to take over a large number of systems, commonly referred to as a botnet today, for the purpose of selling the resources offered

by those systems.

In addition to damages incurred from an attack, recovering a system after an attack can be costly. Once an attacker has taken over a system, it is difficult to trust the system. The attacker may have tampered with the integrity of the system. Traditionally, the best method to recover from an attack is to wipe the system clean and reinstall. We believe that alternative methods of recovery should be investigated. Specifically, there are two important elements of recovery:

1. Checking integrity — In order to recover from an attack, it must be possible to realize that an attack has occurred. There must be a means of checking the current integrity of the system for correctness. An important part of integrity checking is to ensure that the mechanism that checks the system's integrity is not compromised itself. Furthermore, if the integrity is breached and must be restored, it is important to check the integrity after restoration to ensure that the system state is once again intact.

2. Restoring integrity — In the event that an attack has occurred, there must be a method to recover from the attack and to restore the integrity of the system. An important part of restoring integrity is that a copy of the known good state must be kept isolated from the attacker.

In this work, we assume an attacker will tamper with the integrity of the system. This is not true for every attack. Consider a social engineering attack in which the attacker is able to gain a legitimate user's password. The attacker may log into the system, retrieve information, and leave without modifying the integrity of the system. These type of attacks are important classes of attacks; however, we do not address these attacks in this work.

The installation of a rootkit will tamper with the integrity of the system. The purpose of a rootkit is to retain access to a system. Often, an attacker will install a rootkit onto a system that he or she has compromised. A rootkit will not allow an attacker to gain unauthorized access to a system that he has not already gained access to by some other means. An important observation of a rootkit is that it will hide the attacker's activities by altering the system.

In this work, we focus on recovery from rootkit installations. In particular, we explore automated methods of recovery. Below is a description of the notation used in our definitions. Following the notation, we provide a short detail of our computer system model and definitions of important terms. Finally, we conclude this chapter with a discussion of related work.

2.1 Notation

In this work we propose an architecture in which the production system runs inside a virtual machine. A system monitor runs in an isolated machine. A virtual machine monitor (VMM) is used to isolate the monitor from the production machine. We define some notation in terms of computer state.

Specifically, we can describe the integrity of the system in terms of state. State can be either mostly *static* or *dynamic*. Static state is state that should not change frequently. Dynamic state is state that should change frequently. For instance, the core kernel code should be static whereas data structures that the kernel uses are dynamic. We distinguish between static and dynamic state with subscripts *1* and *2* respectively. For example, Γ_1 represents static state of the known good state and Γ_2 represents the dynamic state of the known good state. Further, a subscript of k represents kernel-level state and a subscript of u represents user-level state.

- ω represents the current state of the physical system including the virtual machine monitor (VMM) and guest system in a virtual environment.
- σ represents the current state of the operating system or the guest system in a virtual environment. This state would be the same as ω in a non-virtual environment.
- λ represents state that is isolated from σ, yet processes existing in this state have visibility and control over σ.
- Γ represents the set of known good states for the system. In the virtual environment, this only refers to the state associated with the guest operating system and not the VMM state.

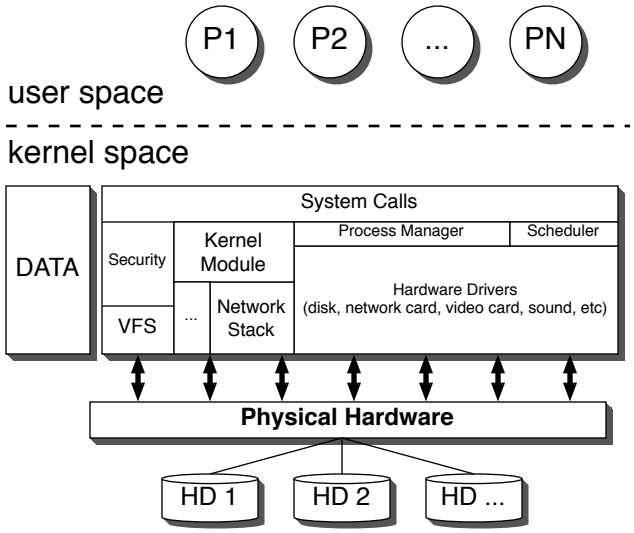


Figure 1: Model of computer system

- α represents state that is produced by an attacker's activities.
- ρ represents state associated with an attacker's rootkit.
- Δ represents a difference between two rootkits.

2.2 Computer System Model

We model the computer system as shown in Figure 1. There are two modes of operation: kernel-level and user-level. The kernel and modules that run at the kernel-level have complete access to the machine. Kernel-level code, existing in state σ_k, has direct access to the hardware and visibility of all of kernel space and all of user space. User-level code, existing in state σ_u, has direct access to its own state but depends on the kernel for access to other state, such as the hardware, based on security mechanisms and policies.

2.3 Terms

2.3.1 Integrity

Integrity means that the state of the system is intact and has not been tampered with in a malicious manner. Integrity, denoted i, is defined to be any subset of the known good state. Integrity is defined to be

$$i = \sigma : \sigma \in \Gamma \tag{1}$$

or in expanded form

$$i = \begin{cases} \sigma_{k_1} : \sigma_{k_1} \in \Gamma_{k_1} \\ \sigma_{k_2} : \sigma_{k_2} \in \Gamma_{k_2} \\ \sigma_{u_1} : \sigma_{u_1} \in \Gamma_{u_1} \\ \sigma_{u_2} : \sigma_{u_2} \in \Gamma_{u_2} \end{cases} . \tag{2}$$

An important assumption for defining integrity is that the range of known good state is known. For static states, one can reason that this is feasible. However, the range of known good dynamic states, such as σ_{k_2}, it is more difficult to describe the range of known good states. We propose some methods to monitor static and dynamic state.

2.3.2 Compromised System

When an attacker has gained some level of unauthorized access on a computer system, the system is said to be *compromised*. If the attacker gains unrestricted access, known as *root access*, the compromise is considered a *root-level compromise*. With root-level privileges, the attacker can change any state within the system. For instance, the attacker can modify the system so that the original reporting processes no longer reports accurate information. We can define a compromised system, denoted as c, in terms of the known good state as

$$c = \sigma : \sigma \notin \Gamma \tag{3}$$

or in expanded form

$$c = \begin{cases} \sigma_{k_1} : \sigma_{k_1} \notin \Gamma_{k_1} \\ or \\ \sigma_{k_2} : \sigma_{k_2} \notin \Gamma_{k_2} \\ or \\ \sigma_{u_1} : \sigma_{u_1} \notin \Gamma_{u_1} \\ or \\ \sigma_{u_2} : \sigma_{u_2} \notin \Gamma_{u_2} \end{cases} . \tag{4}$$

So, if the current state of the system is not in the set of known good states, then the system is said to be compromised such that $c \neq i$. This is only a partial definition because the reverse is not necessarily true. If the current state of the system is i, it is not certain that the machine is not compromised as it is possible that the integrity of the system has not been altered. In this work, however, we are interested in the class of systems where the integrity of the system has been tampered with such that $\sigma \notin \Gamma$.

2.3.3 Rootkit

Once an attacker has gained unauthorized access to a system, he or she will often use a *rootkit* as a tool to covertly retain access to that system. A *rootkit* is a set of utilities that allow the attacker to retain access and hide activities, which will diminish trust in the system.

We classify rootkits into *user-level* and *kernel-level* rootkits. A user-level rootkit will alter operating system tools that run in user mode (e.g. modify system binaries such as */bin/login*). A kernel-level rootkit will modify the kernel (e.g. system calls). A rootkit is said to be the state ρ that compromises the integrity of the system. If the system is compromised with a rootkit, then the following is a description of the state of the system:

$$c = i \wedge \rho \wedge \alpha. \tag{5}$$

However, an ideal rootkit will hide itself so that if the system reads its own state, it will read it as being in state i such that i is defined as follows:

$$i = i \vee (\rho \wedge 0) \vee (\alpha \wedge 0) = i. \tag{6}$$

2.3.4 Trust

We use Charles Pfleeger's and Shari Pfleeger's definition of trust with respect to computer security [52]. Their definition says that a system is trusted if we have confidence that it provides the following four services in a consistent and effective manner:

- memory protection
- file protection
- general object access control
- user authentication

We are interested in maintaining trust in the system. If a compromise occurs, then none of the above services can be guaranteed until recovery action has occurred. Our intrusion recovery system must be able to re-establish trust in the system.

2.3.5 Virtual Machine

A virtual machine is an machine abstraction on top of a physical machine. A VMM runs on the physical machine and partitions the resources of the physical machine into one or more guest machines [57, 54]. Each guest machine is designed to be completely isolated from the physical machine and other guest machines. Examples of virtual machine implementation include VMWare, L4, Xen, and UML [15, 8, 18, 14]. Virtual machines enable state λ that is isolated from the guest system state σ but has complete visibility of σ.

2.3.6 Honeynet

A honeypot is a network resource whose value lies in illicit use of that resource, and a honeynet is a network of honeypots. Honeynets are used to study the tools, tactics, and motives of attackers [58]. We use a honeynet to capture real-world compromises in order to help test our recovery techniques [46].

2.3.7 Self-Healing System

A self-healing system is a system that is capable of automatically recovering from known and unknown attacks. An ideal self-healing system would be able to detect any attack, determine the full extent of the attack, undo any state changes caused by the attack, and prevent further similar attacks. A self-healing system should be able to detect that the system is in state c and restore it to state i.

2.4 Related Work

2.4.1 Intrusion Detection

Forrest et al. have studied anomalous behavior of systems based on a learning period. Their work focuses on user-level applications in which they monitor the sequences of system calls requested by an application [33, 32, 66, 34]. Their methods use a learning period to learn how a program is expected to execute and behave. They focus on user-level applications by monitoring sequences of systems calls that are part of the operating system and assume that the kernel will not be compromised. Some work has demonstrated mimicry techniques as a tactic used by attackers to counter the work of behavior models [65].

Kiriansky et al. describe a method of program shepherding in which control flow transfers are monitored by the operating system and compared against security policies [43]. Their method uses an interpreter, which interprets all flows of execution prior to executing and only permits execution if a policy allows. The translated acceptable paths of execution are cached in order to achieve better performance.

As opposed to tracking the flow of execution, Suh et al. present a method for tracking the information flow in a system [59]. Their method demonstrates how to build a security mechanism that can track the source of information flows and tag them as *spurious* based on security policies. The operating system can specify certain policies that prevent a set of specific operations from spurious information. They show how a simple policy can be implemented to prevent such attacks as buffer overflows.

Recent work has been done on static binary analysis [36, 31, 64] to build models of program behavior for intrusion detection. The statically derived model can be used to

restrict the programs behavior during run-time. Work in static analysis claims to have low false positive rates. However, some programs are designed so that static analysis is very hard in order to prevent reverse engineering, so static models for these programs will likely not be effective.

The main problems with intrusion detection are false positives and false negatives. Anomaly-based intrusion detection systems suffer from false positives and false negatives because the behavior of the system may not stay within the learned boundaries. Signature-based intrusion detection systems suffer from false positives and false negatives because the known bad signatures may not be complete.

2.4.2 Modeling and Understanding Rootkits

Thimbleby, Anderson, and Cairns developed a mathematical framework to model Trojan Horses (Trojans), and viruses [61]. A Trojan is similar to a rootkit in that both are designed to go unnoticed from the computer's legitimate users. Thimbleby et al. state that the detecting a Trojan is at least as difficult as determining whether or not two functions are equal; however this detection of Trojans is more difficult than our detection of rootkits because we assume that a good state is previously known.

Further, Thimbleby et al. discuss a virus that could infect a system querying program in such a way that the querying program itself would be unable to detect that it was infected. Such an infection is possible with kernel-level rootkits. When a kernel-level rootkit is installed, tools that check the kernel to see if a rootkit is installed are relying on the infected program itself. So, the kernel-level rootkit can return false information to the tools so that the tools report that the system is intact. This is why we propose a VM based architecture.

Other research has been conducted on developing a methodology for characterizing rootkits [45, 44]. The methodology to characterize rootkits involves determining the Δ between a baseline system and a system compromised with a user-level or kernel-level rootkit. The Δ is used to characterize rootkits based on checksums, number of files replaced, number of files added, user level verses kernel level, penetration into the kernel, and so forth. This

methodology can be used to determine if two rootkits are distinguishable. We build upon this methodology to develop a survey of rootkits.

The National Infrastructure Security Co-ordination Centre for the United Kingdom has published a report on Trojans and rootkits that discusses detection, remediation, and prevention of rootkits [16]. Their report describes rootkits as Remote Access Tools (RATs) that provide the attacker with a backdoor into the compromised system. The report discusses some of the functionality of RATs, which includes: monitoring system activities (i.e. watch users keystrokes and monitor users), monitor network traffic, use system resources, modify files, relay email (i.e. *spam*).

2.4.3 Detecting Rootkits

The open source and hacker communities have developed various tools to detect and prevent compromises, which include: *chkrootkit* [2], *kern_ check* [6], *CheckIDT* [40], and *Saint Michael* [13]. The chkrootkit tool is a script that checks systems for signs of rootkits. The chkrootkit script can detect many rootkits including both user-level rootkits and kernel-level rootkits, however some rootkits may evade detection because the chkrootkit script relies upon pattern matching to look for signs of known rootkits. The kern_check tool is used to detect kernel-level rootkits. The kern_check tool compares the addresses of system calls as defined in the *System.map* file, generated at kernel compile time, to the current addresses of system calls. The CheckIDT tool is a user-level program that can read and restore the interrupt descriptor table, of which the *0x80th* entry points to the system call handler. The tools that check the integrity of the kernel rely on the kernel and so may in fact not provide accurate results.

Saint Michael is a kernel module that monitors the ktext (kernel code in memory) for modifications and attempts to recover from any modification to running kernel code. If an attacker installs a certain class of kernel-level rootkits on a system equipped with Saint Michael, the system may be able to recover from the rootkit. After installation of the rootkit, Saint Michael will overwrite the rootkit modifications with the original trusted code. One drawback of Saint Michael is that it resides within the kernel and so is not isolated and

protected.

Other work has been conducted towards detecting compromises. Kim and Spafford show how a file system integrity checker, *tripwire*, can be used to monitor files for corruption, change, addition, and deletion [41]. In addition to other uses, tripwire can notify system administrators that system binaries have changed. Tripwire must establish a *baseline* for a known good file system. To establish a baseline, tripwire takes a hash (e.g. MD5, CRC, Snefru) of the files at a known good point. The baseline can be used for comparison at later points in time. A binary-level rootkit will replace system binaries, which will set off the "trip wire" and alert the administrator. However, a rootkit designer can counteract tripwire by breaking trust in the reporting tools upon which tripwire relies (e.g. redirect system calls).

Some other recent work has been conducted verifying the integrity of the system. Petroni et al. have designed Copilot, which consists of a PCI add-in card that is capable of scanning the hosts memory [39]. Their work focuses on detection of kernel-level rootkits and reporting any events to a monitoring station via an interface on the PCI card. They state that there are two categories of kernel-level rootkit detectors, which are those that check the consistency of output based on input and those that check kernel memory. Their card has the ability to check kernel memory from an isolated vantage point. They report a 1% performance penalty is incurred in order to detect a kernel-level rootkit within 30 seconds. The advantages of their approach include that they can achieve hard isolation with hardware, do not need to modify the operating system, and do not sacrifice much performance. The disadvantages of their approach include the need for specialized hardware, the lack of visibility inside the OS, and the lack of visibility of CPU registers.

2.4.4 Forensics

Another area of related work is in forensics in which methods are developed to analyze systems to determine how they were compromised. King and Chen implemented a framework for tracing the entry point of an intrusion called BackTracker [42]. BackTracker logs events for three operating system objects: processes, files, and filenames. Moreover, it keeps track of dependencies among those system events, which reflect how these events affect each

other. An administrator can manually trace back through the logs to determine the point of entry. After a compromise has been detected, e.g. by an altered file, a human investigator runs analysis tools offline on the logged data. The result is a graph of objects and their dependencies that can be backtracked from the detection point to the entry point. The logging added a 9% performance overhead and generated 1.2GB of data per day under an intensive workload [42]. One problem with this work that still needs to be solved is that a human analyst is needed in order to analyze an attack.

2.4.5 Recovery

Candea et al. have discussed the possibility of building systems that are designed to recover rather than building systems designed to never fail [24]. They term such computing recovery oriented computing (ROC). They present the following equation for availability in terms of mean-time-to-fail (MTTF) and mean-time-to-recover (MTTR):

$$Availability = \frac{MTTF}{MTTF + MTTR} \tag{7}$$

Given that systems will inevitably fail, they point out that the availability of systems could be increased if we reduce the MTTR. Two methods of recovery that the ROC group has introduced include microreboots [30] and system undo capability [22]. A microreboot restarts only portions of the system that have failed as opposed to restarting the whole system. System undo provides the ability to undo operations not only at the application level but for the entire system including the operating system.

2.4.6 Hypervisors

In early literature, much work on operating system architecture focused on performance, flexibility, and extensibility. Security has also been discussed mostly focused on safety. The exokernel was an operating system designed around building a small programmable machine on top of the hardware so that mechanisms provided by the machine could be exported and not policy [29]. These goals are close to what we desire for our architecture. The SPIN architecture provides a monolithic approach for flexibility [19]. SPIN is an interesting research project in that it forces security by the Modula-3 compiler, but a more complete

implementation is needed in order to test its practical applications. Although many have argued that a μ-kernel cannot compete with a monolithic kernel in terms of performance, Liedtke argues that such arguments may be based on results from improper implementations [48]. Open source implementations of Liedtke's proposed L4 architecture are available along with ported versions of the Linux kernel that run on top of the L4 μ-kernel [4, 9].

In Liedtke's μ-kernel work, he defines three concepts that a μ-kernel must provide. These concepts are address space, threads and interprocess communication, and unique identifiers. For address space, the μ-kernel should provide grant, map, and flush operations in order to achieve good flexibility. Interprocess communication has made fast μ-kernel implementations difficult, but Liedtke argues that fast interprocess communication can be achieved.

Although virtual machines were originally researched as early as the 1960's [37], recent attention has again focused on virtual machines [63, 67, 54, 57]. Hardware manufacturers are beginning to support virtualization in hardware citing workload isolation, consolidation, and migration as reasons to virtualize [63]. Virtualization efforts on x86 based machines have proven to be difficult and provide low performance. Whitaker et al. have addresses performance issues with the concept of paravirtualization where virtual hardware differs from the underlying hardware [67]

Xen is a recent virtual machine monitor work, which uses the concept of paravirtualization [18]. Like traditional virtual machine monitors, Xen allows multiple commercial guest operating systems to run concurrently on a single server. Xen provides the hardware abstraction on which the operating system runs. This includes providing abstractions for memory management, CPU resources and device I/O. Since the x86 hardware was not designed to provide for this abstraction, Xen must provide the ability to deal with certain privileged instructions. This means that a minimal amount of coding is necessary to port the OS to the Xen environment. Application level binaries are not affected. Different OS instances are protected from each other and cannot affect Xen, which provides isolation. One of the important goals of Xen is to optimize performance. For instance, Xen exists in the top 64MB of every address space in order to avoid TLB flushes every time a process enters or returns from the hypervisor. Further, unlike traditional virtual machine monitors (VMMs), in order

to execute privileged instructions, processes issues hypercalls to the hypervisor, which is similar to a system call.

2.4.7 Security Architectures

With the advent of recent research in virtual machines, many have applied virtual machines to security. Litty suggest the use of a hypervisor for an intrusion detection system [49], and we build upon the idea for our work. Arbaugh et al. demonstrate how a system can be booted in a secure and reliable manner through the use of cryptography hash checks for each layer from the BIOS until the system is operational [17]. This approach builds uses integrity chaining, assuming that the hardware is correct, to verify each layer in the system as it boots up.

Terra is a virtual machine based architecture that suggests applications should be run in different compartments so that they cannot tamper with each others resources [35]. Open-box virtual machines are available to run any OS similar to existing virtual machine (VM) architectures and allow full visibility into the system. Closed-box VMs allow no visibility into the instance by anyone who has not been given explicit access to do so. The main notion of Terra is towards compartmentalized security. The drawback of compartmentalized security is that the system is less flexible.

2.4.8 Self-Healing Software

Some work has been done on fault tolerant software that has the ability to cope with both programming bugs and malicious attacks. Locasto et al. introduce the concept of an Application Community, in which distributed homogeneous applications work together in order to detect zero day flaws or attacks [51]. Demsky and Rinard have developed methods to automatically repair data structures during error conditions [27]. Sidiroglou et al. have developed methods to gracefully cope with either malicious or benign software errors by using an instruction-level emulator.

Our work is similar to the self-healing software work in that we allow for some tolerance in the operating system. We combine a small and secure hypervisor with a host integrity-based intrusion detection system in order to build a more reliable computer. As opposed to

just detecting an intrusion, we also have developed methods to recover from an intrusion. Since we focus our work on rootkits, it is important to understand how to detect rootkits in order to recover from rootkits. Based on our understanding of rootkits, we are able to design a system that can recover from the installation of a rootkit.

CHAPTER 3

SURVEY OF ROOTKITS

Rootkits are a fairly recent phenomenon and have only been around since the early 1990's [38]. The term *root* stems from the traditional Unix user with complete access to a system or more generally the system administrator. The term *root* can be applied to other operating systems such as Microsoft Windows where *Administrator* access is the equivalent of *root*. The goal of a rootkit is to retain *root* access to a system in a covert manner so that the system administrator is unaware of illicit access and usage. The first rootkits used simple techniques to hide files and processes, but their technology has continued to increase in sophistication as time progresses and security professionals develop better methods to detect rootkits. In this chapter, we will give an overview of different methods that have been used by rootkits and also discuss some new methods we expect to see in the future from rootkits. We generalize our discussion to all modern operating systems but focus specific details on Linux based operating systems.

Rootkits are toolkits designed to replace operating system functionality so that some state in the system can be hidden from the computer system users. For example, a rootkit may replace a binary program *ls* with a malicious version of the program that hides files whose filename end with ".hideme". For our definition of a rootkit, we say that a rootkit *must* replace some code that is part of the operating system or state in the computer system.

Rootkits are related to Trojans in their design to deceive a user. Therefore, it is interesting to examine characteristics of Trojans when studying rootkits. A Trojan is a seemingly trustworthy execution path or executable that has a malicious side effect. A user could, for example, install a game believing that the game is not malicious, but the game installs a keystroke logger as a malicious side effect. Both Trojans and rootkits are designed to be undetectable to the user. Thimbleby et al. use a wide definition of Trojans that describes characteristics of Trojans that are similar to rootkits [61]. They categorize Trojans into

direct masquerades, *simple masquerades*, *slip masquerades*, and *environmental masquerades*. Here is a summary of their categorizations with examples of similar rootkit functionality as appropriate:

- Direct masquerades – These malicious programs pretend to be normal programs. For example, */bin/ls* could be replaced with a malicious */bin/ls* that does not list all files as specified.
- Simple masquerades – These malicious programs do not appear to be normal programs but instead appear to be a program that they are not. We do not classify this type of malware, or malicious software, as being part of a rootkit.
- Slip masquerades – These malicious programs pretend to be a possible normal program and exhibit characteristics similar to normal programs. For example, an attacker may create a program named *login2* in order to pretend to be related to the *login* program. We do not classify this type of malware as being part of a rootkit.
- Environmental masquerades – These malicious programs are indirectly invoked by the user. For example, the kernel is indirectly invoked by the user in order to bootup the system. The *sys_getdents* system call in the kernel could be replaced with a malicious *sys_getdents* system call that hides certain directory entries.

In our work, we are primarily interested in *direct masquerades* and *environmental masquerades*. These types of masquerades will modify programs that users directly and indirectly execute in order to hide activities of the attacker. In this chapter, we provide a survey of rootkits that use the techniques of direct masquerades and environmental masquerades. However, at the top level classification, we classify rootkits in terms of the execution space that they modify. We classify rootkits into user-level, kernel-level, device-level, other-level, and hybrid rootkits as seen in Figure 2.

3.1 User-Level Rootkit Definition

User-level rootkits are the earliest form of rootkits. A rootkit is considered a user-level rootkit if it modifies programs that operate in user mode. Many of the extended operating

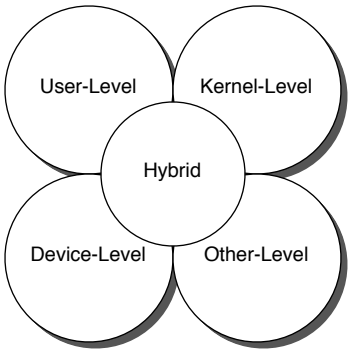


Figure 2: Classification of rootkits

system functions operate in user mode. In Unix or Linux based systems, for example, directory listings, process listings, and network connection listings can be seen from a terminal by executing user space programs such as *ls*, *ps*, and *netstat*. Many of the early rootkits were user-level rootkits that replaced user space programs. An attacker can store hidden files on a compromised system by replacing the *ls* program. A malicious *ls* program can easily be created that filters file listings based on filenames that match a pattern of the attacker's choosing. There are many user space programs that the attacker can target. Table 1 provides a list of common targets with descriptions based on Linux manual pages [10].

For many programs listed in Table 1, there is a direct relationship between the functionality of the program and a goal of an attacker designing a rootkit. For example, if an attacker would like to hide processes, then the attacker can replace the *ps* executable with a malicious copy that filters output based on preset rules. A more interesting program to modify is *ifconfig*. An attacker will often modify the *ifconfig* so that the user cannot determine that a local interface is in promiscuous mode. Often, an attacker may wish to set a network interface in promiscuous mode so that he or she can learn more information about nearby hosts. Another popular target has traditionally been the *login* program. An attacker can modify this program so that he or she can later enter the system with root-level privileges by typing a preset username and password. By understanding the goal of the attacker, it is easy to understand which files the attacker would like to target.

More knowledge is needed in order to understand why some of the utilities in table 1 are modified by rootkit developers. For instance, the *md5sum* is a target replaced by

Table 1: List of user-level rootkit program targets

Program File	Description
atd	Job executor that executes jobs scheduled by *at*
chattr	Change file attributes for ext2 file system
chfn	Change real username and information
chsh	Change login shell
dir	List directory contents
du	Estimate file space usage
find	Search for files in a directory hierarchy
ifconfig	Configure a network interface
init	First process in Linux that is the parent of all processes
in.telnetd	Implementation of telnet daemon
kill	Send a signal to a process
locate	List files in file system database that match a pattern
login	Begin session on the system
ls	List directory contents
lsof	List open files
md5sum	Generate or check MD5 message digests
netstat	Print network connections, routing tables, interface information, etc
ps	Report a snapshot of the current processes
pstree	Display a tree of processes
read	Read from a file descriptor
sshd	Server end of secure communication mechanism between two hosts
su	Change user ID or become super-user
syslogd	Linux system logging utilities
tcpd	Access control facility for Internet services
top	Display Linux tasks
write	Send a message to another user

an attacker. The reason rootkit developers began replacing the *md5sum* binary is because system administrators began using md5sum to verify the checksums of their system utilities. A system administrator could run *md5sum* on all system utilities just after installation and then check the integrity of those utilities at a later date, again using *md5sum*, by examining the current hash values. If a user-level rootkit had replaced utilities on the computer prior to the integrity check, then the system administrator would be able to detect the presence of the rootkit. Therefore, rootkit developers have become motivated to replace the *md5sum* binary as well. A malicious version of *md5sum* would report false hash values of utilities that had been replaced by the rootkit.

Table 2: List of kernel targets for a kernel-level rootkit

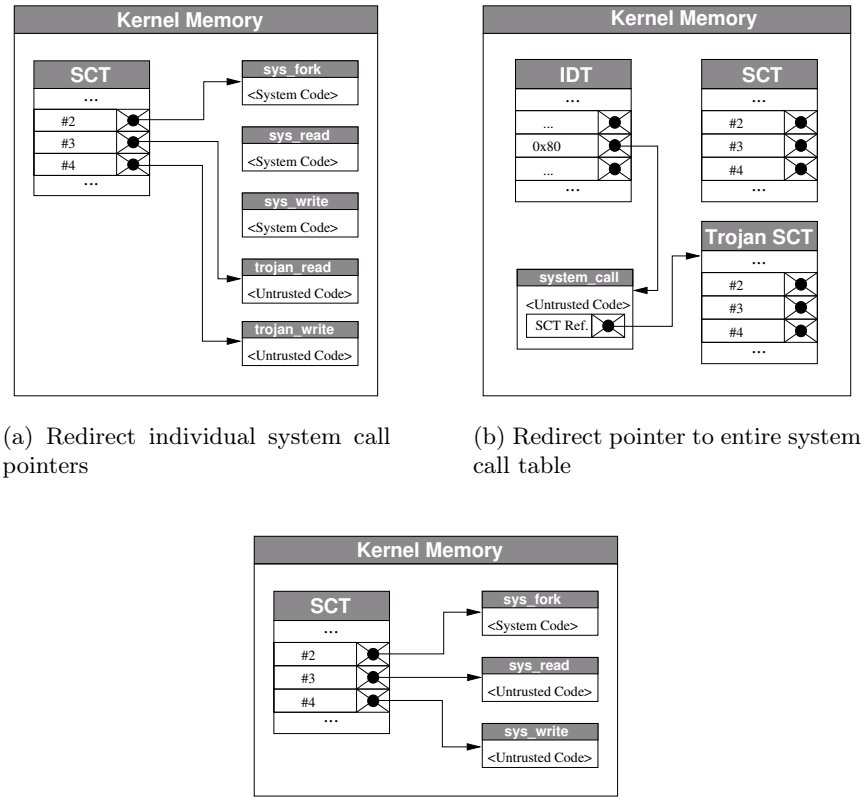
Kernel Subsystem	Description
System calls	Interface to system information for user processes
Virtual file system	Generic file system interface
Network stack	Network functionality
Scheduler	Mechanism to schedule processes
Kernel modules	Ability to insert new kernel functionality
Interrupt handlers	Handles hardware and software interrupts
Drivers	Controls the hardware in the computer system
Entire kernel	Entire kernel could be redirected
Kernel backdoor	An independent machine running in the kernel

3.2 Kernel-Level Rootkit Definition

Kernel-level rootkits are rootkits that modify the kernel. Many different structures and functions in the kernel can be redirected to malicious versions. Table 2 offers a list of kernel targets for kernel-level rootkits. As seen in the list, any kernel subsystem could be the target of a kernel-level rootkit. System calls have been the most widely used target due to their versatile nature. Interrupt handlers including the page fault handler have also been targeted. One interesting possibility that we have not seen is to redirect the entire kernel to a different kernel. Another interesting possibility is to insert a completely independent machine in the kernel that is accessible from a remote network connection. Using this method, the attacker could use the resources of the computer in his or her own machine and have complete access to the target system.

To better understand how a kernel-level rootkit can target the kernel and the level of flexibility offered, we have studied how the system call table can be targeted. Below are three possible ways to target the system call table:

- *Entry Redirection* — Redirects individual system calls within the system call table. Modifies original system call table.
- *Entry Overwrite* — Overwrites individual system call code. Does not modify original system call table.
- *Table Redirection* — Redirects the entire system call table. Does not modify original



(a) Redirect individual system call pointers

(b) Redirect pointer to entire system call table

(c) Overwrite individual system call code

Figure 3: Current rootkit methods to Trojan system call table

system call table.

Figure 3(a) shows how a kernel-level rootkit can redirect individual system calls within the system call table (SCT). The picture represents kernel memory after a kernel-level rootkit with *Entry Redirection* has been installed on the system. In Figure 3(a), the *sys_fork* system call is unmodified. Notice, however, that system calls number three and number four point to Trojan system calls. The trusted *sys_read* and *sys_write* are still resident in memory, but there are no references to them. The system call table now points to *trojan_read* and *trojan_write*. Any binary executable that relies upon the system calls *sys_read* and *sys_write* will receive untrusted information from the Trojaned system calls.

Figure 3(c) represents kernel memory after a rootkit with *Entry Overwrite* has been installed. Again, the *sys_fork* system call is unaltered. Notice, however, that the two

system calls *sys_ read* and *sys_ write* have been overwritten. The actual code for the system calls has been overwritten as opposed to the corresponding table entry that references the system calls. The system call table itself is unaltered with this type of rootkit. We have not studied an implementation of this type of rootkit. The advantage of this type of rootkit is that a program such as *kern_ check* [6] would not be able to detect the presence of the rootkit as *kern_ check* only checks the system call table, but that is only a short-lived advantage as new tools are developed.

Figure 3(b) represents kernel memory after a rootkit with *Table Redirection* has been installed. The picture depicts kernel memory for the i386 architecture and the Linux kernel. Within the Linux kernel code exists a table called the Interrupt Descriptor Table (IDT) that points to kernel handlers for each interrupt. The *0x80th* vector is a software interrupt that points to the system call table. All user processes invoke a software interrupt *0x80* in order to issue a system call [20]. When software interrupt *0x80* is invoked, the interrupt handler for interrupt *0x80* is called, which is the system call handler. The system call handler takes arguments from a user-space process and invokes the requested system call. The system call handler contains a reference to the system call table, which is used to look up requested system calls. This reference can be changed in order to redirect the entire system call table.

As Figure 3(b) shows, the entire system call table has been redirected to a *Trojan* system call table. The Trojan system call table usually contains many of the same entries as the original system call table but with a few key system calls replaced with Trojan system calls. We have not shown how the Trojan system call table points to system calls in Figure 3(b) as it is similar to Figure 3(a).

3.2.1 Sample Rootkits that Trojan the System Call Table

Table 3 shows a sample listing of kernel-level rootkits that we have classified in terms of their characteristics. We show three rootkits that use *Entry Redirection* to Trojan the system call table. The *heroin* rootkit is one of the earliest known kernel-level rootkits and is simply a kernel module that redirects a few key system calls. The *knark* and *adore* rootkits are other module based rootkits that redirect system call table entries.

Table 3: Sample classification of kernel-level rootkits

Rootkit	**Modification**
heroin	Entry Redirection
knark	Entry Redirection
adore	Entry Redirection
sucKIT	Table Redirection
zk	Table Redirection
r.tgz	Table Redirection

The second group of rootkits listed are *sucKIT*, *zk*, and *r.tgz*. These rootkits all use table redirection and access kernel memory through the */dev/kmem* file. The *sucKIT* rootkit appears to be one of the pioneering rootkits for *Table Redirection*. The *r.tgz* rootkit was captured on the Georgia Tech Honeynet [46].

We have not seen any kernel-level rootkits that use *Table Redirection* and are also kernel modules. Similarly, we have not seen any kernel-level rootkits that target the kernel from user space and also use *Entry Redirection*. We believe that future kernel-level rootkits may redirect the software interrupt handler or the entire interrupt descriptor table, but have not seen any rootkits to date that use this technique. Finally, we have not studied any rootkits that use *Entry Overwrite* to Trojan system calls.

3.3 Device-Level Rootkit Definition

A highly specialized rootkit could target a device in the computer system. Many devices today contain their own CPU and reprogrammable memory. For example, redundant array of independent disks (RAID) controllers, network processors, and video cards are all devices that can have reprogrammable memory. The software that runs the device can be updated or changed to meet different user needs. The ability to reprogram the device for an update or other non-malicious change also enables an attacker to reprogram the device with a rootkit version of the device code. For instance, an attacker could reprogram a hard disk controller with software that returns false information to the operating system about the files on the hard disk. We call rootkits that target devices *device-level* rootkits.

Table 4 shows a list of possible devices that a rootkit could target. The table shows

Table 4: List of device-level rootkit targets

Device	Description
Basic Input/Output System	Controller that boots up the system
Hard Disk Controller	Controller that mediates access to a hard disk
RAID Controller	Controller that mediates access to many hard disks
Network Card	Controller that mediates access to a network
Video Card	Controller that mediates access to a video monitor
Sound Card	Controller that mediates access to a sound device
Other Controllers	Other Controllers
Scanner	Independent device that scans images
Digital Camera	Independent device that takes pictures
Mobile Phone	Independent device used for communication
Personal Digital Assistant	Independent device used to carry information
Other Peripherals	Other independent devices

two groupings of devices. The first group of devices focus on devices that are part of the computer system and cannot function independently. The second group of devices focus on devices that can function independently of the computer and usually communicate over a serial bus or some other external bus. This distinction is not rigid because some RAID Controllers have been designed to function independently from a computer. However, the listing demonstrates that different types of devices can be connected to the computer that are vulnerable to a rootkit attack.

The basic input/output system (BIOS) is a particularly concerning target of a rootkit. If the BIOS has been replaced with a malicious version, then even a pristine operating system could not be trusted. For example, an attacker could program the BIOS to bootup the pristine operating system as normal but also instruct the computer to download malicious software from the Internet and execute that software in parallel unbeknownst to the system user. The network card is another example of a concerning target. Some network cards contain multiple processors and a significant amount of RAM. If a rootkit developer has access to such a network card, control over the network card could be easily established and then perhaps control over the entire computer.

Other independent devices could also be the target of a rootkit. All of these devices contain programs that interact with the operating system. So, if a rootkit is installed on

one of these devices, then potentially the attacker could retain access to the computer system by way of the device.

One of the important observations of device-level rootkits is that the rootkit will still exist in the computer system even if the system administrator reformats the computer and reinstalls the main operating system. The ability for device-level rootkits to survive a complete reinstallation makes them particularly dangerous.

3.4 Other-Level Rootkit Definition

Other-level rootkits target a part of the system that is not considered to be at the user-level, kernel-level, and device-level. We define this type of classification of rootkits in order to consider future versions of rootkits. For example, in a system with a virtual machine monitor (VMM), it is conceivable that a rootkit could be installed in the VMM. In general, a rootkit can target any software abstractions or other state in the system that enables the attacker to hide his or her presence.

3.5 Hybrid Rootkit Definition

Based on our survey of rootkits we have developed four classifications of rootkits: user-level, kernel-level, device-level, and other-level. Each rootkit classification is distinctive. However, it is also possible that one rootkit contains elements of two or more different levels. A rootkit could, for example, contain elements of both a user-level rootkit and a kernel-level rootkit. We call these type of rootkits *hybrid* rootkits.

We have captured an example of a hybrid rootkit on the Georgia Tech Honeynet [5]. On June 1, 2003, a honeypot on the Georgia Tech Honeynet was compromised by an attacker [46]. The honeypot was running Red Hat Linux 6.2. The attacker gained access to the machine through a vulnerability in the wu-ftpd2.6.0(1) ftp daemon. Subsequently, the attacker installed a rootkit called *r.tgz* onto the honeypot. The rootkit contained elements of both a kernel-level rootkit and a user-level rootkit.

3.6 Methods of Rootkit Installation

In order to install a rootkit, an attacker must have *root* or administrative privileges on the system. Usually, the attacker will gain root privileges on the system by exploiting a vulnerability or use some other unauthorized means such as social engineering. Depending on the type of rootkit and the system the rootkit is installed on, there are different methods to install a rootkit for user-level, kernel-level, device-level, and other-level rootkits.

To install the rootkit, it must first be downloaded onto the target system. Any standard file transfer protocol can be used to download the rootkit. After installing the rootkit, the downloaded files are usually deleted from the system. Also, any traces of entry into the system are deleted from the system logs. In essence, the attacker would like to remove all indications that the system has been compromised and is now under the direct control of the attacker. In practice, the skill of the attacker as compared to the skill of the system administrator determines how well the attacker can stay hidden.

3.6.1 User-level Rootkit Installation

User-level rootkits can be installed by binary replacement with precompiled binaries, binary replacement with source code, or by using the system's package management software. However, the third case is purely speculative. The installation can be done manually or through an automated process as part of the payload of a worm or virus. When using binary replacement without source code, the attacker downloads precompiled binaries and either runs a script or manually copies the precompiled binaries over the existing binaries.

The attacker may wish to make a hidden copy of the target binary before replacing it with a malicious binary. The reason to make a copy is so that integrity checks of the malicious binary can be redirected to a hidden copy of the original binary so that the system seems intact. The rootkit must include functionality to hide the original binary and redirect checks to the original binary.

An attacker may wish to install a rootkit on the system using the source code and compiling the code on the target. This is often the case when the attacker does not have precompiled binaries for a given system. It is possible that the attacker writes a custom

rootkit for the target system and does development on that system. The only distinction between installing a rootkit using binary replacement with source code versus binary replacement with precompiled binaries is that the attacker first compiles the source code. After the source code is compiled, the installation process is the same.

As a third method to install a user-level rootkit, an attacker could use a system package for the rootkit and install it. Many systems have package or software management tools that manage the programs that are installed on the system. We have not verified that attackers use the system package software, but in theory, it could be an installation method seen in the future. In order to use the package management software, an attacker could develop a rootkit package and install it or alternatively create a new version of an existing package and update the existing package.

3.6.2 Kernel-Level Rootkit Installation

Kernel-level rootkits can be installed either persistently or non-persistently. In order to install a kernel-level rootkit persistently, the attacker needs to install the rootkit in the running kernel and modify files on disk to ensure the rootkit is active after a reboot. To install a kernel-level rootkit non-persistently, the attacker only needs to install the rootkit in the running kernel.

There are different methods to install a kernel-level rootkit into the running kernel depending on the system. If the system does not support changes to the running kernel and there are no vulnerabilities in the kernel that would allow it to be changed, then the attacker cannot install a kernel-level rootkit into the running kernel. Many kernels support loading and unloading of kernel modules, which enables one method to install a kernel-level rootkit. An attacker can create a kernel module that, when loaded, changes the functionality of the kernel. To change entries in the system call table, for example, the kernel module would contain replacement entries for the system call table and, upon being loaded into memory, would change the kernel's system call table entries.

Similar to user-level rootkits, kernel modules can be either copied to the target system or compiled on the target system. It is more likely necessary to compile a kernel module than

to compile a user program. The reason is that kernels vary more from system to system, but user programs are less varying. A kernel module is heavily dependent on the running kernel.

Another method of installation that exists on Linux based system up through Linux 2.6.13 is to install the rootkit through a device file such as */dev/kmem*. The */dev/kmem* file is a special file that gives the root user read and write access to the kernel memory through file I/O primitives. The virtual memory of the kernel is addressable through *seek* operations. Using these primitives, an attacker can locate kernel data structures, such as the system call table or virtual file system structures, and modify those data structures to point to malicious code. In order to allocate new memory in the kernel, it is possible to point an unused system call to the kernel memory allocator function *kmalloc*. Then, a user process can issue a system call *kmalloc* to allocate kernel memory and then copy over malicious code. Using these basic techniques, an attacker can install many different types of kernel-level rootkits onto the target system. These specific techniques will not work on all system but other similar techniques may exist depending on the system implementation. More generally, methods to install a kernel-level rootkit include:

- *DMA* — These type of kernel-level rootkits could patch running kernel code with malicious code by programming an attached hardware device to use direct memory access (DMA) to modify kernel code. The concept was introduced in [56], but we have not seen any implementations.

- *Swapped-out Pages* — With root-level access, the attacker may have raw access to attached hard disks. Memory pages are swapped to the hard disk when memory becomes full. An attacker could use raw hard disk I/O to modify swapped out pages in order to target the kernel. Normally the kernel code is never swapped to the disk, but an attacker could use indirect means to target the kernel through swapped out pages.

- *Local Image* — The kernel image resides as a binary file on the file system. The attacker can modify the kernel image on disk and replace trusted code with Trojaned

code. The next time the system is rebooted, the Trojaned kernel image will be loaded into memory, thus accomplishing the attacker's goal without modifying the running kernel.

- *Distributed Image* — The beginning of the chain of trust starts at the source code and binary distributors. An attacker could compromise a kernel image before it is ever installed on the system (i.e. replace code or binary files with Trojans before the kernel is distributed). As Thompson points out, one must "trust the people who wrote the software," or in this case trust the people who distribute the kernel [62]. This type of installation can be considering a Trojan horse type installation because the user would download the image from a seemingly trustworthy site.

In order to install a rootkit persistently, the attacker needs to modify or add a file to the hard disk. The attacker can replace a kernel module that is always loaded on bootup with a module that installs the rootkit. The attacker can add a malicious kernel module to be loaded at bootup. Alternatively on a Linux system, the attacker can modify a start-up script or program so that it issues a series of file I/O operations on the */dev/kmem* device file. There are many different ways to install a kernel-level rootkit and the exact details will depend on the target system.

3.6.3 Device-Level Rootkit Installation

We have not seen many device-level rootkits in use but predict that they will be a problem in the future. The method of installing a device-level rootkit is very rigid as opposed to installing a user-level or kernel-level rootkit. The firmware on most devices that can be upgraded must be upgraded by a special flash operation. The protocol for this operation is specific to the device but typically consists of copying the new firmware image into the executable memory of the device one byte at a time from start to finish. It is not usually possible to do a partial upgrade and any mistake in the firmware code can result in significant damage to the system. Repair may involve sending the device back to the manufacturer to reload the firmware.

Independent peripherals that connect to the computer may have either firmware or software that can be changed to include rootkit-like functionality. The method of installation of these type of rootkits is to follow the communication channel to the device and install the rootkit. One possibility, depending on the device, would be to issue a software malicious upgrade from the main computer to the peripheral device. The malicious upgrade would contain a rootkit.

Device-level rootkits are particularly hard to develop because each device is unique and there are many different devices. Furthermore, the specifications of the device may not be publicly available, so the attacker may have to reverse engineer the operations of the device in order to design a rootkit. In the future, if device specifications become more open, then driver development will become easier and so will device-level rootkit development.

3.6.4 Other-Level Rootkit Installation

Other-level rootkit installations will be specific to the new types of rootkits seen. In the case of a VMM rootkit, the attacker must have access to the VMM in order to install a rootkit. Depending on the type, it may even be necessary to have physical access to the system in order to install a rootkit. We focuses on remote installation of rootkits for user-level, kernel-level, and device-level methods of installation, but physical access for any of these types and the other-level type of rootkits enables different methods of installation. With physical access, for example, the hard disk of the computer could be removed and installed in a different computer in order to install the rootkit.

3.7 Methods of Detection

It is widely accepted that intrusion detection systems (IDS) fall into one of three categories: *signature*, *anomaly*, and *hybrid* systems as seen in Figure 4. Signature based IDSs have a database of known malicious patterns that match activity against the database to determine if the activity is malicious. Anomaly-based IDSs use a learning algorithm to learn what normal activity should look like and then enter a monitor mode in which they flag unusual activity as malicious. Hybrid IDSs combine signature and anomaly techniques.

Intrusion detection may not be completely accurate. It is possible to detect non-malicious

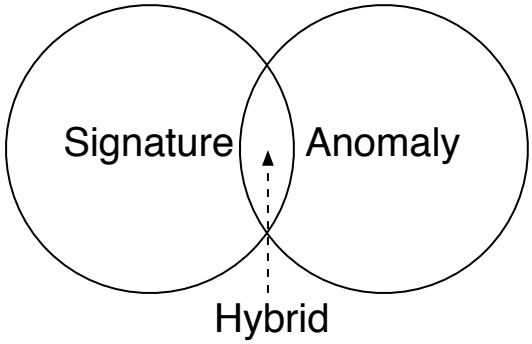


Figure 4: Classic intrusion detection systems

activity as malicious, which is known as a *false positive*. It is also possible to detect malicious activity as non-malicious, which is known as a *false negative*. The goal of an IDS designer is to have no false positives and no false negatives.

Detection techniques are also classified into network and host IDSs. A network IDS monitors network flows for signs of an intrusion or compromise of a node in the network. A host IDS monitors activity on the host for signs of an intrusion. Rootkits can be detected by IDSs. Typically, it is easier to detect a rootkit on the host, but it is important to ensure the rootkit has not disabled the functionality of the host-based IDS.

In addition to signature, anomaly, and hybrid IDSs, we propose a distinguishing type of intrusion detection called *integrity* based intrusion detection as seen in Figure 5. An integrity IDS monitors for an unauthorized change in state on a host. This type of IDS becomes particularly distinguishing when detecting rootkits. Consider that at a given time X, the system is noted to have a given state $s_1 = \sigma$. At a later time Y, an IDS can check to see if the current state $s_2 = \sigma$ is equal to s_1. If it is not, then an intrusion has occurred.

Perhaps an integrity IDS could be classified as a form of an anomalous IDS, but we would prefer to distinguish integrity IDSs from anomalous IDSs. The reason we make this distinction is that monitoring integrity is very rigid whereas monitoring for anomalous detection is not completely rigid. We believe that integrity IDSs are the complement of signature IDSs. A signature IDS monitors for known bad state and an integrity IDS monitors for known good state.

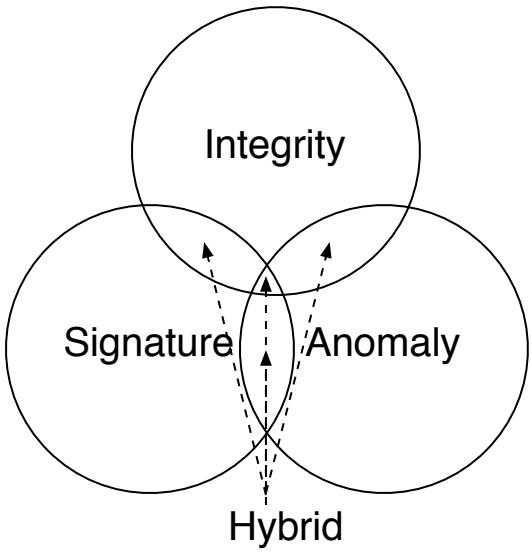


Figure 5: Integrity-based intrusion detection systems

3.7.1 Signature-Based Rootkit Detection

Given that many attackers will use the same rootkit, it is possible to use signature detection to detect that a rootkit has been installed. One of the most widely available tools that checks for signs of a rootkit is known as *chkrootkit* [2]. The chkrootkit tool is a script that performs a series of pattern matching on the current system for known patterns of rootkits. For example, if the file */etc/ttyhash* exists on the system, then the t0rn rootkit may be installed. Signature based rootkit detection can indicate the installation of a rootkit but may fail in the case of a polymorphic rootkit. Also, signature based IDSs can only detect known rootkits or rootkits that match a signature for a known rootkit.

3.7.2 Anomaly-Based Rootkit Detection

It is possible to detect that a rootkit has been installed on the system using an anomaly host-based IDS. One host-based anomaly IDS technique is to monitor the sequences of system calls for programs on the computer system. After a sufficient learning period, deviations from the normal sequences of system calls can indicate the installation of a rootkit. For example, if the *login* program suddenly begins issuing network related system calls then a rootkit may have been installed. Anomalous IDSs are helpful in detecting unknown rootkits.

However, it is possible to have false positives and false negatives with an anomalous IDS if the learning period is not sufficient. Further, the attacker may use a mimicry attack that mimics the expected behavior [65].

3.7.3 Integrity-Based Rootkit Detection

Integrity-based IDSs monitor the integrity of the system state. A snapshot of the state is stored at a known good time, which is called the *known good state*. It is possible to compute hash values of the known good state in order to save storage, although storage costs continue to decline. At a later point in time, the IDS can check to see if the current state of the system has changed from the latest known good state. If it has, then a rootkit may have been installed on the system.

Another way to check the integrity of the system is to verify the consistency of data structures in the system. For example, the kernel has many different data structures that keep track of the same objects. One data structure may be responsible for enumerating the currently running processes via system calls while a different data structure may be responsible for scheduling the processes for execution. If both data structures do not contain the same processes, then an inconsistent condition has been found and a rootkit may have been installed on the system. It is possible that a rootkit has removed a process from the list of enumerated processes but has left the process in the scheduling queue. This would allow the process to execute but it would not show up by the list of processes running.

Example tools that monitor the system integrity include *tripwire*, *aide*, *kern_ check*, and *samhain* [41, 1, 6, 12]. *Tripwire* and *aide* are tools that can monitor the integrity of the file system. *Kern_ check* and *samhain* are tools that can monitor the integrity of the kernel.

3.8 Methods of Recovery

If a rootkit is installed on a system, then the system can be wiped clean and reinstalled. This method will ensure that the rootkit has been removed from the system. It is important to ensure the entire system is reset to a known good state, which would include resetting all the device firmware in the case of a device-level rootkit. Presently, this method is the most widely accepted method to recover from a rootkit installation. We present new methods to

recover from a rootkit, in the following chapters, which is more efficient and equally robust in many situations.

3.9 Known Rootkits

There are many publicly known rootkits available on the Internet today. These publicly available rootkits are designed for many different systems including Linux, Solaris, and Microsoft Windows based systems. Appendix A offers a list of some of the currently used rootkits we have seen.

CHAPTER 4

METHODS OF RECOVERY

There are many key design principles and architecture considerations when building a system that can cope with a successful attack. An important part of recovery is detecting an intrusion. We believe that integrity-based intrusion detection is the best type of mechanism to incorporate into an attack recovery system because it is based on known good state. Known good state is a key concept that is necessary for recovery. Below is a discussion of design principles, key concepts for recovery, and methods of recovery when designing systems that can recover from a malicious attack.

4.1 Design Principles and Architecture Overview

We describe five design principles that are specific to building systems that can cope with malicious attacks. These design principles are *simplicity*, *isolation*, *trust*, *visibility*, and *adaptation* as described below:

1. Simplicity – The system must be designed so that security mechanisms and interfaces are as simple as possible. As the mechanisms grow in complexity, it is difficult to ensure that the functionality meets the specification.

2. Isolation – The security mechanisms must be guaranteed to be isolated from the system. If the system monitor is a part of the system, then a malicious attack could easily disable the monitor.

3. Trust – The design of a self-healing system must rely on a foundation of trust. If the software that is originally installed on a computer is not trustworthy to begin with, then the security mechanisms have no trustworthy baseline to monitor.

4. Visibility – The security mechanisms must have visibility of the entire system. If the monitor cannot interpret state in the machine, then it is difficult to determine

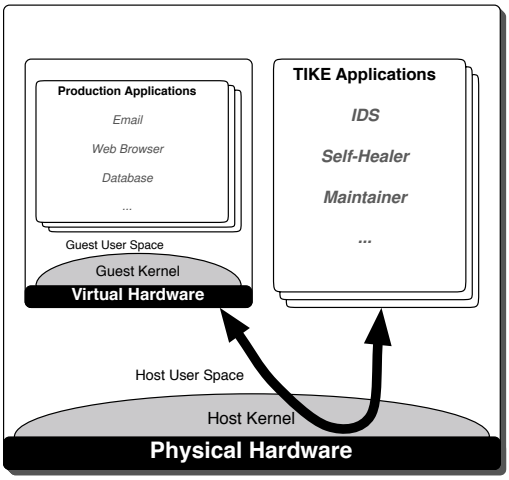


Figure 6: Overview of TIKE architecture

relationships of the state for intrusion detection and recovery.

5. Adaptation – The self-healing system must balance security with performance. If the system is completely secure but sacrifices performance to an unacceptable level, then there will be little value in the system.

Building on these five principles we describe a system architecture that is capable of autonomous self-repair. Figure 6 shows an overview of the self-healing architecture. In this architecture, we propose that the core security mechanisms are made possible by what we call a trusted immutable kernel extension (TIKE). The mechanisms are an intrusion detection mechanism, a self-healing mechanism, and a maintainer mechanism. These three mechanisms work together based on condition policy rules (CPR) in order to monitor for attacks on the system and self-heal the system if a successful attack has occurred.

4.1.1 System Operation

The self-healing mechanisms must be initialized before the computer system is brought online. First, the security mechanisms are installed on the computer system. Then, the production operating system is installed on the system. Next, the security mechanisms establishes a known good state baseline for the production system and initialize the various components including the scheduler, intrusion detection system, self-healer, and the maintainer. After the self-healing mechanisms have has been initialized, the production system

can be brought online. Any legitimate changes to the system will be corroborated with the maintainer, and any malicious changes should be undone and logged.

When power is turned on, the security mechanisms are first initialized. After the security mechanisms have been initialized, they verify the integrity of the production system and begin booting it. The security mechanisms continually monitor the integrity of the production system during the boot process and after it boots. The system should be repaired automatically if needed. The CPU cycles are scheduled between the production machines and the security mechanisms, which is controlled by the monitor.

The isolation of the security mechanisms is guaranteed by the virtual machine monitor (VMM). A virtual machine is used in order to simplify the isolation. The security mechanisms are isolated but retain visibility of the entire production machine. The security mechanisms can control the level of adaptation based on the current threat level.

4.1.2 Trusted Immutable Kernel Extension

The critical component of a self-healing system is the TIKE. The repairing agent resides within the TIKE. The TIKE concept builds upon the design principles of *isolation*, *trust*, and *visibility*. The TIKE is an enabling architecture that serves as an isolated safe-haven for the security mechanisms.

The requirements of the TIKE are embedded in its name, a Trusted Immutable Kernel Extension. Below, we describe the three core requirements of the TIKE.

- *Trusted* – The TIKE must report accurate information about the state of a host, which means the information can be trusted to be true. Furthermore, the TIKE must execute exactly as instructed and do nothing else.

- *Immutable* – In order for the TIKE to be trusted, it must be immutable. If an attacker compromises a system, the attacker must not be able to compromise the TIKE. Further, the attacker must not be able to disable the TIKE's services.

- *Kernel Extension* – In order to monitor the entire state of the system, the TIKE must exist at the kernel level. In order to prevent attackers from compromising the TIKE

or disabling the TIKE, the kernel extension must be isolated from the guest operating system kernel.

Our approach in this work for the TIKE is a virtual machine approach. The TIKE virtual machine architecture consists of a VMM, a *guest* operating system, and a guest monitor. The VMM is considered the core element of trust that is immutable.

The monitor has visibility of the guest operating system. The guest operating system runs as a user space process under the control of the VMM and monitor and is considered the production system. The guest operating system is considered *untrusted* because an attacker may be able to gain access to the guest operating system. However, the attacker is unable to gain access to the VMM or monitor from the guest operating system, even if he has *root* access.

The assumption that the attacker is not able to gain access to the VMM or monitor from within the guest system is an important assumption. If this assumption is false, then the entire system collapses. The argument for this assumption is that it is much easier to build a small virtual machine monitor and guest monitor correctly than to build a full blown operating system correctly. The attacker only has a minimal interface to the VMM. The only way the VMM should be accessible is via the interface it provides to the guest operating system and with physical access to the machine.

4.1.3 Monitor Overview

Our approach to automatically repair compromised systems is a monitor that resides on the system. The monitor consists of a scheduler, intrusion detection system (IDS), self-healing mechanism, and a maintainer as seen in Figure 7.

4.1.4 Scheduler

Conceptually, the scheduler is a central mechanism of the monitor. It controls what runs on the CPU including the production system, which is not show in Figure 7. Priority of the production system, maintainer, IDS, and self-healer, is controlled by the current threat level as observed by the IDS. Therefore, the IDS is responsible for maintaining the adaptive

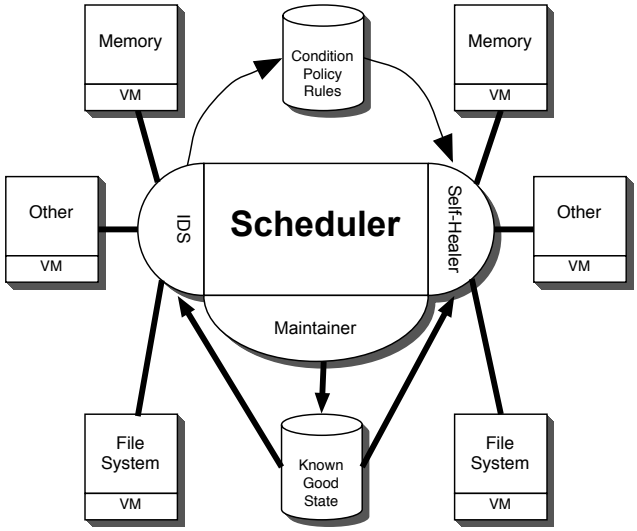


Figure 7: Monitor overview

nature of the monitor. If intrusions are detected, the system alert level is increased and increased priorities are given to the IDS, maintainer, and self-healer. As time progresses after an attack, the system alert level is slowly decreased. The production system only controls its own slice of the CPU and cannot steal cycles from the monitor.

4.1.5 IDS

The IDS is responsible for scanning the production system for a compromise. If a compromise has occurred, the IDS builds a report and sends it to the self-healer matching the compromise with the condition policy rules database. The condition policy rules database contains a list of policies describing actions that are carried out when a given condition is met. There are two major benefits to our host-based IDS approach. First, host-based IDS's have visibility of the entire system. Second, it has the ability to detect compromises even if the compromise bypasses all other security mechanisms. Not only is it possible to prevent many attacks, but it is also possible to detect successful attacks that bypass the preventative intrusion mechanisms.

Figure 8 shows a sample listings of condition policy rules. Each rule consists of four elements: *class* – which class of attack condition, *condition* – specific condition within the class, *policy* – a space separated list of actions to perform, and *adapt* – factor used to adjust

Key – *class : condition : policy : adapt*
file : changed : log replace_file : 5
file : added : log delete_file : 5
net : network_dos : log shutdown_network : 10
net : shell_code : log honeynet : 15
kernel : sct_mod : log replace_sct : 15
kernel : illegal_proc : log kill_process : 15

Figure 8: Sample condition policy rules

the adaptation level. The class and condition describe the exact details of the intrusion, such as the malicious replacement of a file.

The policy describes what action should be taken if the given condition occurs. For example, Figure 8 shows a policy of *log replace_file* if the condition *file changed* occurs. This policy means that the self-healer mechanism would *replace* the file on the guest operating system with a known good backup and *log* the action taken. The log is stored on the host system and is therefore protected.

The *adapt* number represents how much to increase the threat level. There are different models to use for the adaptive method. One model is to use a simple algorithm that sets the new threat level to the level defined by the condition policy rules. The system can have a default maximum scanning period set by a configuration file. If an intrusion occurs, then the corresponding *adapt* number is used to divide the period of scanning. In subsequent scans, if an intrusion is not detected, then the scanning period is incremented linearly back to the default maximum scanning period. If a second intrusion occurs after an initial intrusion, then the current scanning period is divided by the maximum *alert* number.

4.1.6 Self-Healing Mechanism

The self-healing mechanism is the component responsible for repairing a system that has been compromised. It operates on compromise events sent from the monitor and performs any action necessary to repair the compromise event based on the condition policy rules. The self-repairing mechanism is not scheduled unless there are outstanding compromise events that need to be serviced.

4.1.7 Maintainer

The maintainer is responsible for keeping a copy of the known good system state up to date. If the system state changes due to an upgrade, then the maintainer will update the corresponding known good state. The maintainer is a critical component of the system. It must authenticate system updates as legitimate updates. Root access to the production operating system and authentication access to the maintainer should be disjoint.

4.1.8 Adaptation and Performance

Under normal operations, the monitor tracks the system based on optimal timing intervals that will have minimum impact on system performance but maintain a given level of security. If an intrusion occurs, however, the threat level increases and the monitor will take more CPU cycles but performance will suffer. The system has two classes of monitoring levels that can vary on their taxing of the system depending on the exact operations performed. At the normal level, the system just monitors the system to prevent intrusions and detect if an intrusion has bypassed the preventative mechanisms. If an intrusion occurs, the system moves to the threat level. In the threat level, the system begins to heavily track information in the system such as which processes cause which states changes. Over time, the threat level goes back to normal.

4.2 Known Good State

The known good state is the set of all possible instantaneous sets that are considered to be good, denoted as Γ. Understanding the Γ for a system in an efficient and comprehensive manner is a key contribution of this work. The models and methods described in this work make a significant effort to achieve an understanding of Γ in a systematic way. Our results are not comprehensive to the problem, but provide a significant contribution focusing on automatic detection and removal of rootkits.

4.2.1 System Integrity

The system integrity, i, can be computed mathematically by testing the current state of the system to determine if it is a subset of Γ. If $i \subseteq \Gamma$, then the integrity of the system is intact.

Otherwise, $i \nsubseteq \Gamma$ and the integrity of the system is not intact.

The state of the system consists of CPU registers, memory, disk drives, device state, and other state. Ideally, an instantaneous snapshot of the entire state of the system could be computed and tested. Such a method to verify state could be instrumented in hardware; however, this may not be most efficient method to verify state in that it would be very costly.

Alternatively, we propose that all of the state can be verified incrementally over time. The state can be divided up into different sections called *buckets*. During a given amount of time, each of the n buckets b_k can verified so that the integrity of the system can be computed as:

$$i = \prod_{k=1}^{n} b_k \qquad \text{such that } b_k = \begin{cases} 0 & \text{if } b_k \nsubseteq \Gamma \\ 1 & \text{if } b_k \subseteq \Gamma \end{cases} . \tag{8}$$

If $i = 1$, then the integrity is intact. Otherwise, $i = 0$ and the integrity is not intact.

Three concepts that are important for maintaining integrity are *attestation*, *hashing*, and *re-attestation* as described below:

- *Attestation* – The software that is installed on the system must be trusted to perform as specified. In this work, we trust that the software vendors or developers can attest that their software is trustworthy. It is possible that the software contains programming errors that an attacker can take advantage of to gain access to the system. The IDS should be able to monitor the programs to determine if an attacker has exploited such programming bugs. Attestation focuses on the trust and Trojan horse problem in that the software vendor or developer attests that the software does not contain a Trojan horse.

- *Hashing* – A hash is a one-way function that can be computed on a state bucket that returns a small number representing the hash of that state bucket. Hashing is important in that it can be used to efficiently verify the integrity of state buckets. Additionally, hashing can be used with digital signatures from the software vendor

or developer in order to distribute software. The software vendor or developer can digitally sign the software in order to attest the integrity of the software.

- *Re-attestation* – The self-healing mechanism is responsible for re-attesting the integrity of the system after a compromise has occurred. If a rootkit is installed on the system, then the state of system cannot be attested. The self-healing system must restore the latest known good state and re-attest the integrity of the system. Trust is an important concept for a system that has been compromised. Conventional wisdom states that once a system has been compromised, it can no longer be trusted. However, the self-healing mechanism can re-attest the integrity of the system, which will re-establish trust in the system.

The concept of a self-healing system relies upon these three concepts. Attestation and hashing can be used for intrusion detection. Re-attestation is used for recovery from a successful intrusion.

4.2.2 Root Access

One of the critical elements of our design is that even if an attacker has gained root level privileges on the guest system, he will not have access to disable the monitor. Many have recognized the ability to separate control of the virtual machine from control of the physical machine. For example, Chen and Noble note that in the context of virtual machines even if the entire guest operating system is replaced, the attacker cannot disable or alter logging information on the host operating system [25].

4.2.3 Originating Entry Point and Patch

When a system is compromised, there must be some vector into the system that allowed the compromise. This vector could be a vulnerability in the security mechanisms on the system, or the attacker may have tricked a system user or administrator into granting access. If the system is successfully compromised, then the self-healing system should be able to detect that a compromise has occurred and backtrack the source of the compromise. In the case of a system vulnerability, the self-healing system should be able to determine what component

of the system is vulnerable and shutdown that component until it is fixed. In the case of social engineering, the self-healing system should alert the user that his or her account has been compromised and require the user to update their authentication tokens.

A method to trace the originating entry point is to backtrack events that lead to a compromised state. King and Chen introduced a manual framework for backtracking intrusions called BackTracker in [42]. BackTracker logs events regarding three operating system objects: processes, files, and filenames. Moreover, it keeps track of dependencies among those system events, which reflect how these events affect each other. After a compromise has been detected e.g. by an altered file, a human investigator runs analysis tools offline on the logged data. The result is a graph of objects and their dependencies leading to the alteration of the file in question (the 'detection point'). The BackTracker tools are able to filter less relevant dependencies to make the graph more readable for humans.

It is appealing to integrate the backtracking analysis in an automated fashion into the monitor to automatically eliminate the vulnerability that has been used to compromise the system. This is compelling in order to avoid a re-compromise of the healed system. IDS data can be used to serve as a detection point for an observed compromise. We believe it is possible to use the detection point to backtrack to the point of the compromise. However, the details and full theoretical and practical solutions are left for future work.

One important point is that the elimination of a vulnerability, depending on how it is eliminated, could lead to collateral damage resulting in a denial of certain services the system is supposed to offer. For example, it might be necessary to shut down a vulnerable service until a patch can be updated. However, there may be other ways to deal with the vulnerability besides shutting down the service completely. The method for dealing with the vulnerability will be a trade off between security and offering services.

4.2.4 Denial of Service

If the attacker is aware that the monitor is on the system, then he or she may consider ways to use the adaptive nature of the monitor to cause denial of service on the host. This must be considered when building the monitor. The general approach we offer to this problem

is that it can be handled by the adaptive nature of the system. When the adaptive level increases, the monitor can become more aggressive in how it responds to system events. The monitor should eventually learn exactly what processes the attacker owns and kill those processes, thus removing the attackers ability to control any aspect of the system. These details are also left for future work.

4.2.5 Statehold

Secure storage is a requirement for maintaining a copy of the known good state for a self-healing system. The reasoning is deduced from understanding the case in which secure storage is not available. In such a case, the attacker can compromise the system and also gain access to the known good state data. With this access, the attacker can easily change state in the system and also change the corresponding state in the known good state data to match his or her changes. In this case, the integrity of the system would appear to be valid because the known good state data would match the current state of the system. Thus, the known good state must be stored in isolated storage that is not accessible from the production machine even with complete access to that system. We call the secure storage in our system the *statehold*, which is a stronghold that can store known good state.

4.3 User Recovery

User recovery focuses on recovering from compromises that occur in user space. User space includes user applications, user data, operating system applications, and operating system data. The state associated with user space includes the file system and memory for processes. Although allocation of this state is mediated through the kernel, user space applications are often given full control over the state. The operating system as a whole is responsible for maintaining the policies of the system, and in many system architectures, this policy is enforced by user space extensions of the operating system. User-level rootkits typically target files on the hard disk as this is the easiest target. Below we discuss the details of how to recover from known and unknown user-level rootkits and a discussion of the details of different user space challenges and recovery methods.

4.3.1 File system

In our approach, the file system is key for monitoring and recovering from user-level rootkits. The file system contains many different types of state organized into files that can be monitored. The types include both user and system files. Specific types of files include executable binaries, data files, database files, configuration files, password files, key chains, secret keys, user right files, temp files, and registry files. Some of these files are fairly easy to monitor and can be monitored based our our attestation and hashing approach. Other files require more concern. Table 5 provides a list of different files and their difficult in monitoring them for recovery. The executable file types, including libraries and modules, are the easiest types of files to monitor. They should not change unless a non-malicious software upgrade is performed. The more difficult types of files include data files, configuration files, communication files, and temporary files. These files are more difficult to monitor because of their dynamic nature.

Figure 9 shows a model of a dependency graph for the different types of user space state. The subscript of each node name denotes the verification difficulty, which is one of h, m, or l that maps to high, medium, or low respectively. In our model, the non-executable objects all depend on the executable objects for state manipulation. This dependency is shown with the solid arrows coming from the group of executable objects $execute_l$, $library_l$, and $module_l$. The contents of a file in any of the non-executable objects is controlled through the executable objects. Therefore, if we can verify the integrity of the executable objects, then the state changes made to the non-executable objects can also be verified. This observation is interesting because the executable objects may be easier to verify than the non-executable objects.

Verifying the executable objects is not a complete solution, however, as there is feedback into the flow of the executable objects from the non-executable objects. For example, a configuration file may be changed so that the executable objects now allow an unauthorized user access to the system. This feedback loop is shown with the dotted arrows that feedback into the executable objects. A key observation in our work is that a true rootkit must change an executable object in order to hide itself. Therefore, monitoring the integrity of

Table 5: Difficulty of measuring integrity of different types of files

Type	Description	Difficulty
Executable	Binary files that are executable. These may include operating system files and user applications.	low
Executable Library	Binary files that are executable files may depend on. These files contain code that may be executed.	low
Kernel Modules	Binary files that may be inserted into the kernel for execution.	low
Data	Binary or ASCII files that contain user data. These files are not independently executable.	high
Security	Binary or ASCII files that contain system security sensitive information such as passwords, keys, or permissions.	medium
Configuration	Binary or ASCII files that allow the user to configure an application or the system.	high
Log	Binary or ASCII files that log application activity or system activity.	medium
Communication	Binary or ASCII files used for communication between processes.	high
Temporary	Binary or ASCII files that are temporarily created by processes.	high

the executable objects in user space is sufficient for detecting rootkits. However, we also discuss some methods for recovering from attacks against the non-executable objects.

4.3.2 Executable Binaries

Executable binaries include operating system binaries such as *ps*, *ls*, and *netstat* and user installed applications such as a web browser, document editor, or an email client. Furthermore, executable binaries refers to dynamically loaded libraries that link to executables and any other type of executable object files such as kernel module files and kernel image files. Any executable binary can be the target of a rootkit developer.

In order to recover from the installation of a user-level rootkit, a copy of the known good binaries that is replaced must be available. This copy of the binaries should exist in the statehold outside the reach of an attacker. A rootkit can be detected by comparing the existing executable binaries to the latest known good state stored in the statehold.

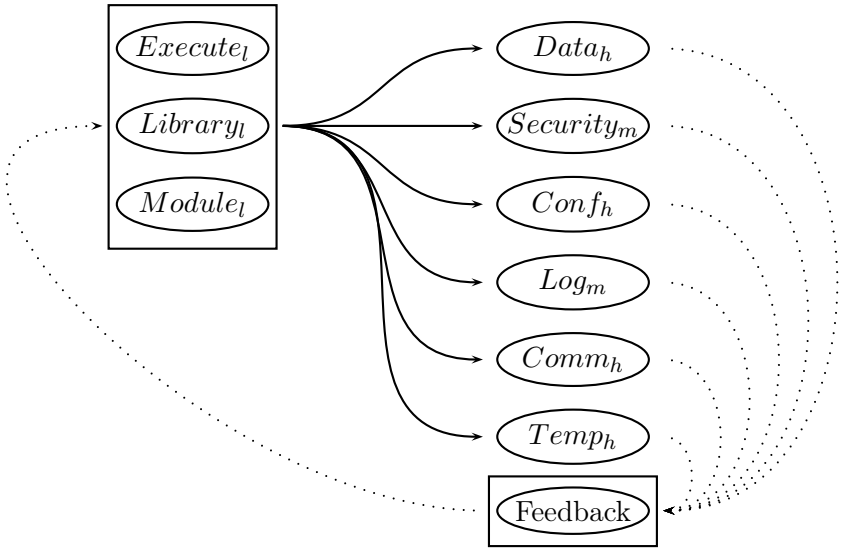


Figure 9: Dependency graph of user space state

In order to more efficiently monitor executable binaries, a hash of each binary is also stored in the statehold. To test if a given binary has changed without authorization, a hash value of the binary on the production system can be computed and compared against the hash in the statehold. A difference in hashes means that the executable binary has changed without authorization. Any authorized changes must be updated in the statehold.

4.3.3 Non-executable Files

Non-executable files include files that are both part of the operating system and part of user files. These files are more difficult to monitor than the executable files due to their dynamic nature. However, these files are the important files to protect. The user is more interested in protecting the privacy of his or her data files than his or her operating system files. Our approach in monitoring the user space state is to monitor the integrity of the executable objects, which control the integrity of the non-executable objects. There are some details that must be addressed, however, because our approach is not complete. Below are some important details for recovery from user space rootkits.

- *Configuration files* describe the execution parameters of programs. Programs read the contents of configuration files at initialization and at decision points, for example. Some parts of configuration files may control the security of a system. For instance, a web server configuration file may control what users can access the web server.

Further, configurations that are not directly related to security need to be secured. An important part of a secure system is that unauthorized users must not have access to change the configuration.

- *Security files* are files that contain security sensitive tokens such as passwords or encryption keys. These files are sensitive for two different reasons. The first reason is that an attacker could retrieve the passwords or keys needed to log in to the machine. The second reason is that the attacker could add or change the contents of these type of files to retain access.

- *Log files* report a log of activity seen on the computer. Since the attacker does not want to be seen, he or she will often delete the contents of log files. Furthermore, many rootkits will replace logging functionality in programs with filtered functionality. In this way, the activity of the attacker will not show up in the log files.

- *Data files* represent any type of data that the users of the system may wish to store. The data includes databases, documents, email, and media information. It is very important to maintain the integrity and privacy of these files. These files are the most important files to protect, but may also be the most difficult files to protect directly.

- *Other files* may include special files such as those used for communication or temporary files. These files are highly volatile and may be just an extension of user space applications. They are also very difficult to monitor, but they are not necessarily as sensitive as data files.

Based on these details, there are some important observations to make. One key observation is that if an attacker gains access to a computer system, then he or she may gain access to the security tokens such as passwords or keys that can grant future access. Therefore, if a successful attack occurs, then it is important to change the security tokens so that the attacker cannot regain access to the system. An alternative design could be to move the security tokens into the statehold. However, this decreases the flexibility of the operating system and increases complexity in the monitor.

Another key observation is that many types of state are highly dynamic, and tracking the integrity of the highly dynamic state can be very difficult. However, many files do not change very often or they may only grow in size. Log files, for instance, should always grow. So, an anomaly-based IDS could alert the user if a log file suddenly shrinks in size. Many complexities exist that make developing such a system difficult in practical terms.

Instead, the best theoretical approach may be to track all state changes and roll back any state changes that are the result of an attacker. This method also quickly reaches high levels of complexity because a legitimate state change may occur that depends on the state change made by an attacker. Some work has been done in this area [50]. We leave the methods and results of these problems for future work but recognize that these results are needed in order to realize a complete self-healing system.

4.3.4 User Space Processes

In addition to the file system, some memory is also allocated to user space. This memory makes up the user space processes, which include operating system tasks that run in user space and user tasks. A theoretical approach to recover from compromises that attack a running process is to track all state changes for all processes on the system. Deviations from known good state changes would be stopped or immediately recovered from by rolling back the state change. This approach does not scale in practical systems.

Many different host-based IDSs address the issue of user space processes. Some work focuses on anomalies of system call traces [33, 32]. If a user space process begins executing system calls in a sequence that deviates from the expected sequences, then the IDS takes some form of action in response to a possible attack. These IDSs rely on a learning period during normal operation. Other work has focused on the buffer overflow problem [26]. For example, a canary can be placed on the stack in order to make it more difficult for an attacker to exploit a buffer overflow [26]. These different types of host-based intrusion detection systems offer ways of detecting attacks on the user processes. These methods can be coupled with recovery actions that terminate the process upon malicious detection and restart the process.

Securing a user space process can be thought of as a simple form of securing a kernel. In this respect, many of our discussions of recovering from kernel attacks can also be applied to user space processes. When comparing methods used in the kernel to methods that can be used in user space, there are a few distinctions to note. There are many different types of user space processes. Some may be similar to the kernel in terms of flexibility and complexity, while others may be very simple to secure. Also, there are typically many user space applications running at one time with different dependency relationships. Abstractly, there are similarities that can be applied, but future work is needed to further clarify the distinctions.

4.4 Kernel Recovery

In order for the production system to sustain continuous operation under the supervision of the monitor, state in the memory must be tracked. The two main sections of memory are the kernel space and the user space. The kernel space includes such things as a list of running processes, the system call table, the interrupt descriptor table, file caches, modules, core kernel code, and other structures.

4.4.1 Text

Text refers to the executable machine code that is loaded when the kernel is first instantiated. Typically, a boot loader will load the kernel from a hard disk or other medium at bootup. The text is the section of the kernel image that is executable. After being loaded at bootup, the text of the kernel should not change.

Ideally, the memory that contains the text of the kernel would be marked as read/execute only. With such an enforced policy, it would not be possible to change the text of the kernel, so recovery of kernel text would not be necessary. However, there may be situations where it is desirable to have read/write/execute text. These policies are more flexible, which is one of the goals of many modern commodity operating systems.

In order to monitor the text that can be altered, a known good copy of the text must be stored in the statehold. The text can be divided into buckets of size n, for which a hash value of each bucket can be computed. To check a bucket, the monitor computes a hash of

the current bucket and checks that hash with the known good hash. If the check shows no change, then no action is needed. If the check shows a change, then recovery is needed.

If a compromise is detected in the text of the kernel, then the text can be restored from the latest known good state. There are a few important considerations to understand. Incorrect restoration of the text can crash the system or in the worst case destroy data on the system. It is important to restore all of the kernel functionality at one time because relationships and dependencies between different subsystems may be altered by the installation of a kernel-level compromise. Another issue that arises is that detection may occur during the installation process, so it would be possible to detect only part of the compromise that has occurred. Detecting only part of the compromise and recovering from that part can lead to an unstable system.

In order to address these considerations, all processes related to the rootkit installation must be terminated upon detection. If a kernel-level rootkit is detected, then all computer activity must be ceased except for the recovery mechanisms that repair the kernel. The recovery mechanism must not only repair detected damage, but a full scan of the kernel must be completed. Further, if the process that initiated the damage is still active, it must be terminated to ensure that it will not continue its installation process after normal computer operation is resumed.

We assume that the relationships between different functionality in the kernel are not significantly changed by the installation of a kernel-level rootkit. By significant, we mean that restoring the text of the kernel will not be problematic. We make this assumption because many rootkits are designed to be as similar to the system as possible, which means that they will ideally not change the system into a state that is difficult to recover from. If the rootkit does make recovery without rebooting significantly difficult, then recovery by rebooting the system may be a better option. Future work must address these assumptions, however, because the nature of rootkits is to break assumptions.

4.4.2 Modules

Kernel modules increase flexibility in the kernel by allowing the system administrator to insert and remove functionality. Functionality enabled by kernel modules includes device drivers, file systems, application support, networking subsystems, and other functionality. Such flexibility allows different code to be inserted into the kernel without rebooting the system and loading a different kernel.

The functionality added by the kernel modules can be exploited by a kernel-level rootkit developer, so it is important to consider recovery methods. We model a kernel module as a small form of the kernel itself. Thus, the same techniques that are applied to the kernel can also be extended to kernel modules.

Kernel modules exist as object files on a hard disk. When a kernel module is inserted into the kernel part of its image is the *text* section, similar to the kernel. Thus, the text can be monitored in a similar fashion.

A distinction between monitoring the text of a module and the text of the kernel is that modules are much more flexible in that they can be inserted and removed. Therefore, it is important for the monitor to be able to adapt to different states in the system that would be consistent.

4.4.3 Data Structures

The data structures in the kernel may be just as important if not more important than the text of the kernel and modules. The data structures are used to support flexible code in the kernel. For example, the system call table is a data structure that determines what code is executed for each system call. Some data structures define the relationships between the executable code in the kernel.

There are different methods to monitor data structures. The simplest method is to define all the data structures and enumerate all possible values for those data structures. Then, the monitor could check the data structures at any point in time in order to test their integrity. This approach is feasible for some data structures such as the system call table because the system call table should not change often.

A second approach is to track all changes to the data structures. At the point that a compromise is detected, the monitor could backtrack to the point of the compromise. All changes that are the result of the compromise could be rolled back. This approach may be possible, but we have not studied it in much detail. We believe that the overhead would be significant.

Another approach is to learn the different acceptable values for data structures. This is similar to the first approach, except that it is more flexible and less rigid. We discuss this approach in more detail in Chapter 5.

4.5 Device Recovery

Devices include any type of hardware that is connected to the computer. The integrity of devices can affect the integrity of the computer, so it is important to consider them. Some devices are more critical than other devices. For example, the integrity of the basic input/output system (BIOS) is highly critical as it is responsible for system initialization. It is possible for the BIOS to load the system into a malicious state that would be difficult to recover from. We divide the state associated with devices into volatile state and persistent state and discuss the recovery considerations with each below.

4.5.1 Volatile State

Volatile state includes state that is temporary or volatile. Such state can include control registers or memory cells contained within the device. Similar to the traditional approach of reformatting a computer system, a device could be reset to known good state. Alternatively, the state could be tracked in a similar manner as described by the sections on user space and kernel space. However, tracking volatile state may introduce significant performance overhead.

4.5.2 Persistent State

Persistent state is the memory associated with the device that persists even after the computer system is rebooted. Many devices have flash memory or some other type of persistent memory that contains initialization firmware that can be upgraded. It is possible for a

rootkit developer to develop a device-level rootkit that replaces the existing flash. In order to recover from such an attack, a known good copy of the firmware can be maintained in the statehold. Integrity of the firmware can be monitored, and the integrity restoration process would be to restore the firmware to the latest known good state.

4.6 Other Recovery

Besides the user recovery, kernel recovery, and device recovery, other recovery refers to recovery of any other state associated with the system. This state may include CPU registers, abstract network state, and distributed state. Particularly, the distributed state may show signs that the machine is part of a distributed network of machines that has been compromised. We believe such recovery is possible, but distributed cooperation is necessary.

4.7 Trusted Computing Base

As many people have recognized, the security of a computer system relies on the trusted computing base. This realization is particularly important in our work because we assume that the attacker has almost full control over the system. If the attacker has complete or full control over the system, then the ability to detect the installation of a rootkit becomes much more difficult. Our approach to recovering from rootkits relies on a trusted computing base that supports the statehold. We believe that only minimal functionality should be included in the trusted computing base in order to support the statehold and a system monitor. This minimal functionality can increase the security of the system and still allow for a highly flexible system.

CHAPTER 5

INTRUSION RECOVERY SYSTEM

Building on the concept of an integrity-based intrusion detection system and methods of recovery discussed in Chapter 4, we propose the concept of an intrusion recovery system (IRS). An IRS is a system designed to detect unknown attacks and minimize the amount of damage caused by an attack by recovering from attack very quickly. The ability to detect an attack using an IDS is a critical part of the IRS.

IDSs have certain false positive and false negative ratios that determine their effectiveness in practical usage, and they are traditionally classified into signature-based and anomaly-based systems. Signature-based IDSs focuses on signatures of known malware. Anomaly-based IDSs focus on detecting deviations from expected behavior based on a learning algorithm. Because an IRS builds upon an IDS, the ratios of false positives and false negatives from the IDS are amplified in the IRS. In an IRS, the acceptable ratios of false positives and false negatives are near zero. Ideally, the ratios are zero.

In order to meet the needs of an IRS, we propose a third classification of IDSs, which we call integrity-based IDSs. Integrity-based IDSs watch a system to make sure that it does not deviate from a known good state. These type of IDSs can be considered the opposite of signature-based IDSs, which monitor for known bad state. Unlike signature-based systems, integrity-based IDSs monitor the system to ensure that the state of the system is known good.

In order to support an IRS, we present a system architecture called *spine* that is an extension of the TIKE architecture. We specifically study how *spine* supports detection and recovery from kernel-level and user-level rootkits. The architecture includes a trusted computing base for an intrusion recovery system (IRS). The spine architecture is a multi-tiered approach, relying on the integrity of a small μ-kernel based hypervisor for correctness at the base level. Spine vertebrae are positioned at each level in the system in order to

overcome the semantic gap in the understanding of system state.

We conduct a detailed study of layer V_2, the layer just below the production system, in order to investigate the advantages of the layered architecture. Specifically, we discuss the design and implementation of an execution tracker in layer V_2. The tracker relies upon a new model to track indirect branches within a virtual machine. The ability to track indirect branches in a guest kernel enables the detection of kernel-level rootkits because unauthorized execution paths can be identified.

A learning algorithm can learn the dynamic relationship between control flow graphs, which we call the dynamic control flow graph. A method is introduced to discover the dynamic control flow graph of a running guest kernel so that all possible paths of execution in the guest kernel can be tracked. No prior knowledge of the guest kernel is needed, and no modifications of the source code for the guest kernel are needed. Hash values of memory cells containing executable code can be computed in order to monitor the integrity of the executable code. After a learning period, indirect branches that deviate from expected execution paths can indicate the installation of a kernel-level rootkit.

Since we are using the layered architecture, a fast virtual machine monitor (VMM) controls the guest kernel and the monitor in order to minimize the performance loss while adding additional security. The VMM enables a monitor running in one virtual machine to control the guest operating system in another virtual machine. The IDS in the monitor of layer V_2 can be considered a hybrid between an integrity and an anomaly-based IDS. In addition to the detailed layer V_2 discussion, we also discuss the some details of the other layers below.

5.1 Architecture Design and Reasoning

A rootkit is designed to hide the state α, state associated with the attacker's activities, and the state ρ, state associated with the rootkit itself. Further, in a system with state σ, the rootkit will conceivably modify any state in σ in order to hide the state of α and ρ. It is noteworthy that α and ρ are subsets of σ. Given this arrangement, it is important to design an architecture that supports a state λ, which is isolated from state σ and has the capability

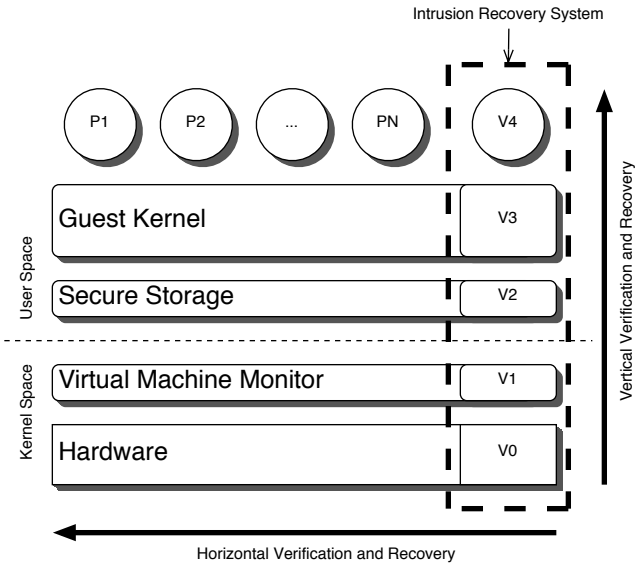


Figure 10: Overview of spine architecture

to verify the integrity of σ. This state is part of the trusted computing base. We believe that a small trusted computing base is the best architecture and that a hypervisor is a good solution to achieving a small trusted computing base.

Our trusted computing based is based on a μ-kernel, which can be considered a form of a virtual machine monitor. Figure 10 shows an overview of the architecture. The μ-kernel is the only component that runs in the kernel space or the privileged execution mode. It runs directly on the hardware providing a thin interface to the guest kernel. The guest kernel and all processes it supports, P1 through PN, run in user space at the unprivileged execution mode. On the right side of the figure is the vertically integrated IRS. We term this architecture the *spine* architecture because the components of the IRS, called the vertebrae, seep throughout the host as noted by V_0 through V_4 in the figure.

We divide the state in each layer into abstract state and detailed state. Abstract state is state that can easily be interpreted from a lower layer. Detailed state is state that is difficult to interpret from a lower layer. Each layer is responsible for monitoring the abstract state for the layer above it and the detailed state for its layer.

5.1.1 Fast and Secure Virtual Machine Monitor

Three architectures that are closely related are hypervisors, VMMs, and μ-kernels. A *hypervisor* is a more general term for a small layer that runs below the operating system, directly on the hardware, providing a method for running multiple operating systems independently and isolated from one another. A *virtual machine monitor* (VMM) is a layer that runs below an operating system that mediates privileged hardware instructions, which is a form of a hypervisor. Also, a *μ-kernel* is a specific form of hypervisor that consists of a minimal kernel that exports all system services to user space tasks, but guarantees a few minimal requirements. We base our work on the L4 Fiasco μ-kernel, the L4Env, and L4Linux [4, 7, 9]. The combination of these three software components is similar enough to a virtual machine architecture that we generalize our discussion to the virtual machine realm but discuss some distinctions.

Work in computer security usually relies upon some form of a trusted computing base. The kernel in commodity operating systems could be relied upon for a trusted computing base. However, in the case of flexible systems that allow the kernel to be modified, it is dangerous to assume that the kernel has not been compromised. This is demonstrated by the wide development of kernel-level rootkits. In order to help balance the needs of flexibility and security, we present a system architecture that relies upon a small VMM for security that still allows flexibility in the guest OS.

In order to prevent a rootkit from disabling the IRS, the IRS must be protected by partially isolating it from the guest OS. If an attacker gains full access to a guest OS, he or she does not have access outside of the virtual machine in which the guest OS resides. We believe that a VMM is simple enough that isolation between virtual machines can be guaranteed. In addition to providing security, the VMM must be fast enough to ensure the performance loss does not outweigh the added security. In order for the VMM to be secure, it must be well designed, very simple, and very carefully implemented. We believe that it is possible to build such a VMM, and for the purposes of a proof of concept prototype, we use one of many freely available existing VMMs called the L4 Fiasco μ-kernel in our work [4]. Recent publicly developed VMMs include different implementations of L4 and the Xen

VMM [18, 8, 48].

The VMM in our model needs to support a few performance critical design requirements. One of the key details for the VMM is interprocess communication (IPC) support. The VMM should support very fast IPC. Another important design requirement is the memory mapping requirement. The guest monitor needs access to the guest kernel's memory. More generally, context switches on today's hardware is very expensive, so the VMM needs to try and minimize context switches. In the future, more hardware support for virtual machines can drastically improve the performance of our monitoring system and could change the performance critical design requirements.

Litty discusses the use of a hypervisor for an IDS [49]. We extend this notion for our IRS system and further specify the requirements. First, we believe that the hypervisor should provide minimal mechanisms sufficient to guarantee isolation and not sacrifice significant performance. One hypervisor that meets these requirements is a μ-kernel, which is a form of a virtual machine monitor (VMM). The performance of μ-kernels has been of debate in past literature [19, 29, 48]. Liedtke discusses how μ-kernels can achieve good performance and that beliefs to the contrary are not necessarily true [48]. Further, Liedtke suggest three minimal requirements for a μ-kernel in [48] as described below.

- *Address Space:* The μ-kernel is responsible for managing address spaces. Three operations *grant*, *map*, and *flush* are described so that memory can be managed with good flexibility. The μ-kernel must enforce this management so as to protect its own address space; however, the μ-kernel can grant or map memory to a user space memory manager and flush access rights if necessary.

- *Threads and Interprocess Communication:* Threads are tied to address spaces, and so basic thread support must be handled by the μ-kernel. Further, cross-address-space communication must also be handled by the μ-kernel.

- *Unique Identifiers:* Each task must have a unique identification for efficient communication.

One of our assumptions is that V_0 and V_1 will not be compromised by the attacker. The

V_0 layer exists in hardware, which may be possible to verify as operating correctly [23]. It is more difficult to prove V_1 is immutable and correct; however, we have designed the system toward the goal of achieving this immutability from the perspective of the guest system. The reasoning of how our approach can reach this goal is deduced from code size, simplicity, and limited interface. The code size of the μ-kernel is small, on the order of 20,000 lines of code. Further, we believe the μ-kernel is as simple as possible while achieving reasonable performance and strict isolation. Finally, there is a small interface that the μ-kernel provides to tasks, which is on the order of 10 system calls.

Although we base our architecture on Liedtke's suggested minimal requirements for a μ-kernel, we add one addition requirement as described below.

- *Task Control:* Task control includes the ability to inspect and modify another task's control block, which includes CPU registers. Specifically, it is important to be able to inspect another task's program counter and the value of any registers that can be altered based on input from untrustworthy information flows. The V_1 layer should export this system call to the V_2 layer so that the V_2 layer can verify that the guest kernel is operating correctly.

5.1.2 Statehold

The *statehold* must be provided by a layer below V_3. In our work, layer V_2 provides the statehold for the IRS. This storage is used for storing a copy of the known good state, called Γ, for the entire guest system. For reasoning, consider this mechanism was not provided. Then, the Γ must be stored within the guest system as a subset of the state σ. Now, since Γ is a subset of σ, a rootkit would be able to hide itself by modifying Γ, what the IRS system considers to be known good state.

Each higher level in the IRS system requires read access to Γ in order to verify the integrity of its realm. V_2 can map Γ to the address space of itself, V_3, and V_4 in order to achieve this goal. V_2 must have access to an isolated storage disk in order to maintain persistent state and for large volumes of state such as a copy of the known good file system.

A significant portion of system state is not static. Therefore, it is important to address

a method for updating Γ. The important consideration to understand is how to authorize updates to Γ. There must not be a direct call to update Γ from within σ without authorization. Specifically, if an attacker gains root-level privileges within σ, he or she should not be able to update Γ. An independent method of authorization must be supported.

We suggest four different methods of authorization. The first method is to update σ from an independent machine in which updates are digitally signed and authenticated based on the public key of the system that is being updated. A second method is to require any update to σ to be performed locally at the machine based on a hardware authentication that grants authorization. The third method is to automatically update σ using digitally signed hash values of software upgrades. Finally, a fourth method would be a hybrid of any of the previously three methods. Studying these methods in detail to determine the best theoretical versus the best practical approach is left for future work.

It is conceivable that layer V_2 could verify the entire state σ. However, a *semantic gap* exists between V_2 and σ. For example, it is difficult to interpret guest kernel data structures (e.g. process tables) from the perspective of V_2, although possible. Instead of adding this complicated code to V_2, we believe a layered IRS approach should be used.

Each vertebrae in the IRS system understands how to verify state at its level. However, with this architecture, portions of the IRS system itself, namely V_3 and V_4, are vulnerable to attack from within the guest system. Therefore, each layer of the IRS system must be verified for integrity. We assume that V_0, V_1, and V_2 are intact. V_2 then must verify that V_3 is intact and V_3 must verify that V_4 is intact. V_3 is also responsible for verifying state of the guest kernel, which is difficult to interpret from V_2. For example, V_2 can easily verify that the guest kernel text is correct as this is well defined, but data structures need access to kernel functions in the guest kernel text for interpretation. Thus, V_2 must be able to verify the integrity of V_3 in addition to the guest kernel.

5.1.3 Layer V_4

The fifth level in the IRS is called *layer* V_4. Layer V_4 is responsible for:

- Verifying the integrity of the state in the system that cannot be easily interpreted by

V_3, V_2, V_1, or V_0, denoted ψ.

- Repairing ψ if necessary.
- Providing an interface to the user reporting the activity observed by the IRS.

In our architecture, we implement the file system integrity in layer V_4. The file system integrity could also be implemented in layer V_3. The benefits of implementing the file system integrity in layer V_4 is that the development process is much easier. Tools already exists such as aide and tripwire that implement file system integrity checks in layer V_4 [1, 41]. Theoretically, the entire system could be monitored from layer V_0; however, we believe that, practically, such a verification process would be very difficult.

5.1.4 Layer V_3

The fourth level in the IRS resides in the guest kernel and is called *layer V_3*. Layer V_3 is responsible for:

- Verifying integrity of state for V_4 and for the state in the guest kernel that V_2 cannot easily interpret, denoted ϕ.
- Repairing V_4 and ϕ if necessary.

The V_3 layer of the IRS exists as a part of the guest kernel. Examples of state ϕ include page tables, process tables, process state, and inserted modules all of which are not easily interpreted from layer V_2. Most of the hardware resources are given to the guest kernel and use of those resources are monitored by layer V_3; however, layers V_1 and V_2 maintain enough control over the hardware that recovery is feasible from malicious actions taken by the guest kernel.

5.1.5 Layer V_2

The third level in the IRS is called *layer V_2*. Layer V_2 resides just above the μ-kernel. Layer V_2 is responsible for

- Providing a read interface to secure storage for higher layers.

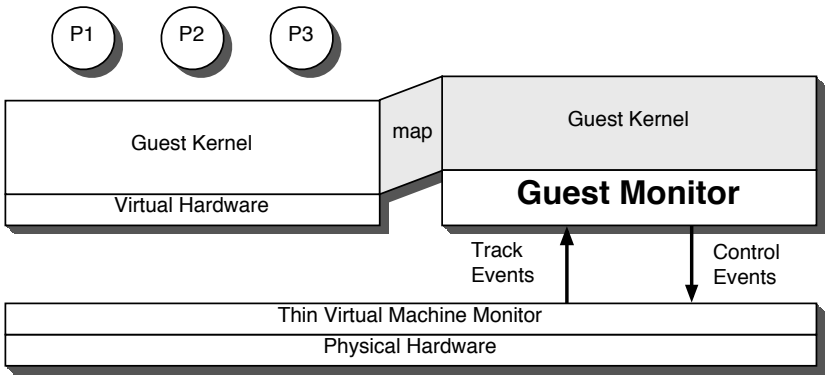


Figure 11: System architecture for dynamic branch tracking of guest kernel

- Verifying the integrity of the guest kernel and layer V_3.
- Verify integrity of statehold.
- Repairing the guest kernel and layer V_3 when the integrity is not correct.

In this work, we study layer V_2 in detail and propose a new method to verify the integrity of layer V_3. Figure 11 shows an overview of the detail for layer V_2. The figure shows the VMM as a thin layer that runs directly on the physical hardware, which comes from the overall architecture. The layer V_2 monitor is running in one virtual machine while the guest OS runs inside a separate virtual machine. The monitor is able to map the state associated with the guest kernel into its own machine; however, the guest OS has no visibility of the monitor or its virtual machine. The monitor receives tracking events from the VMM about the guest OS and can control the guest OS upon detection of malicious activity. Detection of malicious activity can be quickly recognized. Upon detection of malicious activity, recovery actions can be executed.

Our detail of layer V_2 relies on a hybrid of integrity and anomaly-based intrusion detection in which the integrity of the system is learned over time. The monitor in layer V_2 can be considered partially anomaly based because normal paths of execution are learned over time. The monitor can also be considered partially integrity based because the executing code is monitored to ensure the attacker has not tampered with its integrity.

When the system is first set up, the monitor in layer V_2 begins the learning phase. During this period, the monitor learns acceptable execution paths. It is vital to ensure a kernel-level rootkit is not installed during this initialization phase as we assume that all paths of execution are not malicious during the learning phase. After the learning phase, the monitor transitions to the monitor phase in which it can detect deviations from acceptable paths of execution and recover the system in the event that an unacceptable path of execution occurs. Further action can be taken upon rootkit detection based on security policies. Our policy in this work is to recover from the attack and log the event. In order to support flexibility, the monitor can support lessons at controlled points in time. During a lesson period, the monitor reenters the learning phase and learns new paths of acceptable execution. The authorization for a lesson period is the focus of future work.

5.1.6 Layer V_1

The second level in the IRS is called *layer* V_1. Layer V_1 is responsible for:

- Providing a process control interface to layer V_2.
- Verifying the integrity of layer V_2.
- Verifying the integrity of devices.
- Repairing V_2 if the integrity is compromised.

Layers V_2, V_1, and V_0 are part of the trusted computing base. In our current design, layer V_1 only has minimal ability to repair layer V_2 because it is assumed that V_2 cannot be compromised very easily. Checking can be conducted on layer V_2 at initialization and during run-time. Layer V_1 is responsible for verifying the integrity of layer V_2 and if possible repairing layer V_2 if the integrity is not intact. The statehold exists in layer V_2, so if V_2 has been compromised, then the best action may be to halt the machine and alert the system administrator. Layer V_1 should have this minimal capability.

Since all device changes are mediated by layer V_1, layer V_1 is responsible for verifying the integrity of devices. The known good state for devices is stored in the statehold, which

means that devices could potentially have access to the known good state. In order to ensure that the known good state for devices and any other state in the statehold is not tampered with, a public/private key pair stored in tamper-proof hardware can be used to digitally sign the state.

5.1.7 Layer V_0

The bottom layer in the IRS is called *layer* V_0. Layer V_0 is responsible for:

- Providing hardware support to guarantee isolation and enhance performance.
- Verify integrity of layer V_1.

The V_0 layer provides hardware support necessary to meet the isolation requirements of the system. Some have suggested that the hardware itself needs additional support to build secure systems [47]. We also believe that the hardware should provide integrity support. We specify that the hardware should provide integrity support for V_1, in the event that devices try to tamper with the integrity of V_1. Since the hardware is at layer V_0, there is no layer below the hardware that can provide verification of the hardware. Therefore, the hardware must be correct and trustworthy. The final specification of layer V_0 we describe is that it should provide a small tamper proof statehold to store a system key. This key can be used to verify integrity of higher layers and the integrity of the expanded statehold in layer V_2. We believe that more specific hardware enhancements would strengthen the design of our system, particularly from a performance perspective, and plan to explore such enhancements in future work.

5.1.8 Integrity

Integrity chaining consists of a root link that verifies the integrity of the next link, which verifies the integrity of the next link and so forth. As with any security, it is important to ensure that each link in the chain is secure. Otherwise, the weakest link can be exploited. There are two pieces of the algorithm we describe in order to verify the integrity of execution chains. Given links C_1 and C_2, where C_2 is a link that depends on C_1, C_1 must have a copy of the known good state for the code for C_2. Then, the first part of the algorithm is to verify

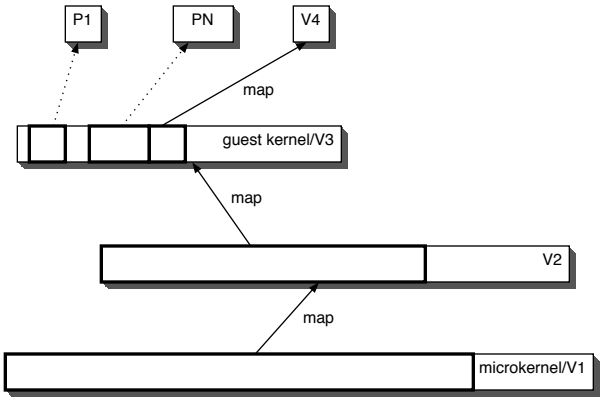


Figure 12: Memory hierarchy

the code for C_2 is intact and repair if necessary. The second part of the algorithm is to verify that C_2 is executing as expected. Our method for the second part of the verification is to periodically monitor the scheduler to ensure the instruction pointer for C_2 is executing.

In order to monitor and repair higher layers, each layer must have visibility of the layer it is monitoring. However, the reverse is not necessary and in fact we believe should not be allowed. A higher layer should have very limited visibility of a lower layer. Using Liedtke's model of granting, mapping, and flushing memory, we are able to achieve this visibility [48]. Figure 12 shows the memory hierarchy for the system. Just above the hardware layer V_0, the μ-kernel owns all memory. It maps a large portion of the memory to layer V_2, keeping some memory for data structures such as page tables. It is important that the memory distributions are mappings, as opposed to grants, in order to retain visibility. Next, layer V_2 uses part of main memory for the statehold and maps the rest of the memory to the guest kernel. The guest kernel in turn can in turn map portions of its memory, represented by the dotted lines pointing to P1 and PN, to processes. Note that the guest kernel maps a portion of its memory to the V_4 process. Thus, there is a chain of mappings from V_1 through V_4 so that the IRS has visibility over the entire system.

5.1.9 Learning

We assume that the integrity of the computer system is intact when the system is first booted and initialized. Based on this assumption, we propose the use of a learning algorithm after system initialization in which the IRS learns the acceptable states that determine the

integrity of the system. The learning algorithm should quickly learn acceptable states for the computer so that the initial learning period is short. In the case of systems that are widely deployed, a learning algorithm could be a part of the software development process in which a known good state file is distributed with the software. Upon installation into the system, the known good state file could be copied into the statehold assuming proper authorization is granted. It is critical that a rootkit is not installed in the system during the learning period.

The majority of the learning should occur at system initialization. The period can be controlled administratively, based on a certain period of time, or automatically based on probability that the user has extinguished all possible paths of execution. It is important to be able to alter the system at a later point. The learning algorithm must be able to learn legitimate system changes. The approach is to allow a lesson to be learned at a later period. This approach requires a form of authorization that must be secure. We do not specify the exact authorization mechanism in this work.

5.1.10 Hashing and State Buckets

The intrusion detection portion of our system relies on the ability to verify state. We considered two approaches for this capability. The first approach is hashing comparisons and the second approach is byte by byte comparisons. We consider the best approach of the two is to hash the current state and compare against a hash of the known good state. The reasoning relies on the way computer systems will be designed for foreseeable future. We believe that memory accesses will continue to be more costly than arithmetic instructions because arithmetic instructions do not have a high memory latency. Therefore, since the hashing method requires roughly half as many memory accesses, hashing is a much more efficient means to verify state. One risk is that an attacker could manipulate state such that the hash does not change; however, this is highly unlikely for good hashing schemes.

One approach to hash all of the state in the system is to divide the state into buckets. For a given state S, where any level L can be responsible for maintaining state S, L divides the state S up in to N buckets. L maintains an independent hash for each bucket. Then, for

any bucket in N, L can verify the integrity of that bucket. One of the benefits of comparing state byte by byte is that the exact state that has changed will be detected. When hashing the entire state S, only a binary result specifying the validity of the entire state is computed. However, using the bucket approach, benefits of both hashing and byte by byte comparison can be used. Using the bucket approach, the bucket that contains a state change can be quickly identified. Then, the known good state for that bucket can be compared against the existing state to determine exactly what state has changed.

There are a number of possibilities that arise when using the bucket algorithm to verify state. First, consider that one of the important goals of the system is to maintain high performance or minimize the CPU cycles consumed by the IRS. To achieve this goal, not all of the state in the system can be verified in one sweep as performance concerns, such as latency, would be harshly affected. By using the bucket algorithm, small sections of the state can be verified quickly and independently. This monitoring method enables an algorithm in which all buckets are independently and periodically checked over time. Further, in order to thwart an adversary, a random sequence of bucket checking can be conducted so that the adversary does not know which state will be checked next.

5.1.11 Adaptation Model

Under normal operations, the IRS should not tax the system very heavily. Most of the CPU cycles should go to the system. However, in the event of an attack, we believe that the IRS should receive more CPU cycles. While under attack, the most important objective is to recover from the attack. We also believe that the likelihood of detecting intrusions is significantly increased after the initial detection point. Based on these assumptions, we present the algorithm in Figure 13 for monitoring the state of the system.

For simplicity, the presented algorithm shows sequential checking of buckets and represents checking at each level in the system. Under normal operations, *sleep_time* is set to a reasonable rate so that performance is acceptable. However, in the event that an inconsistency is detected (bad_hash()), first the consistency is repaired, second *sleep_time* is set to a more adaptive level, and third the iteration count is reset. After increasing the alert level,

```
while(true)
{
   for(i = 0; i < num_buckets; i++)
   {
      // delay for 1/sleep_time
      delay(sleep_time);
      if(bad_hash(buckets[i])
      {
         repair_bucket(i);
         sleep_time = ADAPT_HIGH;
         count = 0;
      }
   }
   if(sleep_time > ADAPT_LOW)
   {
      if(count > BACKOFF_SCALE)
      {
         sleep_time--;
         count = 0;
      }
   }
   count++;
}
```

Figure 13: Adaptation algorithm

the system will remain at that level through BACKOFF_SCALE iterations of checking the entire state. In the event that more inconsistencies occur, the count of iterations will be reset to zero again. If no more inconsistencies occur, then the system will slowly back down to the low adaptive level after (number of alert levels)*BACKOFF_SCALE iterations.

We also consider a detailed adaptation for the layer V_2 monitor. One of the monitoring methods of layer V_2 is to track indirect calls in the guest kernel. In order to trade some CPU cycles for performance, it is possible to only monitor a subset of the indirect calls issued. A simple algorithm that maintains reasonable security and adds significant performance is to check each indirect call with a probability of p. As the value of p decreases, the chances of not detecting a compromise increases. However, an adaptive algorithm can be used so that in the event of a compromise, the system will increase the monitor's aggressiveness by increasing the probability p of tracking a given branch. Possible reasoning would be that if the system is attacked once, it will likely be attacked again in the near future.

5.2 Rootkit Detection and Recovery

5.2.1 User-Level Rootkits

User-level rootkits replace system binaries, add malicious utilities, change configuration files, delete files, or launch malicious processes. Repairing the damage done by a user-level rootkit is not difficult if the extent of the damage is understood. We believe that the extent of the damage can be understood if a copy of the known good state is available. The algorithm is then to compute the differences between the known good state and the current working state. For each difference, copy the known good state over the current working state. This yields pairings of the form action/reaction: <file replaced>/<restore original>, <utility added>/<remove utility>, <config change>/<restore config>, <file deleted>/<restore original>, <malicious process>/<kill process>, and so on.

5.2.2 Kernel-Level Rootkits

Abstractly, kernel-level rootkits are similar to user-level rootkits. They replace known good state with malicious state. However, the details of kernel-level rootkits are much more complicated than user-level rootkits. Kernel-level rootkits modify running kernel code, which can drastically effect the stability of the system. Previously seen rootkits will modify the system call table, virtual file system, and kernel data structures. Future rootkits will likely attack these vectors but will also conceivably hide their presence using other methods. For example, the page table data structures could be modified to redirect the entire kernel such that static checks against the previous kernel addresses would remain valid. For simple redirections, the same algorithm for repairing user-level rootkits can be applied to kernel-level rootkits where example pairings would be: <system call redirected>/<restore original>, <vfs redirected>/<restore original>, <malicious module inserted>/<remove module>, and so on. However, other complications exists with kernel-level rootkits. For instance, locking issues become important because it is necessary to make sure the kernel is not executing in memory that is being repaired. Also, memory allocated to a malicious redirection must be reclaimed. As for attacks such as page table redirections, these can be detected and repaired by the V_2 component. For more sophisticated structures, the V_3 component can periodically

perform consistency checking of the data structures . The extent of methods for recovery from all kernel attacks will be the results of future work. One important complication is highly dynamic structures.

5.2.3 Device-Level Rootkits

Device-level rootkits are also similar to user-level and kernel-level rootkits except they target state associated with devices instead of state associated with user or kernel space. The firmware in devices is the key target to attack because state changes in firmware will persist after a reboot and also after a reinstallation of the operating system and software applications. Thus, a known good copy of the firmware must be stored in order to recover from a device-level rootkit. One action/reaction pair for device-level rootkits is <firmware replaced>/<restore firmware>.

5.2.4 Other-Level Rootkits

Other types of rootkits may also exists in future systems depending on the types of systems that are designed in the future. For example, mobile devices, distributed systems, low-power devices, and other types of computing devices are likely to gain in popularity. To generalize rootkit detection and recovery from these other-level rootkits, we propose that the integrity of these devices can also be monitored. The general rootkit recovery pairing is <state changed to malicious state>/<restore state to previous known good state>.

5.3 Layer V_2 Detail: Dynamic Control Flow Graph Monitor

We have studied layer V_2 in detail and present a new method to detect kernel-level rootkits. When considering methods to detect and recover from kernel-level rootkits, one of the simplest approaches would be the classic reference monitor concept in which an image of the guest operating system (OS) runs in a virtual machine (VM). The virtual machine monitor (VMM) intercepts and monitors all relevant system calls and memory modifications. Any action which does not comply with a certain security policy causes the VMM to recover from the action or shut down the guest OS and alert the system administrator [55, 21]. This approach is clearly too inefficient since almost every machine instruction of the guest OS

needs to be checked.

A second method to detect kernel-level rootkits could be to design a form of load sharing in which a primary machine runs as a guest OS in the VM and a secondary machine observes and checks all or part of the system calls in parallel. The secondary monitoring machine can use memory mapping and tracking to observe state changes. It is still necessary to trap or interrupt the guest OS with many machine instructions, and so the overhead is still very high.

The third method is to not check every machine instruction but only those that are at special points in the execution path. By checking these points, an accurate and complete path of execution can be traced dynamically in the kernel. The secondary machine in layer V_2 can observe the run-time relationships of the control flow graphs by monitoring indirect branches in the guest kernel. After a sufficient learning period, the layer V_2 monitor can detect deviations from the expected relationships between the control flow graphs and recover from installations of malicious paths.

A *control flow graph* (CFG) is a graph that represents the possible paths of execution in a program from a given entry point to an exit point. More specifically, we refer to a CFG that can be traced from one entry point as a *static* CFG. When determining what paths of execution the kernel can take, it would be ideal to compute one CFG for the entire kernel. However, modern kernels are designed to be modular and use indirect branches to achieve flexibility so that it is difficult to determine all paths of execution given one entry point. In order to determine all possible paths of execution, we have developed a model that can track indirect branches at run-time. Using this model, it is possible to learn all paths of execution in a kernel. We call the set of all acceptable paths of execution the *dynamic control flow graph*.

In order to determine the dynamic CFG, the monitor must be able to trace all indirect branches. In our model, the monitor disassembles the kernel code to locate indirect branches so that they can be be tagged for tracing. Any new targets that are the result of an indirect branch must also be tracked in a similar manner. Figure 14 shows a simple example. The start of the execution path is denoted by "Entry Point Start". The "Entry Point Start"

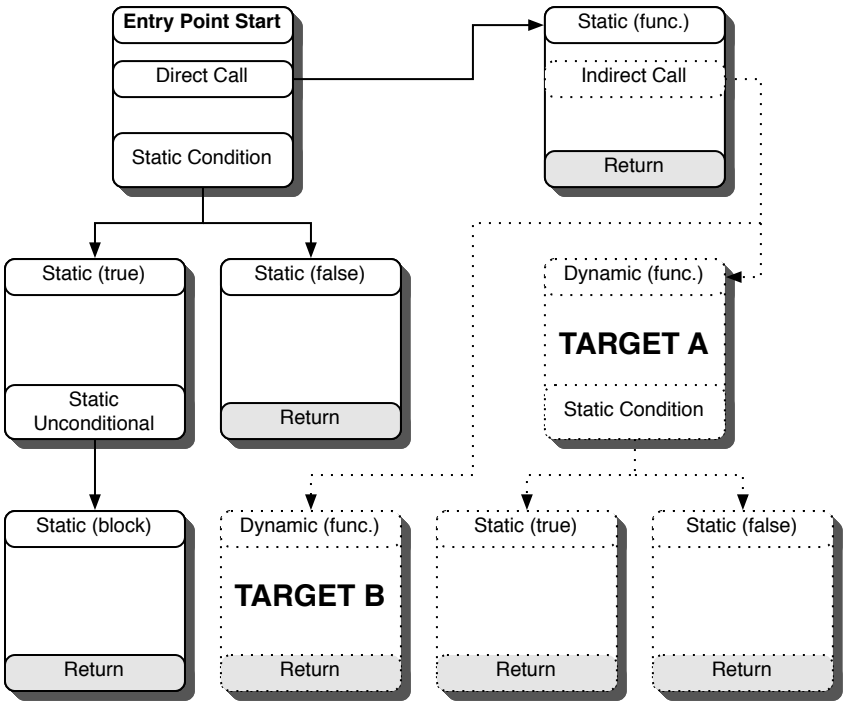


Figure 14: Tracking the dynamic control flow graph

could represent the initial point of execution in a guest kernel. The execution graph can be created by starting with the entry point and tracing all paths of execution that can be tracked statically. If any indirect branch is encountered, then that branch is tagged for tracing. When execution reaches an indirect branch, the tag will send a message to the monitor to track the target of that indirect branch. The key is that a target of an indirect branch can more easily be computed at run-time.

Each box in Figure 14 represents an execution block. The boxes with a solid border can be statically computed based on conditional and unconditional direct branch targets and other executable instructions that are rooted from the initial entry point. The boxes drawn with dotted lines are tracked by monitoring the execution at run-time. In the case shown, an indirect call is being tracked with targets A and B as possible paths of execution. Several direct branches are also tracked.

5.3.1 Learning Period

A key point for layer V_2 indirect branch tracking is that all paths of execution can be tracked during the learning period. After a sufficient learning period, the monitor will ensure that the guest kernel does not execute instructions that are not marked as acceptable during the learning period. It is important to ensure that during the learning period, all paths of execution are realized. In order to provide more flexibility, the system can support lessons at a later point so that new acceptable paths of execution can be realized.

This execution design for layer V_2 is an example of an intrusion recovery system that uses a hybrid integrity and anomaly-based IDS for detection of attack. The learning period falls under the anomaly component of the IDS. During the learning period, layer V_2 learns what the integrity of guest kernel should be. After the learning period, the monitor can check for deviations from integrity and recover from these deviations.

The goal during the learning period is to build data structures that represent the relationships between all the extended execution blocks of executable code. An alternative to learning the code relationships is to specify the relationships with a security policy. However, specifying the relationships may reduce the flexibility of the system. The goal is to determine the dynamic control flow graph of all acceptable paths of execution. Below is an ordered list of required items to meet that goal.

1. Executable Memory Cells — Determine which cells of memory contain code that can possibly be executed as part of the guest kernel's control flow graph.

2. Extended Execution Blocks — Determine sequences of executable memory cells that contain instructions that can possibly execute.

3. Static Control Flow Graphs — Determine the extended execution blocks that make up a static control flow graph based on an entry point.

4. Dynamic Control Flow Graph — Determine all relationships of all static control flow graphs that determine the executable paths in the guest kernel.

5.3.2 Entry Points

An *entry point* is the first instruction in a static CFG that must be given to the monitor in order to trace the extent of the CFG. It is important that the VMM can track any new entry points into the guest kernel and alert the guest monitor of these new entry points. Otherwise, an attacker could request a new entry point that is not tracked and install the kernel-level rootkit in an untracked portion of the kernel. There are different types of entry points to track. Below is a list of possible entry points:

- created thread — A new thread is created. The first instruction of the new thread defines a new entry point to be tracked.

- changed thread — The instruction pointer of an existing thread is changed. The new value of the instruction pointer defines a new entry point that needs to be tracked. If the VMM supports an operation to change thread register contents, such events should be passed to the guest monitor as new entry points.

- hardware handler — Some kernel code is never executed until a hardware event is triggered. The kernel initialization code points the hardware handlers to the correct code. On the x86 architecture, for example, the interrupt descriptor table (IDT) table is initialized at boot. When an interrupt occurs, the hardware looks up a function pointer in the IDT table to determine what code to execute. Each hardware handler defines an entry point. In a VMM, hardware handlers can be abstracted away with software, so these entry points would not necessarily exist in guest operating systems.

- software handler — If the VMM supports some other type of software handler routines, each routine is defined to be an entry point. This entry point may be covered by the creation of a new thread to handle the routine, but it is possible that a special VMM event occurs when registering software handlers in which case the event would need to be tracked.

- dynamic branch target — Targets of dynamic branches can be treated as entry points. When a dynamic branch is encountered, the target of the dynamic branch can be sent

to the monitor in order to track a new execution path. This is handled by the VMM.

5.3.3 Execution Block

An *execution block* is a sequence of instructions that should execute once the first instruction in the block begins execution. This is distinguished from a basic block in that an execution block can contain a *call* branch. Therefore, the sequence of instructions in an execution block do not necessarily execute one after the other because a *call* subroutine may temporarily divert the flow of execution. Once the sequence of instructions is started, however, all instructions in the sequence are guaranteed to execute. Figure 14 shows how execution blocks can be linked together to form a dynamic CFG. An execution block always ends with a branch, which can include static branches and dynamic branches.

We define an *extended execution block* as a sequence of instructions that does not contain an unconditional branch. A conditional branch will branch to another location if a given condition is met. If the given condition is not met, execution will continue with the instruction following the conditional branch instruction. Therefore, not all instructions in an extended execution block are guaranteed to execute sequentially, but it is possible. This type of execution block is useful for building a structure of executable memory cells as it can save storage requirements for the data structures. We are interested in constructing a graph of all possible paths of execution when discovering the dynamic CFG. In our implementation, these paths are built by relating extended execution blocks, as opposed to basic blocks or execution blocks, in order to help minimize the data structure overhead.

5.3.4 Branches

Branches determine the relationship between extended execution blocks when building the dynamic CFG. *Direct* branches can be computed directly from the instruction disassembly information. For example, a direct jump may have a relative offset parameter that can be used to compute the target of the direct jump. *Indirect* branches must be computed at run-time. For instance, it can be considerably difficult to compute the target of an indirect call that uses the value of a register for the target address. Figure 14 shows examples of both direct and indirect branches.

Table 6: Branches that require dynamic tracking on x86 architecture

	conditional	unconditional	subroutine	return
direct				
indirect		X	X	X

We can sub-classify branches into conditional, unconditional, subroutine, and return branches. A *conditional* branch will branch to the specified target only if a given condition is met. An *unconditional* branch will always jump to the specified target. A *subroutine* branch will branch to the specified target with the intent that the target code will eventually return the flow of execution back to the next instruction following the subroutine call. A *return* branch returns from a subroutine call and branches to the instruction immediately following the call instruction assuming the stack has not been corrupted.

Table 6 shows a matrix of branches that require dynamic tracking for the x86 architecture. Of the eight types of branches, only three require dynamic tracking. All of the direct branches can be traced immediately. On the x86 architecture, there are not any indirect conditional branches and there is not a direct return instruction. Only the indirect unconditional, indirect subroutine, and indirect return branches need to be tagged for dynamic tracking.

In order to track an indirect branch, the target of the branch must be computed and sent to the monitor in real-time. Thus, every indirect branch must be tagged in the guest kernel so that it may be traced. There are different methods of tagging the indirect branches. Our implementation replaces the indirect branches with a software interrupt. When a software interrupt is generated by a tagged branch, the handler first computes and securely stores the target information and then executes the branch.

5.3.5 End Block and Termination Points

The end of an extended execution block is usually a branch, which will begin a new extended execution block or terminate a path. However, there are some special instructions that also end an execution block. For example, the undefined instruction on the x86 architecture will

end a block. Also, there may be special instructions on the virtual architecture supported by the VMM that end an execution block.

The end of a path in a static control flow graph is usually a return branch but can also be one of the special instructions. The end of a path can also be an indirect unconditional jump. Each static CFG can vary in size depending on the entry point. The dynamic control flow graph should include all paths of execution in the kernel that begin with entry points and end by some type of termination point.

5.3.6 Executable Memory Graph

For many standard executable file formats, such as the elf standard, the memory addresses of executable code are encoded into the file header tables. This is also true for the guest kernel executable that is loaded into the virtual machine in our prototype. However, kernels are typically designed differently than many user programs. A kernel is designed to be modular and flexible so that different execution paths are possible. A kernel module can be loaded into memory and new execution paths can be inserted into the kernel. Furthermore, although the text section denotes executable code, not all bytes in that section will necessarily be executed. Our execution tracking method can show exactly which bytes in memory are being executed by the guest kernel.

Starting with the initial entry point, memory cells containing executable instructions are marked in the executable memory graph. Each extended execution block maps into the memory of the guest kernel. After the learning period, the executable memory graph should stabilize and deviations from the graph can indicate the presence of a kernel-level rootkit.

Figure 15 shows a simple example of executable memory for the guest kernel. The figure shows a matrix of 256 memory cells. The figure depicts the graph after the monitor has learned which cells are executable based on four entry points. It should not be possible for cells that are not marked as trusted, by being tracked by an entry point, to be executed. This example points out many possible scenarios that can occur during the learning period. First, the executable code may or may not be contiguous. Second, some entry points may result in significantly larger execution graphs than other entry points. Third, it is possible

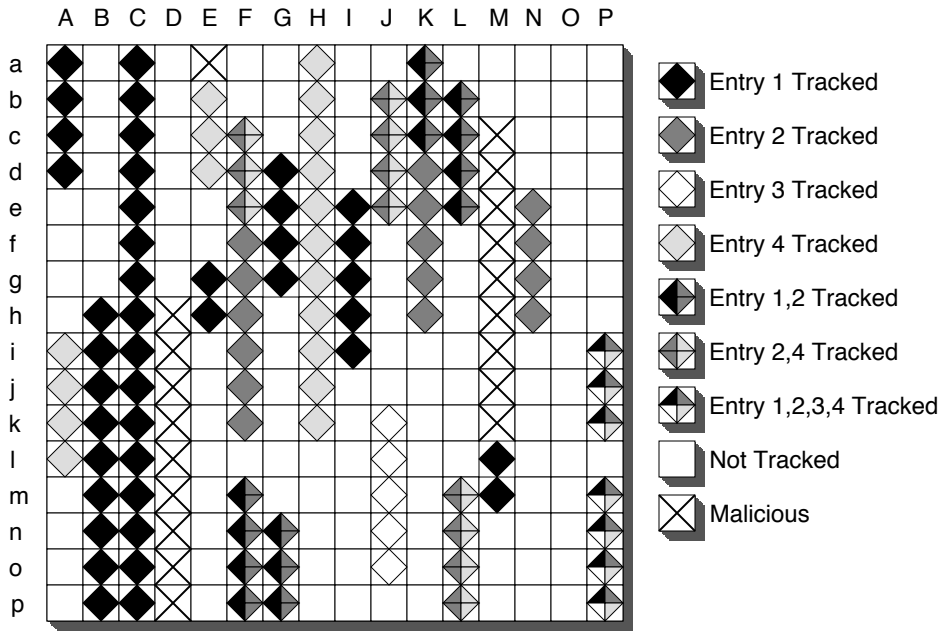


Figure 15: Memory cell graph of kernel executable code

that execution paths from different entry points overlap.

Now consider cells (D,h) through (E, a) and (M,c) through (M,k). These cells have not been marked as trusted executable kernel code. In the figure, in fact,we have marked them as malicious. If a new dynamic target suddenly points to these cells, then a kernel-level rootkit may have been installed on the system.

5.3.7 Timing Sensitive and Other Special Code

It may be important to have some knowledge of the guest kernel code. Tracking dynamic targets adds some overhead to the guest kernel execution. This can cause a problem in special functions that may be time critical, for example. A function that is calibrating the speed of the processor may return an incorrect result due to the extra time required to track indirect branches. Care must be taken with similar special code.

5.3.8 Monitoring Dynamic Control Flow Graph

After the learning period, the monitoring machine switches to monitor mode. In this mode, the dynamic CFG should not change from the expected value. A change from the expected, or learned value, indicates that some kind of malicious activity may be occurring. This

change will happen if a kernel-level rootkit is installed.

A simple method to monitor the dynamic CFG is to store a set of acceptable targets for each indirect branch. Then, the monitor can track all indirect branches and ensure that their targets exist within the set of acceptable targets. More complex monitoring methods can be applied as well.

5.3.9 Integrity of Executable Memory

In order to maintain flexibility, a copy and hash of the executable memory must be stored in the monitoring machine. For example, a user may wish to install a kernel module, uninstall it, and then reinstall it at a different memory location at a later time. The kernel will have an indirect branch that jumps to the kernel module for each different installation, which may not be located at the same virtual memory address. Therefore, the contents of the executable memory are a part of the target description. So, a target that jumps to one piece of code at location X and later jumps to that same code at location Y is the same target.

It is possible to mark pages of memory as read/execute only. If there is a good reason to mark the executable text of the main kernel or any loadable modules as read/write/execute, then it would still be possible to verify the integrity of that code. The contents of the memory can be periodically checked and compared against the known good state as determined from the learning algorithm.

5.3.10 Recovery from Kernel-Level Rootkit

We consider three different possibilities to recover from a kernel-level rootkit once it has been detected by layer V_3. The IRS can repair the kernel without rebooting, restore and reboot the guest OS, or halt the guest OS and wait for user intervention. Ideally, the IRS would always be able to repair the kernel without rebooting the guest OS.

5.4 Experimental Implementation

Based on our design and reason, we have implemented a large portion of our design and experimented with the performance and recovery capabilities of the system. We have leveraged the work of an implementation of Liedtke's μ-kernel specification, known as Fiasco [4].

This kernel serves as our μ-kernel. Furthermore, a port of the Linux kernel to the μ-kernel architecture has been done [9]. We use this kernel as our guest kernel. Our implementation has been done for the i386 architecture.

We have created basic V_4, V_3, V_2, and V_1 layers and use existing hardware for the V_0 layer. The V_4 component monitors the file system and has the capacity to undo illegitimate changes and some other consistency checks for the process listing verses file system listing. The V_3 component does some minimal consistency checking. The V_2 layer verifies that the guest kernel text is not modified including data structures such as the system call table and the virtual file system structures by monitoring the execution paths of the kernel. The V_2 component also has secure storage that is completely isolated from the rest of the machine. Layer V_1 was modified to support the additional requirements of our IRS. Finally, we used existing hardware for layer V_0. We have only partially implemented the vertical integrity checking, such as checking to make sure the V_4 user space process is running and tracking the program counter for each layer.

We have used the sha1 implementation for our hashing algorithms. This hashing algorithm is used in the V_2 and V_4 layers. The sha1 algorithm may not be the most secure hashing algorithm, as current unpublished investigations have claimed, but another hashing algorithm could easily be replaced in our implementation.

Some things that we have not yet implemented include exporting the secure storage interface to the higher layers, implementing persistent storage, and enforcing a secure boot process. We found that some details in the system are more difficult in reality than in theory. For instance, memory management on i386 computers is complicated by various holes and legacy backwards compatible hardware. Although we have run into a number of complicated issues, we do think with reasonable amount of effort it is possible to build a mature and reliable system. Furthermore, future systems can be designed to include support for an intrusion recovery system from the initial stages.

5.4.1 Experimental Test System

We have implemented a prototype of the dynamic CFG tracking on an x86 based system. The thin virtual machine monitor is based on the Fiasco implementation of the L4 μ-kernel [8]. The guest kernel is a port of the Linux kernel to the L4 architecture, which is called the L4Linux server [9]. All experiments and testing were conducted on a Pentium IV CPU running at 2.53GHz with 1GB of RAM. The RAM available to the operating system was limited to 256MB. This memory constraint enabled us to give the same amount of RAM to the virtual machines as to the standard Linux system in order to more accurately compare performance.

5.4.2 Learning Period

For our implementation, we control the learning period using a simple authentication mechanism in the IRS. The learning period is turned on after the system is first installed so that the IRS can learn about the integrity of the system. In our experiments, we perform a series of tasks that represents how a user would normally use the system. After the initialization, the learning period is turned off and the monitoring period is turned on. We also support lessons during which trusted upgrades of the system can be performed so that new integrity can be learned by the IRS.

5.4.3 Monitoring Period

The monitor period is less performance impacting than the learning period, especially for the V_2 layer. Once the system enters the monitoring period, the production system is monitored for possible compromises. Different layers perform different monitoring actions.

One of the main monitoring tasks of the V_2 layer is to track indirect branches of the guest kernel. The system follows a typical producer/consumer solution. All targets are queued by the VMM thread with the corresponding branch location. The monitor machine can then later on dequeue the target information and check for legitimate targets. If a malicious target is found, then a kernel-level rootkit may have been installed. We have only implemented minimal monitoring functionality that enables us to detect and recover kernel-level rootkits.

Basic functionality exists in the implementation to support these features.

5.4.4 Layer V_4

We have implemented file system monitoring and recovery in layer V_4. Using an open source tool called aide, we are able to build a set of known good state for the file system. We then periodically run aide to determine if files on the file system have changed. If they have changed, then the layer V_4 component rolls back the file system to the latest known good state.

5.4.5 Layer V_3

We have implemented minimal consistency checking in layer V_3. Some data structures in the kernel can be changed that would result in hidden processes, files, or network connections. The consistency checking can ensure that data structures remain intact. We have only implemented minimal functionality in this layer.

5.4.6 Layer V_2

We have implemented extensive execution tracking in layer V_2. Layer V_2 also implements the statehold, which can be used by higher layers. We have not exported the statehold storage to higher layers in our implementation. Most of the work in layer V_2 has focused on indirect branch tracking in order to trace the dynamic control flow graph.

Indirect calls or jumps can be traced by replacing their x86 machine code with an *int* instruction. The *int* instruction must have corresponding handler code in the kernel that properly deals with the event. This turns out to be very convenient because the *int* instruction is smaller than or equal to any indirect call or jump instruction on the x86 architecture. This means that overwriting the indirect branch will not overwrite the instruction immediately following the branch instruction.

Before overwriting the instruction, the indirect branch is redirected to an allocated region that is also executable. This redirection is based on a hash value of the memory location. The *int* handler in the kernel uses the same hash function to compute the location of the branch. Before resuming execution, however, the kernel first computes the target and ensures

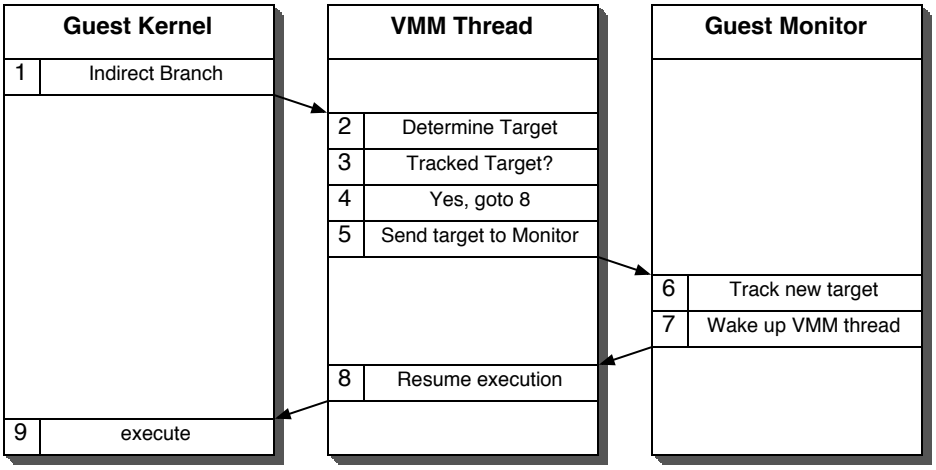


Figure 16: Time sequence of instructions for tracking a new target

that the target has already been tracked. If it has not been tracked, then it sends the target to the monitor to track it. If the monitor is in the learning period, it assumes that the target should be tracked and treats it as a new entry point.

Figure 16 shows a time sequence diagram of the learning algorithm. Any instruction that is not an indirect branch, executes normally. The figure shows how an indirect branch is handled. In our implementation, indirect branch instructions (e.g. *call %eax*, *call *0x08(%edx)*, and *jmp *0x0063deef*) in the L4Linux server are replaced with the *int* τ instruction. The initial entry point into the L4Linux guest kernel is the first instruction that is executed by the process, which is the first instruction in the *main()* function in our implementation. All static execution paths are traced and any indirect branches are replaced with the *int* τ instruction. We have chosen a software interrupt number τ for the *int* instruction that is not currently in use by Fiasco. The *int* τ instruction causes the guest kernel to redirect execution into the VMM thread. We have written a custom interrupt handler for the τ software interrupt that tracks the indirect branches.

During the learning period, the target of all indirect branches is assumed to be a non-malicious target. Therefore, upon reaching an *int* τ, the guest kernel redirects execution to the VMM thread as noted by step 1 in Figure 16. The VMM thread determines if the target associated with the indirect branch has previously been tracked. If it has not been tracked, a notification is sent to the guest monitor and the VMM thread sleeps. The guest monitor

treats the new target as a new entry point and statically analyzes the new execution paths and replaces indirect branches. The data structures of the guest monitor are updated with the newly tracked entry point. After completely tracking all static paths, the VMM thread is notified to resume execution. The VMM thread then redirects the guest kernel to the original indirect branch instruction located at the present EIP value.

Minimal overhead is required to check targets that have already been tracked. A small penalty is incurred in order to redirect the flow of execution into the VMM thread and test to see if the target has been tracked. It is much more expensive to track a target that has not been previously tracked. This sequence of instructions requires a context switch and a potentially lengthy duration of x86 decoding. The length of the decoding depends on how many paths exists from the entry point that have not been previously tracked.

5.4.7 Layer V_1

We have based our layer V_1 implementation on the L4 Fiasco μ-kernel. We have modified the kernel to support the higher layers. Most of our implementation work has focused on adding functionality needed to support layer V_2.

5.4.8 Layer V_0

We have used existing hardware for layer V_0. We believe that hardware support for layer V_0 will enhance our work. Studying hardware support will be an important contribution of future work.

CHAPTER 6

EVALUATION

6.1 Rootkit Benchmark Suite

In order to evaluate our IRS, we have collected a suite of rootkits that include user-level and kernel-level rootkits. The rootkits were collected from the web and from a honeynet [11, 46]. Table 7 shows the list of user-level rootkits that we have collected for testing. The user-level rootkits were selected in order to have a representative sample of different attacks that can be conducted by user-level rootkits.

The *4553-invader* rootkit modifies existing binaries with executable code that opens a remote shell when the binary is executed. The *ark* rootkit replaces binaries, and the source code for *ark* is not available. The *cb-rOOtkit*, *flea*, *rootkit*, and *trojanit* rootkits all replace different binary files. The *lrk4* and *lrk5* rootkits replace binaries and include a network sniffer in order to watch network traffic on the compromised system. The *tOrn* rootkit replaces binaries and also mimics the timestamps of replaced binaries. The *wu-ftpd-trojan* rootkit targets a specific server binary, namely the *wu-ftpd* server. Overall, we believe these rootkits provide a representative sample of user-level rootkit functionality.

Table 7: User-level rootkits in benchmark suite

Rootkit	**Type**	**Description**
4553-invader	User-Level	Modifies binaries
ark	User-Level	Replace binaries; no source code available
cb-rOOtkit	User-Level	Replace binaries
flea	User-Level	Replace binaries
lrk4	User-Level	replaces binaries; includes sniffer
lrk5	User-Level	Later version of lrk4
rootkit	User-Level	Replace binaries;
tOrn	User-Level	Replace binaries; mimics timestamps
trojanit	User-Level	Replace binaries
wu-ftpd-trojan	User-Level	Replaces ftp server

Table 8: Kernel-level rootkits in benchmark suite

Rootkit	Type	Description
adore	Kernel-Level	Redirects system calls
adore-ng	Kernel-Level	Virtual File System Layer redirection
heroin	Kernel-Level	Early Linux kernel-level rootkit
kbd	Kernel-Level	Backdoor access
kis	Kernel-Level	client/server rootkit
knark	Kernel-Level	System call table entry redirection
r.tgz	Kernel/User-Level	Blended rootkit captured on honeynet
rial	Kernel-Level	Redirects system calls
sucKIT	Kernel-Level	System call table redirection
zk	Kernel-Level	System call table redirection

Table 8 shows a list of the kernel-level rootkits included in the rootkit suite. The *heroin* rootkit is believed to be one of the first publicly available kernel-level rootkits. The *adore*, *kbd*, *kis*, *knark*, and *rial* rootkits all replace system calls in the system call table. The *sucKIT* an *zk* rootkits use table redirection to rootkit the system call table. The *adore-ng* rootkit targets the virtual file system layer. The *r.tgz* rootkit was a rootkit captured in the wild on a honeypot [46]. We believe these rootkits represent an accurate sample of the most widely used kernel-level rootkits.

Some of the rootkits in the suite are designed for older systems than our testing system. We ported the functionality of these rootkits so that they worked on our new system. The system was able to recover from the ported rootkits, which had similar functionality on the test system as they would have on the older systems.

6.2 Results on User-Level Rootkit Recovery

In order to test recovery from the user-level rootkits, we first initialized the IRS system with the known good state of the system. The initialization of the user-level state is on the order of minutes as the IRS only needs to take a snapshot of the file system. After initialization, the IRS enters the monitoring mode in which it monitors the integrity of the system. We then tested recovery from each user-level rootkit by installing the rootkit on the system.

Figure 17 shows an example attack on the guest operating system and the automatic self-healing process. In the attack shown, we install the *lrk4 rootkit* onto the guest operating

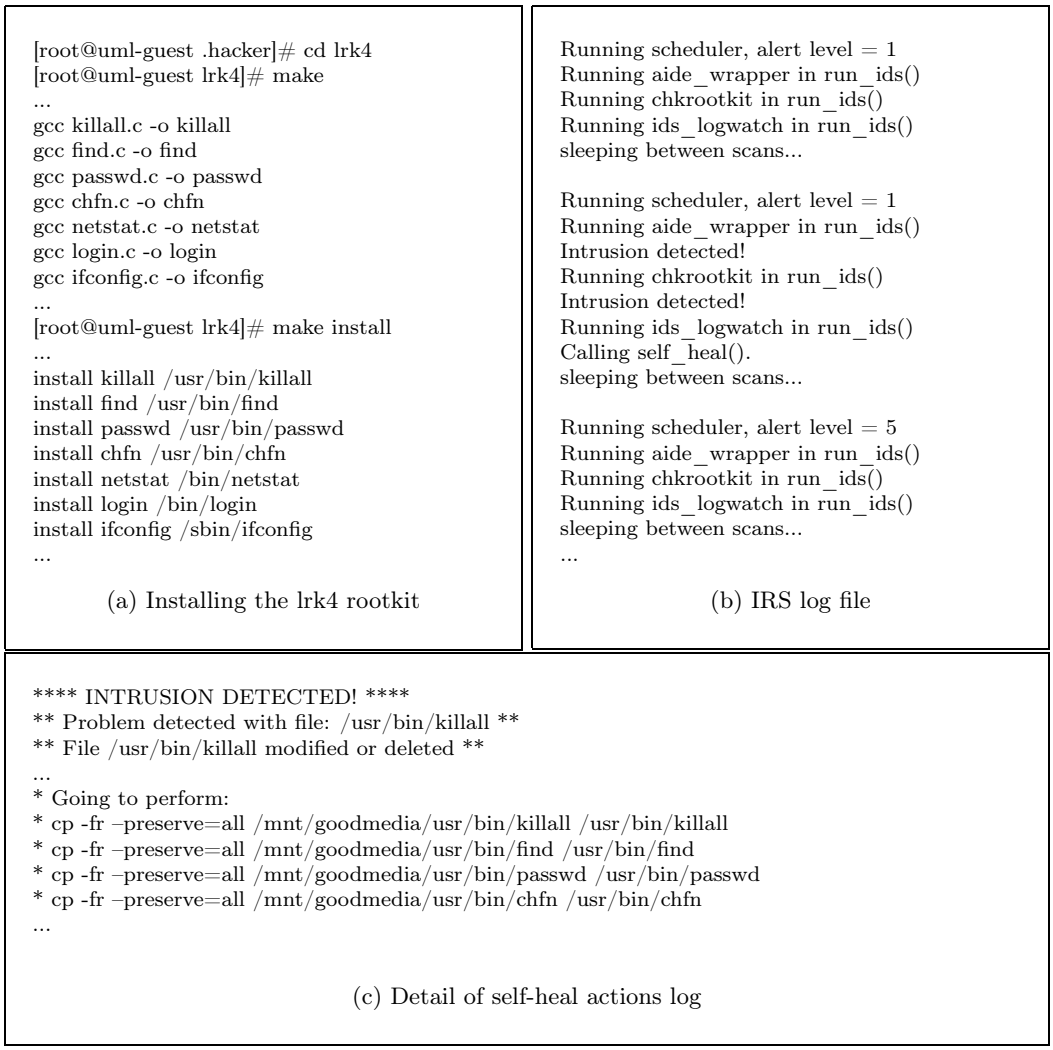


Figure 17: Recovering from a user-level rootkit installation

system. Figures 17(a) shows the rootkit installation. Figure 17(b) shows an overview of monitor status. Figure17(c) shows the self-healing action details.

We installed all rootkits on the test system and examined the logs to determine if the recovery was successful. Full detection and recovery from all rootkits was successful with no false negatives or false positives. Detection of the user-level rootkits was on the order of minutes. Many optimizations could be done on the implementation to achieve near instantaneous detection. For example, our implementation scans the hard disk to check for file changes. A more optimal approach would be to track system calls in order to tag files that have changed.

6.3 Results on Kernel-Level Rootkit Recovery

As with the user-level rootkit recovery, we first initialized the IRS with the known good state of the system before testing recovery from kernel-level rootkits. The learning period for the kernel is less rigid than the learning period for the file system because our algorithm for learning the kernel requires that all subsystems of the kernel are executed. Learning the file system, on the other hand, does not necessarily require all processes to be executed.

Figure 18 shows the results from recovering from three different rootkits. We also tested recovery from the other rootkits in the testing benchmark. All kernel-level rootkits were recovered from with no false positives and no false negatives. Below is a detailed explanation of the recovery from the kernel-level rootkits seen in figure 18.

6.3.1 Recovering from knark

In the first test, we install the *knark* rootkit. The results can be seen in Figure 18(a). The first step is to install *knark*. Since knark is loaded as a kernel module, we insert knark with the *insmod* command. The kernel prints a message warning that *knark.o* does not have an agreeable license. The second step is to hide a binary, which we have placed in the */bin* directory, called *rootme*. The *rootme* binary is part of the knark rootkit and is used to execute binaries with root-level permissions from a regular user account. The *hidef* utility is part of the knark rootkit and is used to hide utilities. In the third step, we list files in the */bin* directory that begin with *root*. No files are shown indicating that our system cannot be trusted. After installation, the IRS detects a breach of integrity and self-heals the system. Notice that upon listing files again, the file *rootme* is seen. Trust has been re-established in the compromised system. Note that during normal operation, the detection of kernel-level rootkits happens much more quickly in our system. However, we slowed down the detection for the purposes of demonstrating the rootkit functionality before and after repair occurred.

6.3.2 Recovering from sucKIT

In our second test, we install the *sucKIT* rootkit. The results can be seen in Figure 18(b). The steps are similar to that of knark. We install the rootkit, show that some files are

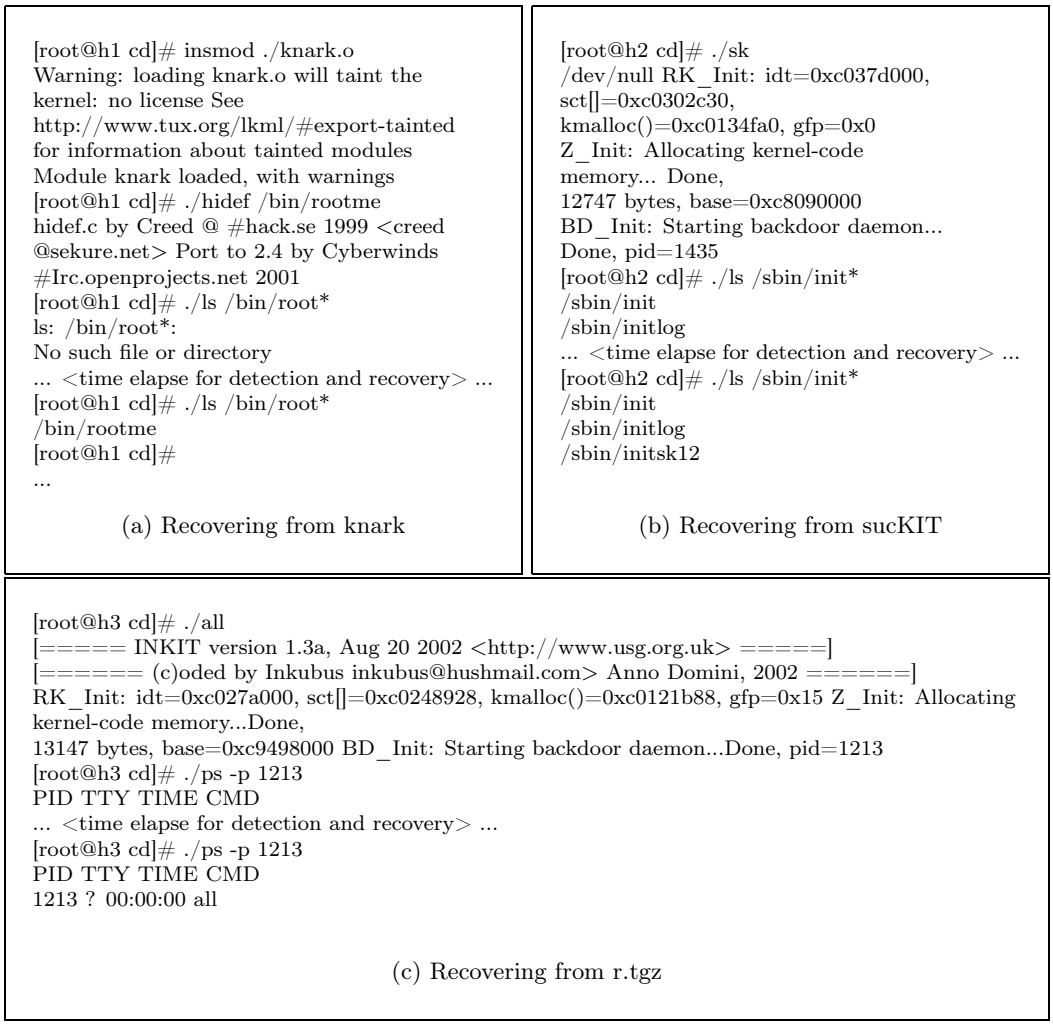


Figure 18: Recovering from kernel-level rootkit installations

hidden when running the *ls* utility, trust is restored by the IRS, and finally we show that the hidden files appear. The *sucKIT* rootkit hides files that have a certain extension, in our case "sk12". The initsk12 file is used in coordination with the init file to load sucKIT upon a reboot. Trust has been re-established in a system that has been compromised with a kernel-level rootkit that redirects the system call table.

6.3.3 Recovering from r.tgz

In our third test, we install the *r.tgz* rootkit. The results can be seen in Figure 18(c). This rootkit is an example of a real-world scenario. In our scenario, an attacker has compromised the system and starts a malicious process with the *all* utility. The *all* utility is part of

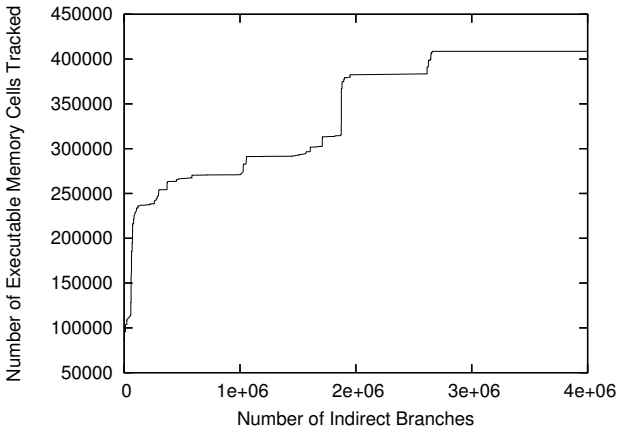


Figure 19: Indirect branches executed over time versus executable memory tracked

the *r.tgz* rootkit. Initially, the process is hidden, as seen by the first *ps* execution. Then, the IRS recovers from the attack. Now the hidden process shows up. The IRS is able to successfully re-established trust in a compromised system that was compromised based on a real-world scenario.

6.4 Learning Results

We have studied the learning period for layer V_2 by measuring the learning methods for the dynamic control flow graph of the guest kernel. We did not study the learning for other layers in much detail. Therefore, in this section, we show results of the learning period for layer V_2.

In order to speed up the learning period, we wrote a script to execute all common tasks that would be commonly executed on a desktop system. Our results of the learning period can be seen in Figure 19, which shows that a stair-step curve that should typical for the learning period. The x-axis shows the number of indirect branches tracked and the y-axis shows how many executable memory cells have been tracked. The rise in each stair step represents significant learning periods where new activity is occurring. The first stair-step shows the kernel initialization activity. The second stair-step occurs when user space applications begin executing and issuing a series of system calls. The final step occurs when we execute our script that generates typical user activity. After the final step, the learning levels off to a slope of zero.

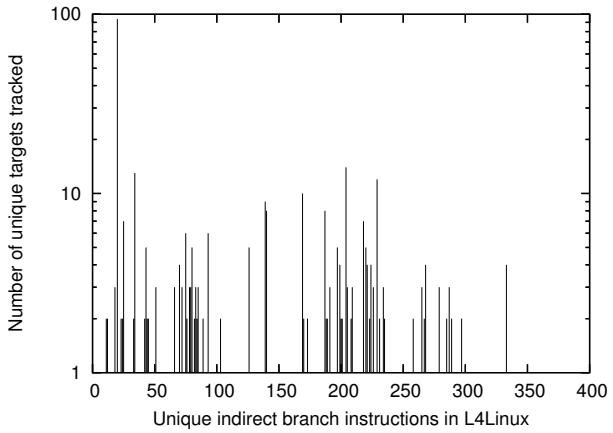


Figure 20: Indirect branches versus unique targets observed during learning period

Figure 20 shows an important result that we have learned from the implementation. It shows a plot of all indirect branches verses the number of different targets they take. This graph shows how the system should operate after the learning period. If this graph changes during execution of the guest OS, then it is likely that malicious activity is occurring. In particular, note the spike of 100 different targets. This represents the indirect branch that is used to execute system calls. On a system that has been infected with a kernel-level rootkit that modifies the system call table, the number of branches tracked by that indirect branch will increase by the number of redirected system calls.

Figure 21 shows how much memory is tracked by each entry point during the learning period. Each new target of an indirect jump is considered a new entry point. Each integer x value represents one entry point that is tracked. The figure shows that there are approximately 475 different entry points that are tracked that vary as to how much new memory is tracked.

Figure 22 shows the executable memory of the kernel. The dark regions indicate memory cells that are known to have executable code based on the learning algorithm. The figure shows the state of the graph after the learning period. There are many light regions that have not been marked for executable regions. These regions include drivers and other subsystems in the kernel that are not executed on the test system. Based on the learned graph, if an indirect target suddenly targets a new section of kernel memory, then a kernel-level rootkit may have been installed on the system. In such a case, the kernel is scanned so that the

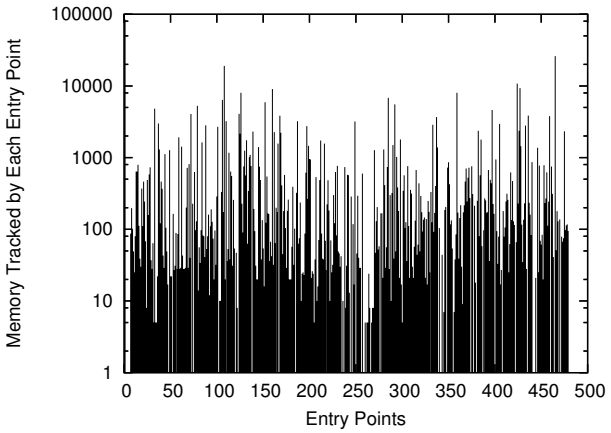


Figure 21: Amount of memory tracked by each entry point in bytes

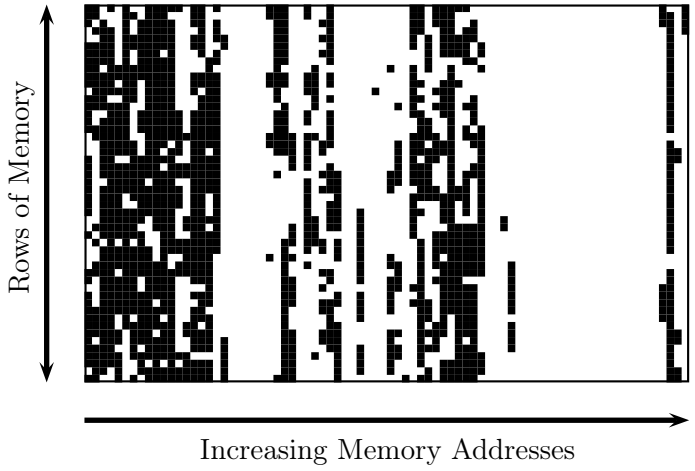


Figure 22: Executable kernel memory cells tracked by V_2 (2.6 MB memory shown)

kernel-level rootkit is removed from the system.

6.5 Performance

We have done some performance analysis of our system comparing the following systems: native Linux, L4Linux, L4Linux with spine and low adaptation, L4Linux with spine and high adaptation. L4Linux is the guest kernel implementation that runs on top of the Fiasco μ-kernel. All tests were conducted on a Pentium IV 3 GHz machine with 1 GB of memory. Although the computer contained 1 GB of memory, we only allowed the production system to use 256 MB of memory. For the spine architecture, the remaining memory was used

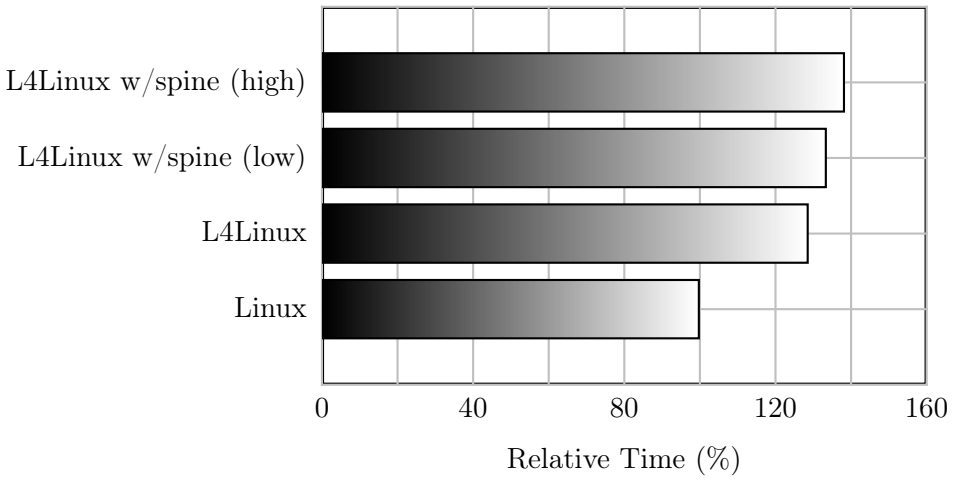


Figure 23: Relative performance in compiling Linux kernel

for the VMM and for secure storage. For the Linux system, the remaining memory was not used. The biggest impact on performance of our system is from the kernel integrity monitoring in layer V_2. We used the 2.6 branch of L4Linux, which is not as optimized for speed as the 2.4 branch. We expect the performance could be dramatically increased by optimizing the L4Linux kernel implementation on top of L4.

6.5.1 Monitoring Period Performance

The first performances results we describe indicate how much performance overhead is caused by tracking the integrity of the kernel. In these results, we do not track the dynamic control graph, but only check the integrity of the kernel text periodically. In these results, we compare Linux, L4Linux, L4Linux with spine during low adaptation, and L4Linux with spine during high adaptation. All systems are based on the stable Debian Sarge distribution [3]. Figure 23 shows the amount of time required to compile a stock Linux kernel. There is a performance loss noticeable in this figure. A loss of 32% is incurred while the system is in its most adaptive state. This performance loss may be acceptable given the higher degree of assurance that the system is operating as expected. The unmodified L4Linux system incurs a 24% performance loss.

Figure 24 shows the number of bytes transferred per second using a TCP/IP connection.

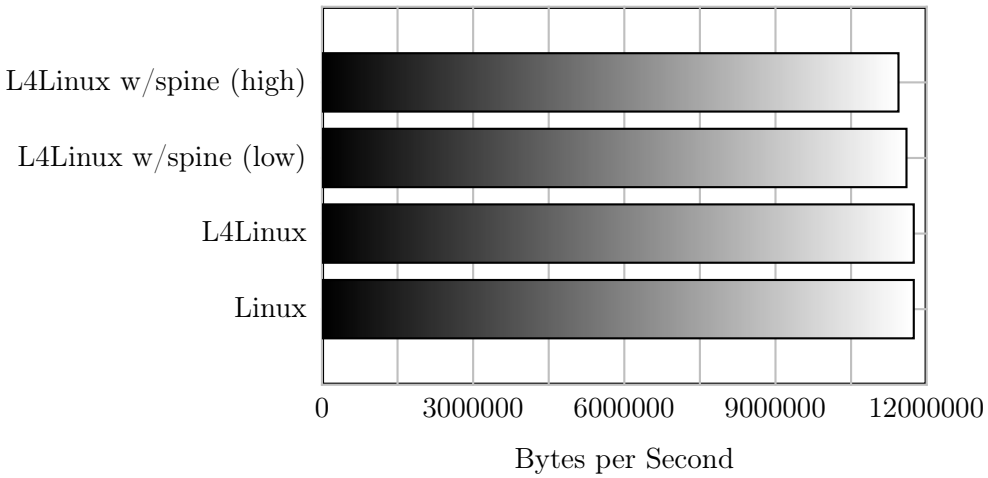


Figure 24: Number of bytes transmitted per second

The test system acted as a TCP sink and another identical Linux system served as a TCP source for each test. The TCP source sent packets of length 1500 with no delay for a period of 60 seconds. There was not much performance loss from the native system as compared to the adaptive L4Linux with spine. The difference is 3%. We expected to see a bigger performance loss. However, our current implementation does not schedule V_1-V_4 with priority over networking events. Therefore, the networking code gets priority, and this is why even the adaptive system performs well.

Our results on the integrity checking show reasonable performance. Next, we present results on the dynamic control flow graph monitoring. In these tests, three different systems are compared with the goal of evaluating additional overhead of the indirect branch tracker. The three systems are referred to as *Linux*, *L4Linux*, and *spine*. We have not implemented varying adaptive levels of monitoring the dynamic control flow graph in the current prototype. The Linux system consists of a minimal and stable Sarge Debian distribution running a stock Linux 2.6.14.1 kernel. The L4Linux system consists of the Fiasco L4 μ-kernel and virtual machine environment, called L4Env, and a port of the Linux 2.6.14 kernel to the L4Env virtual machine environment. The underlying Debian distribution is the same for all systems. The spine system consists of the L4Linux system with the added tracking capability. We do not do any integrity tracking in these tests, so that the overhead of tracking the

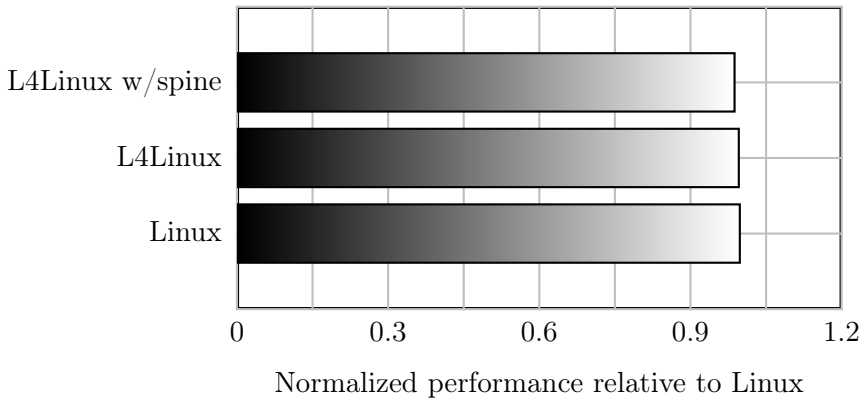


Figure 25: SPEC CPU 2000 benchmarks comparing Linux, L4Linux, and spine

dynamic control flow graph can be assessed.

Figure 25, Table 9, and Table 10 show performance results of the spine architecture as compared to the normal L4Linux system and the stock Linux kernel. The SPEC CPU 2000 benchmarks show virtually no difference in performance. These benchmarks are designed to test the CPU more than the operating system, and so the results are expected. There is slight performance degradations for L4Linux and spine with the SPEC benchmarks, but the difference is minimal.

The lmbench benchmarks better demonstrate that a performance penalty is incurred for running Linux on top of the L4 μ-kernel. The spine column shows a performance penalty near 30% relative to L4Linux but has a worst case of a 74% increase in penalty. The lmbench results show a significant performance penalty as compared to Linux, however the Linux kernel compile time benchmark in Table 10 provides a better approximation of how the overal system will be penalized. Table 10 shows a performance penalty of about 34% for the spine system as compared to Linux system. It should be noted that the implementation of spine is not optimized. We believe that the performance penalty could be as low as 10-20% on a highly optimized system. We tested an early branch of the L4Linux kernel based on Linux 2.4 and found that its performance penalty as compared to Linux was close to 10%. We believe that the 2.6 branch of the L4Linux kernel could be more highly optimized. Furthermore, a system that has hardware support for our architecuture would drastically

improve the performance.

The main factor that affects performance in the L4 architecture is interprocess communication. The μ-kernel architecture introduce a significant increase in interprocess communication because different functions of the operating system are separated into different processes. These processes need to share information, which requires interprocess communication. Monolithic kernels, on the other hand, do not separate different functions into different processes. Instead, monolithic kernels have all functions in one process. Communication can be accomplished with shared memory, which eliminates the need for expensive interprocess communication. However, the reliability and safety of the operating system may not be as robust in monolithic systems.

The benefits of μ-kernels is the added isolation to the system, which can improve security. Our prototype has elements of both μ-kernels and monolithic kernels. The μ-kernel runs directly on the hardware and separates the production system from the monitor. The production system uses L4Linux, which can be considered a monolithic kernel running on top of the μ-kernel. We believe our system offers a good balance between security and performance, although more optimizations are needed.

The integrity checking is less performance intensive than the dynamic control flow graph monitoring. Many optimizations can be made in our current implementation. For example, target tracking for the dynamic control flow graph is currently done for every indirect call. We could track a subset of calls during a low threat level. Based on our initial study on the performance of the system, we do believe that an optimized system could be built that has minimal overhead.

6.5.2 Adaptation

Figure 26 demonstrates the adaptive nature of the integrity checking. Impulses of integrity checking are drawn over time on the x axis. Each impulse represents the initialization of an integrity check of the system. At the point where the impulses are so rapid that the figure appears solid, a rootkit was installed on the system. No other attacks were made against the system during the shown time frame. The system recovers from the installation of the

Table 9: lmbench comparison of Linux, L4Linux, and spine

Benchmark	Linux	L4Linux	Spine	units
Simple syscall:	0.4715	3.19	3.78	microseconds
Simple read:	0.6049	3.56	4.62	microseconds
Simple write:	0.5746	3.46	4.61	microseconds
Simple stat:	2.2575	7.10	9.38	microseconds
Simple fstat:	0.7957	3.62	4.19	microseconds
Simple open/close:	3.8819	14.78	22.04	microseconds
Select on 10 fd's:	1.2585	6.61	8.84	microseconds
Select on 100 fd's:	12.1202	9.65	12.20	microseconds
Select on 250 fd's:	30.3049	14.57	17.14	microseconds
Select on 500 fd's:	60.337	22.82	25.63	microseconds
Select on 10 tcp fd's:	1.2956	6.85	9.15	microseconds
Select on 100 tcp fd's:	13.5445	11.84	14.23	microseconds
Select on 250 tcp fd's:	34.4013	20.07	22.60	microseconds
Select on 500 tcp fd's:	68.8380	34.07	36.39	microseconds
Signal handler installation:	0.9774	5.15	6.62	microseconds
Signal handler overhead:	2.5824	10.44	14.43	microseconds
Pipe latency:	7.6937	25.80	41.74	microseconds
AF_UNIX sock stream latency:	13.47	42.10	73.22	microseconds
Process fork+exit:	182.6694	902.04	1152.78	microseconds
Process fork+execve:	624.8340	2161.20	2827.01	microseconds
Process fork+binsh -c:	6963.7278	12089.23	14414.23	microseconds
File tmpXXX write bandwidth:	29156	28729.00	27868.00	KBsec
Pagefaults on tmpXXX:	2.3161	10.66	14.79	microseconds

Table 10: Comparison of compile time of Linux kernel

Architecture	**Compile Time**	**Relative Time**
Linux	464 seconds	100%
L4Linux	576 seconds	124%
Spine	624 seconds	134%

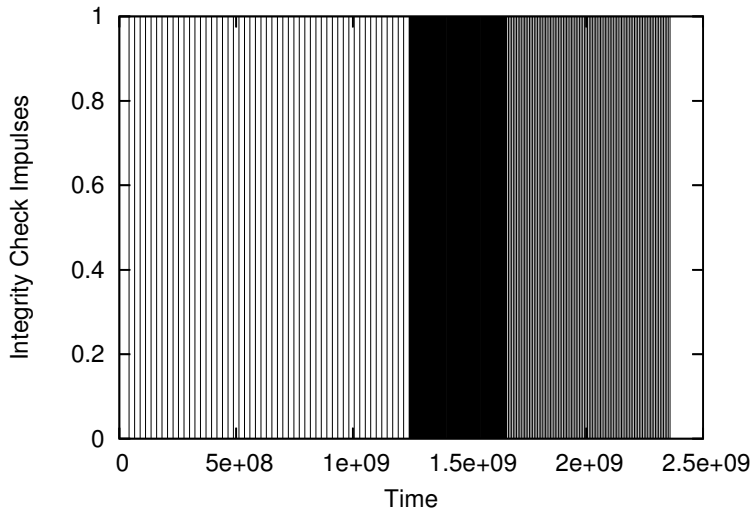


Figure 26: Adaptation after rootkit install

rootkit. As time progresses after the attack, the system slowly adapts back to a normal level of integrity checking. This can be seen by the lighter shades of gray seen by less integrity impulses for a given time period.

6.6 Evaluation of Self-Healing System

Based on our design, implementation, and testing of a self-healing system we can begin to evaluate the notion of a self-healing system. One important result is that systems should support a method to verify their own integrity. Given that a system may be compromised, the mechanism that verifies the integrity of the system is also subject to being compromised. Therefore, another important result is that the mechanism that verifies the integrity of a system should be isolated and independent of the system that it is verifying. However, we have realized that there is significant difficulty in understanding the integrity of a system from a completely independent perspective. Thus, a third important result is that a layered architecture in the system can help verify the integrity of the system. Another result that we highlight in this section is that a system should be able to recover from attacks. If it is possible to determine that the integrity of the system has been compromised, then it should also be possible to determine how to restore the integrity.

Our results show that self-healing systems can be built that can recover from one of the

most difficult types of attacks, which is the installation of a rootkit. We have also seen that a performance penalty is incurred from our design. However, our design could be optimized with better hardware support and a more efficient implementation. We believe that the performance penalty incurred may be worth the added security in many situations.

CHAPTER 7

CONCLUSIONS

7.1 Conclusions

We have presented methods to re-establish trust in a compromised system without completely reformatting and reinstalling the system. Specifically, we have focused on recovery from rootkits that are installed on a compromised system. A rootkit breaks trust in a computer system because it replaces system functionality with functionality that no longer reports trustworthy information. By maintaining a trusted computing base, it is possible to automatically detect and recover from rootkit installations so that trust can be restored to the system. An important concept to re-establish trust is that a complete understanding of the attack is necessary in order to restore trust, which can be understood by understanding the integrity of the system.

The concept of a trusted computing base is vital to building self-healing systems. We have presented an architecture that provides a trusted computing base by running the production system inside a virtual machine. The trusted computing base consists of the virtual machine monitor (VMM) and a system monitor that resides in a second virtual machine. The VMM guarantees that the production machine will be isolated from the trusted computing base.

We believe that the VMM and system monitor can be built in a manner so that it would be significantly difficult to compromise the trusted computing base. The design hinges on only separating the necessary self-healing mechanisms from the production machine, so that the trusted computing base is small. Using this design, the size of trusted computing base is very small, which enables a reasonable analysis of the software for correctness.

To extend the virtual machine design, we have designed a virtual layer architecture that consists of a user layer (V_4), guest kernel layer (V_3), monitor layer (V_2), virtual machine monitor layer (V_1), and hardware layer (V_0). Using a layered architecture, we have presented an intrusion recovery system (IRS) that is capable of detecting and recovering from attacks.

We have shown how the layered architecture can increase simplicity of the monitor, increase the security of the system, and maintain a reasonable performance.

An important part of our work is that we have shown that integrity is another class of intrusion detection systems (IDS) that is distinctive from previous classifications of IDSs. Previous classifications of IDSs specify that IDSs are signature-based, anomaly-based, or hybrid-based. Our additional classification of IDSs is called integrity-based IDSs.

Under previous classifications of IDSs, integrity-based IDSs would fall under the category of anomaly-based IDSs. However, in this work we have shown that a clear distinction exists between anomaly-based IDSs and integrity-based IDSs. While anomaly-based IDSs learn what normal behavior should look like and trigger anomalies from the normal behavior, integrity-based IDSs monitor the integrity of the system in order to detect a compromise. Integrity-based IDSs are the opposite of signature-based IDSs. While signature-based IDSs use a database of known bad state, integrity-based IDSs use a database of known good state. With integrity-based IDSs, we believe the ratio of false positives and the ratio of false negatives approach zero. However, an integrity-based IDS cannot detect a compromise in which the integrity of the system remains intact.

We have developed methods of intrusion detection that can be considered a hybrid of anomaly and integrity-based IDSs. In our work, we have shown that a system can learn what the integrity of the system is during an initialization phase. After the initialization phase, the IDS enters the monitor phase in which it checks the current state of the system against the known good state.

As a part of our host-based IDS, we have introduced a method of tracking the dynamic control flow graph. We have focused this method on tracking the execution paths of the guest kernel. The method is able to dynamically track paths of execution that cannot be tracked statically within reasonable limits of complexity. We have shown that this method can be used to detect and recover from kernel-level rootkits.

Finally, we have built a prototype of our designed system. We developed a suite of 20 user-level and kernel-level rootkits to test the effectiveness of the system. The system was able to recover from all attacks with no false positives and no false negatives. A performance

loss of about 30% is seen in the system, but we believe that many software optimizations and hardware support can dramatically increase the performance of the system.

7.2 Future Work

Based on our work in building self-healing computer systems, we have identified many areas of future research. The areas of research include secure architectures, intrusion detection, and recovery. Below is a summary of some of the important areas of future research.

- *Recovery Dependencies* – In our current work, we have not studied dependency relationships of the state in the system. Future work in this area could develop models for recovery based on dependencies in the system in order to ensure proper recovery actions are taken.

- *Detecting Entry Point* – We believe that it is possible to automatically detect the point of entry into a system. By logging enough event information, we believe that the IRS can automatically backtrack to the point of entry in the system. This backtracking would enable the system to prevent future attacks through the entry point.

- *Integrity Future Work* – We have identified integrity as a form of intrusion detection system. There are many other applications of integrity-based intrusion detection systems that can be studied in future work.

- *Dynamic Control Flow Graph Future Work* – We have introduced a method to dynamically track program execution in order to detect rootkits. We believe that there are other applications of this method. For example, this method may be used to detect buffer overflows.

- *Trusted Computing Base* – We believe that building a small trusted computing base increases the security of the software. In future work, we would like to study this problem in depth to determine if it is possible to prove or at least approach a provably correct trusted computing base.

- *Hardware Support* – We have designed and implemented our prototype for existing hardware. We would like to study new hardware and introduce new hardware techniques that can support or enhance our work.

- *Performance Optimizations* – In addition to hardware support, there are many software optimizations to our existing work. We would like to study methods of optimizing the software components of our work.

APPENDIX A

KNOWN ROOTKITS

Below is a list of currently known rootkits. Many of these rootkit are known and detected by chkrootkit as specified on their web page [2]. Many of the rootkits listed below were downloaded from packetstorm and their descriptions are based on packetstorm's description [11]. These rootkits target different systems. There are likely many more rootkits in use today that are not included in this list either because they are not widely known or are new rootkits that have not been discovered.

Table 11: List of example rootkits

Rootkit	**Description**
4553-invader	Appends parasitic executable executable code to ELF binaries to send a shell to a remote host
adore-0.13	LKM based rootkit that includes PROMISC flag hiding, persistent file and directory hiding, process-hiding, and rootshell-backdoor
adore-0.14	Revision to Adore
adore-0.42	Revision to Adore
adore-ng-0.31	Next Generation of Adore that modifies virtual file system
adore-ng-0.41	Revision to Adore-ng
AjaKit	Rootkit detected by chkrootkit
all-root	LKM that gives all users root
ARK	Replaces syslogd, login, ssh, ls, du, ps, pstree, killall, top, and netstat
Anonoying	Rootkit detected by chkrootkit
Aquatica	Rootkit detected by chkrootkit
Bobkit	Rootkit detected by chkrootkit
cb-r00tkit	Backdoors many binaries and wipes logs
darkside	Hides processes and their children, hides files, manipulates uid's, and modifies the tcp/ip stack to hide connnections
dica	Most likely a t0rn variant
dsc-rootkit	Rootkit detected by chkrootkit
duarawkz	Rootkit detected by chkrootkit
Ducoci	Rootkit detected by chkrootkit
ESRK rootkit	Rootkit detected by chkrootkit

Rootkit	Description
falcon-ssh-diffs	Diffs between a normal ssh server and a rootkited ssh server
FreeBSD	Rootkit detected by chkrootkit
Fu	Rootkit detected by chkrootkit
George	Rootkit detected by chkrootkit
Gold2	Rootkit detected by chkrootkit
heroin	One of the first kernel-level rootkits
Hidrootkit	Rootkit detected by chkrootkit
Illogic	Rootkit detected by chkrootkit
kdbv2	Modifies sys_stat and sys_getuid to allow unauthorized root access
Kenga3	Rootkit detected by chkrootkit
kenny-rk	Rootkit detected by chkrootkit
knark LKM	Kernel-level rootkit that modifies system calls
linux	User-level rootkit that replaces many binaries
linuxroo	User-level rootkit that replaces many binaries
linspy2beta2	Keystroke logger that records TTY activity
LOC	Rootkit detected by chkrootkit
lrk3	Linux rootkit version 3
lrk4	Linux rootkit version 4
lrk5	Linux rootkit version 5
lrk6	Linux rootkit version 6
lrkn	Linux rootkit version n
Madalin	Rootkit detected by chkrootkit
Maniac-RK	Rootkit detected by chkrootkit
MithRa's	Rootkit detected by chkrootkit
OpenBSD rk v1	OpenBSD rootkit
Optickit	Rootkit detected by chkrootkit
phide	Process hiding kernel module
Pizdakit	Rootkit detected by chkrootkit
pop3d-trojan	Rootkit version of pop3 server
rial	LKM that can hide processes, files, directories, LKMs, connections, and file parts
rh[67]-shaper	Rootkit dtected by chkrootkit
rk	Rootkit received from the wild
RK17	Rootkit dtected by chkrootkit
rkssh4	SSH rootkit
rkssh5	Revision to rkssh
rkssh5	Revision to rkssh
Romanian	Rootkit detected by chkrootkit
r.tgz	Rootkit retrieved from Georgia Tech Honeynet
rootedoor	Rootkit detected by chkrootkit
rootkit	User-level rootkit
rootkit–2	Revision to rootkit
rootkitLinux	Linux rootkit

Rootkit	**Description**
rootkitSunOS	Rootkit for Sun Solaris
RSHA	Rootkit detected by chkrootkit
RST.b trojan	Rootkit detected by chkrootkit
sebek LKM	Key stroke logger created for honeynets
sendm-8.9.3trojan	Rootkited sendmail
Shkit	Rootkit detected by chkrootkit
Showtee	Rootkit detected by chkrootkit
shtroj2	Executes commands based on TERM environment variable
shv4	Rootkit detected by chkrootkit
Solaris	Rootkit detected by chkrootkit
sucKIT	Rootkit that redirects SCT via /dev/kmem
sun-5.5.1	Rootkit for Sun Solaris
tasklgt	LKM that gives root to process that reads a special file in /proc
t0rn	Widely used rootkit that was used in worms
toolkit	Rootkit that hides files and processes on Red Hat 9.0
trojanit	Rootkit for Linux and may work on BSD
ulogin	Login rootkit
Volc	Rootkit detected by chkrootkit
wu-ftpd-trojan	Rootkit version of wu-ftp
zaRwT	Rootkit detected by chkrootkit
ZK	Kernel-Level rootkit similar to sucKIT

REFERENCES

[1] "AIDE - advanced intrusion detection environment." http://www.cs.tut.fi/~rammer/aide.html, Date Accessed: June/2004.

[2] "The chkrootkit website." http://www.chkrootkit.org/, Date Accessed: January/2004.

[3] "Debian – the universal operating system." http://www.debian.org/, Date Accessed: March/2006.

[4] "The fiasco microkernel." http://os.inf.tu-dresden.de/fiasco/, Date Accessed: September/2004.

[5] "Georgia Tech honeynet research project." http://www.ece.gatech.edu/research/labs/nsa/honeynet.shtml, Date Accessed: March/2006.

[6] "kern_check.c." http://la-samhna.de/library/kern_check.c Date Accessed: September/2003.

[7] "L4 environment." http://os.inf.tu-dresden.de/l4env/, Date Accessed: March/2006.

[8] "The L4 microkernel family." http://os.inf.tu-dresden.de/L4/ Date Accessed: September/2003.

[9] "L4linux." http://os.inf.tu-dresden.de/L4/LinuxOnL4, Date Accessed: November/2004.

[10] "Linux man pages online." http://os.inf.tu-dresden.de/fiasco/ Date Accessed: March/2006.

[11] "packet storm." http://packetstormsecurity.org/, Date Accessed: March/2006.

[12] "Samhain labs | samhain." http://la-samhna.de/samhain/, Date Accessed: March/2006.

[13] "St. Michael project site." http://sourceforge.net/projects/stjude/, Date Accessed: October/2002.

[14] "The user-mode linux kernel home page." http://user-mode-linux.sourceforge.net/, Date Accessed: February/2006.

[15] "VMware - virtualization software." http://www.vmware.com/, Date Accessed: February/2006.

[16] "Trojan horse programs and rootkits," Tech. Rep. 08/03, National Infrastructure Security Co-Ordination Centre, August 2003.

[17] Arbaugh, W. A., Farber, D. J., and Smith, J. M., "A secure and reliable bootstrap architecture," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 65–71, May 1997.

[18] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pp. 164–177, ACM Press, 2003.

[19] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S., "Extensibility, safety and performance in the spin operating system," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, ACM, December 1995.

[20] Bovet, D. and Cesati, M., *Understanding the Linux Kernel*. Sebastopol, CA: O'Reilly&Associates, 2003.

[21] Bressoud, T. C. and Schneider, F. B., "Hypervisor-based fault tolerance," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 80–107, 1996.

[22] Brown, A. B. and Patterson, D. A., "Undo for operators: Building an undoable e-mail store," in *Proceedings Usenix Annual Technical Conference*, pp. 1–14, Usenix Assoc., 2003.

[23] Busch, H. and Venzl, G., "Proof-aided design of verified hardware," in *28th ACM/IEEE Design Automation Conference*, pp. 391–396, June.

[24] Candea, G., Brown, A. B., Fox, A., and Patterson, D., "Recovery-oriented computing: Building multitier dependability," *Computer*, vol. 37, no. 11, 2004.

[25] Chen, P. M. and Noble, B. D., "When virtual is better than real," in *Proceedings 2001 Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[26] Cowan, C., Pu, C., Maier, D., Hinton, H., and Walpole, J., "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attaks," in *7th USENIX Security Conference*, pp. 63–78, January 1998.

[27] Demsky, B. and Rinard, M., "Data structure repair using goal-directed reasoning," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, (New York, NY, USA), pp. 176–185, ACM Press, 2005.

[28] Denning, D. E., "An intrusion-detection model," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 222–232, february 1987.

[29] Engler, D. R., Kaashoek, F., and Jr., J. O., "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, ACM, December 1995.

[30] et al., G. C., "Microreboot—a technique for cheap recovery," in *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 31–44, Usenix Assoc., 2004.

[31] Feng, H. H., Giffin, J. T., Huang, Y., Jha, S., Lee, W., and Miller, B., "Formalizing sensitivity in static analysis for intrusion detection," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 194–208, May 2004.

[32] Forrest, S., Hofmeyr, S. A., and Somayaji, A., "Computer immunology," *Commun. ACM*, vol. 40, no. 10, pp. 88–96, 1997.

[33] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A., "A sense of self for unix processes," in *1996 IEEE Symposium on Security and Privacy*, pp. 120–128, 1996.

[34] Gao, D., Reiter, M. K., and Song, D., "On gray-box program tracking for anomaly detection," in *13th USENIX Security Symposium*, pp. 103–118, 2004.

[35] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D., "Terra: a virtual machine-based platform for trusted computing," in *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pp. 193–206, ACM Press, 2003.

[36] Giffin, J. T., Dagon, D., Jha, S., Lee, W., and Miller, B. P., "Environment-sensitive intrusion detection," in *8th International Symposium on Recent Advances in Intrusion Detection*, September 2005.

[37] Goldberg, R. P., "Survey of virtual machine research," vol. 7, no. 6, pp. 34–45, 1974.

[38] Hoglund, G. and Butler, J., *Rootkits: Subverting the Windows Kernel*. Upper Saddle River, NJ: Addison Wesley, 2005.

[39] Jr., N. L. P., Fraser, T., Molina, J., and Arbaugh, W. A., "Copilot - a coprocessor-based kernel runtime integrity monitor.," in *USENIX Security Symposium*, pp. 179–194, 2004.

[40] kad (pseudo), "Handling interrupt descriptor table for fun and profit, issue 59, article 4." `http://www.phrack.org/`, July 2002.

[41] Kim, G. H. and Spafford, E. H., "The design and implementation of tripwire: a file system integrity checker," in *ACM Conference on Computer and Communications Security*, pp. 18–29, 1994.

[42] King, S. T. and Chen, P. M., "Backtracking intrusions," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 223–236, ACM Press, 2003.

[43] Kiriansky, V., Bruening, D., and Amarasinghe, S., "Secure execution via program shepherding," in *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[44] Levine, J., Grizzard, J., and Owen, H., "A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table," in *Proceedings of the second IEEE International Information Assurance Workshop*, pp. 107–125, IEEE, April 2004.

[45] Levine, J. G., Grizzard, J. B., Hutto, P. W., and Owen, H. L., "A methodology to characterize kernel level rootkit exploits that overwrite the system call table," in *Proceedings of IEEE SoutheastCon*, pp. 25–31, IEEE, March 2004.

[46] Levine, J. G., Grizzard, J. B., and Owen, H. L., "Application of a methodology to characterize rootkits retrieved from honeynets," in *Proceedings from the fifth IEEE Systems, Man and Cybernetics Information Assurance Workshop*, pp. 15–21, June 2004.

[47] Lie, D., Thekkath, C. A., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J. C., and Horowitz, M., "Architectural support for copy and tamper resistant software," in *Architectural Support for Programming Languages and Operating Systems*, pp. 168–177, 2000.

[48] Liedtke, J., "On μ-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pp. 237–250, ACM, December 1995.

[49] Litty, L., *Hypervisor-Based Intrusion Detection*. PhD thesis, University of Toronto, 2005.

[50] Liu, P., Ammann, P., and Jajodia, S., "Rewriting histories: Recovering from malicious transactions," *Distributed and Parallel Databases*, vol. 8, no. 1, pp. 7–40, 2000.

[51] Locasto, M. E., Sidiroglou, S., and Keromytis, A. D., "Software self-healing using collaborative application communities," in *13th Annual Network and Distributed System Security Symposium*, 2006.

[52] Pfleeger, C. P. and Pfleeger, S. L., *Security in Computing*. Upper Saddle River, NJ: Prentice Hall, 2003.

[53] Reynolds, J., Just, J., Lawson, E., Clough, L., Maglich, R., and Kevitt, K., "The design and implementation of an intrusion tolerant system," in *International Conference on Dependable Systems and Networks*, 2002.

[54] Rosenblum, M. and Garfinkel, T., "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, pp. 39–47, May 2005.

[55] Schneider, F. B., "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[56] sd (pseudo) and devik (pseudo), "Linux on-the-fly kernel patching without lkm, issue 58, article 7." `http://www.phrack.org/`, July 2001.

[57] Smith, J. E. and Nair, R., "The architecture of virtual machines," *Computer*, vol. 38, pp. 32–38, May 2005.

[58] Spitzner, L., *Honeypots: Tracking Hackers*. Boston, MA: Addison Wesley, 2002.

[59] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S., "Secure program execution via dynamic information flow tracking," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 85–96, ACM Press, 2004.

[60] Tevis, J.-E. J. and Hamilton, J. A., "Methods for the prevention, detection and removal of software security vulnerabilities," in *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, (New York, NY, USA), pp. 197–202, ACM Press, 2004.

[61] Thimbleby, H., Anderson, S., and Cairns, P., "A framework for modelling trojans and computer virus infection," *The Computer Journal*, vol. 41, no. 7, pp. 445–458, 1998.

[62] Thompson, K., "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[63] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kägi, A., Leung, F. H., and Smith, L., "Intel virtualization technology," *Computer*, vol. 38, pp. 48–56, May 2005.

[64] Wagner, D. and Dean, D., "Intrusion detection via static analysis," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2001.

[65] Wagner, D. and Soto, P., "Mimicry attacks on host-based intrusion detection systems," in *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 255–264, ACM Press, 2002.

[66] Warrender, C., Forrest, S., and Pearlmutter, B., "Detecting intrusions using system calls: Alternative data models," in *1999 IEEE Symposium on Security and Privacy*, pp. 133–145, May 1999.

[67] Whitaker, A., Cox, R. S., Shaw, M., and Gribble, S. D., "Rethinking the design of virtual machine monitors," *Computer*, vol. 38, pp. 57–62, May 2005.

VITA

Julian Grizzard was born in Greenville, South Carolina in 1980. He received his B.S. in Electrical and Computer Engineering from Clemson University in 2002. He joined the School of Electrical and Computer Engineering at the Georgia Institute of Technology in 2002 from which he received a M.S.E.E. in 2004. He completed his studies at the Georgia Institute of Technology when he graduated with a Ph.D in 2006. His research interests include end-user security, network security, operating systems, visualizations, and educational teaching methods.