PARADIGMS FOR VIRTUALIZATION BASED HOST SECURITY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Tal Garfinkel
March 2010

Acknowledgments

The completion of this thesis is the end of a long journey that has been supported by many a friend, family member, teacher and mentor.

My advisor Mendel Rosenblum has been there for me consistently with sage advice, moral support basically whatever I needed at the time over the last eight years. He has been incredibly generous with his time. He has never pushed his own agenda on me but instead encouraged me to find what I felt really passionate about and pursue it. By example he has taught me the importance of believing in others and myself, and simply giving people space to let their potential unfold. He has also shared with me the gift of patience, not simply chasing after publications or recognition, but instead being willing to wait for the really interesting, really inspiring, really impactful ideas then pursue them fully. As I reach the end of this journey more than simply being happy with my work, I'm deeply happy with the whole experience, the lessons I have learned, and who I have become as a person, and Mendel has no small part in that.

My co-advisor Dan Boneh has also been an amazing source of support. He has never flagged in his encouragement, faith in my abilities, and as a tireless source of good ideas. I have never met someone with such a deep and infectious zeal for research. As a source of encouragement, inspiration and pure fun to work with, Dan has been second to none.

I'm also very grateful to John Mitchell, David Dill, Dawson Engler, Monica Lam, John Ousterhout, and many others on the Stanford faculty for the ideas and encouragement.

The friends who have have helped me through this have also been a huge source of ideas and inspiration. I'm very grateful to Jim Chow and Ben Pfaff for their friendship and collaboration. Also to Martin Casado, Doantam Phan, Jeff Klinger, and to the rest of my friends and cohort who accompanied me through the program.

My friends and supporters at VMware have also been amazingly helpful an generous. I'm especially grateful to Ben Verghese and Steve Herrod for being unflagging supporters of my work, and inspiring people. I'm also very grateful for my other friends, collaborators and mentors at VMware - in particular, Carl Waldspurger, Pratap Suprahmanyam, Mike Chen, Ole Agesen, and Eli Collins and Keith Adams. You guys are awesome.

My family has been an amazing support through undergrad and gradschool and on the long road here. I'm thankful to my mother for her constant support over the years, for teaching me strength and determination that has carried me through this and many other challenges and for encouraging me to pursue my passions. To my dad for his love and support. To my Uncle Zvika who originally taught me to appreciate the beauty of technology, without his passion as a source of inspiration I might never have found my way to computer science. To my Grandparents Dan and Rachel, whose love and support have never been in question. To my Grandmother Arlene for her unfailing love, to my Grandfather Harold, the most inspiring researcher and innovative thinker I know - and an amazingly loving grandfather. Who taught me never to settle for cliches, to keep looking deeper - to learn to really love the process of inquiry.

I'm thankful for my mentors from undergrad who paved the way for all this. Steve Gribble, a brother, friend and amazing researcher. David Wagner, who really inspired my deep enthusiasm for security and taught me a different way of seeing the world. Nikita Borisov and Ian Goldberg and the rest of my friends from 445.

Finally I'm thankful for the security and systems community and my friends and colleagues that have made this so much fun, Neils Provos, Andrew Warfield, Christian Kriebach.

There are so many others who have loved and supported me on this journey, I apologize in advanced for having failed to mention them here.

Abstract

Virtualization has been one of the most potent forces reshaping the landscape of systems software in the last 10 years and has become ubiquitous in the realm of enterprise compute infrastructure and in the emerging field of cloud computing. This presents a variety of new opportunities when designing host based security architectures. We present several paradigms for enhancing host security leveraging the new capabilities afforded by virtualization. First, we present a virtualization based approach to trusted computing. This allows multiple virtual hosts with different assurance levels to run concurrently on the same platform using a novel "open box" and "closed box" model that allows the virtualized platform to present the best properties of traditional open and closed platforms on a single physical platform. Next, we present virtual machine introspection, an approach to enhancing the attack resistance intrusion detection and prevention systems by moving them "out of the box" i.e. out of the virtual host they are monitoring and into a seperate protection domain where they can inspect the host they are monitoring from a more protected vantage point. Finally, we present overshadow data protection, an approach for providing a last line of defense for application data even if the guest OS running an application has been compromised. We accomplish this by presenting two views of virtual memory, an encrypted view to the operating system and a plain text view to the application the owning that memory. This approach more generally illustrates the mechanisms necessary to introduce new orthogonal protection mechanisms into a Guest Operating system from the virtualization layer while maintaining backwards compatibility with existing operating systems and applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis we explore three new paradigms for using virtualization to improve host security. Our exploration is particularly timely given the resurgence of interest in virtualization in the last decade that has propelled it from relative obscurity to its present ubiquity in enterprise computing, software development, and more recently, in the emerging field of cloud computing.

We begin by briefly reviewing the causes for this historical resurgence, and then examining the key properties afforded by virtualization technology that make it such a powerful enabling technology. We conclude our introduction by briefly summarizing each of the three new host security paradigms that make up the central contribution of this thesis.

At the end of the 1960s, the virtual machine monitor (VMM) came into being as a software- abstraction layer that partitions a hardware platform into one or more virtual machines. Each of these virtual machines was sufficiently similar to the underlying physical machine to run existing software unmodified. At the time, general-purpose computing was the domain of large, expensive mainframe hardware, and users found that VMMs provided a compelling way to multiplex scarce resources among multiple applications. Thus, for a brief period, this technology flourished both in industry and in academic research.

The 1980s and 1990s, brought modern multitasking operating systems and a simultaneous drop in hardware cost, which eroded the value of VMMs. As mainframes gave way to minicomputers and then PCs, VMMs disappeared to the extent that computer architectures no longer provided the necessary hardware to implement them efficiently. By the late 1980s, neither academics nor industry practitioners viewed VMMs as much more than a historical curiosity.

The past decade seen a dramatic turn around in this trend as virtualization has again been recognized as a timely solution to a wide range of pressing problems brought on by the complexity of modern multi-tasking operating systems both individually and when deployed at scale.

Ironically, the capabilities of modern operating systems and the drop in hardware cost – the very combination that had obviated the use of VMMs during the 1980s to caused the problems that fueled their revival. Less expensive hardware had led to a proliferation of machines that were often underused and incurred significant space and management overhead. And the increased functionality that had made operating systems more capable had also made them fragile and vulnerable. To reduce the effects of system crashes and breakins, system administrators again resorted to a computing model with one application running per machine. This in turn increased hardware requirements, imposing significant cost and management overhead. Moving applications that once ran on many physical machines into virtual machines and consolidating those virtual machines onto just a few physical platforms increased use efficiency and reduced space and management costs. Thus, the VMMs ability to serve as a means of multiplexing hardware this time in the name of server consolidation led it to prominence.

Moving forward, a VMM will be less a vehicle for multitasking, as it was originally, and more a vehicle for enhancing security, reliability and ease of management. In many ways VMMs give operating system developers another opportunity to develop functionality no longer practical in today's complex and ossified operating systems, where innovation moves at a geologic pace. Functions like migration and security that have proved difficult to achieve in modern operating systems can often be better supported, or significantly augmented at the VMM layer. In this context, VMMs provide a backward-capability path for deploying innovative operating system solutions, while providing the ability to safely pull along the existing software base.

1.0.1 Virtualization as an Enabling Technology

The VMM decouples the software from the hardware by forming a level of indirection between the software running in the virtual machine (layer above the VMM) and the hardware. This level of indirection lets the VMM exert tremendous control over how guest operating systems (GuestOSs) operating systems running inside a virtual machine uses hardware resources, and affords a variety of key properties that make it a powerful tool for enabling innovation at the OS and architecture layer.

A VMM provides a *compatible*, uniform view of underlying hardware, making machines from different vendors with different I/O subsystems look the same, which means that virtual machines can run on any available computer. Thus, instead of worrying about individual machines with tightly coupled hardware and software dependencies, administrators can view hardware simply as a pool of resources that can run arbitrary services on demand. A related benefit of virtualization is *OS independence*. The virtualization layer provides a unified layer for many complex management tasks from individual machine backup, to security and monitoring. Tasks done at this layer are often substantially less dependent (and sometimes totally independent), of the semantics of the guest OS - reducing or eliminating concerns about supporting different OSes, OS versions, etc.

Because the VMM also offers complete *encapsulation* of a virtual machines software state, the VMM layer can map and remap virtual machines to available hardware resources at will and even migrate virtual machines across machines. Load balancing among a collection of machines thus becomes trivial, and there is a robust model for dealing with hardware failures or for scaling systems. When a computer fails and must go offline or when a new machine comes online, the VMM layer can simply remap virtual machines accordingly. Virtual machines are also easy to replicate, which lets administrators bring new services online as needed. Encapsulation also means that administrators can suspend virtual machines and resume them at arbitrary times or checkpoint them and roll them back to a previous execution state. With this general-purpose undo capability, systems can easily recover from crashes or configuration errors. Encapsulation also supports a very general mobility model, since users can copy a suspended virtual machine over a network or store and transport it on removable media.

The VMM can also provide *total mediation* of all interactions between the virtual machine and underlying hardware, thus allowing *strong isolation* between virtual machines and supporting the multiplexing of many virtual machines on a single hardware platform. The VMM can then consolidate a collection of virtual machines with low resources onto a single computer, thereby lowering hardware costs and space requirements. Strong isolation is also valuable for reliability and security. Applications that previously ran together on one machine can now separate into different virtual machines. If one application crashes the operating system because of a bug, the other applications are isolated from this fault and can continue running undisturbed. Total mediation also allows the virtualization layer act as a reference monitor for hardware access, supporting addition of new protection models and intrusion detection capabilities.

1.0.2 Contributions

In this thesis we explore three major paradigms for employing the capabilities of virtualization to improve host security. Each is embodied in a reference architecture and prototype system that illustrates the unique capabilities and limitations of each approach.

First, we present a flexible architecture for trusted computing, called Terra, that allows applications with a wide range of security requirements to run simultaneously on commodity hardware. Applications on Terra enjoy the semantics of running on a separate, dedicated, tamper-resistant hardware platform, while retaining the ability to run side-by-side with normal applications on a general-purpose computing platform. Terra achieves this synthesis by use of a *trusted virtual machine monitor* (TVMM) that partitions a tamper-resistant hardware platform into multiple, isolated virtual machines (VM), providing the appearance of multiple boxes on a single, general-purpose platform. To each VM, the TVMM provides the semantics of either an "open box," i.e. a general-purpose hardware platform like today's PCs and workstations, or a "closed box," an opaque special-purpose platform that protects the privacy and integrity of its contents like today's game consoles and cellular phones. The software stack in each VM can be tailored from the hardware interface up to meet the security requirements of its application(s). The hardware and TVMM can act as a trusted party to allow closed-box VMs to cryptographically identify the software they run, i.e. what is in the box, to remote parties. We explore the strengths and limitations of this architecture by describing our prototype implementation and several applications that we developed for it.

Next, we present an approach to intrusion detection called virtual machine introspection that pulls traditional host based intrusion detection and prevention mechanisms out of the GuestOS and allows them to inspect the guest from the outside. This approach finds inspiration in the difficult choice that traditional host and network based intrusion detection and prevention systems for designers to make. If the IDS resides on the host, it has an excellent view of what is happening in that host's software, but is highly susceptible to attack. On the other hand, if the IDS resides in the network, it is more resistant to attack, but has a poor view of what is happening inside the host, making it more susceptible to evasion. We present an architecture that retains the visibility of a host-based IDS, but pulls the IDS outside of the host for greater attack resistance. We achieve this through the use of a virtual machine monitor. Using this approach allows us to isolate the IDS from the monitored host but still retain excellent visibility into the host's state. The VMM also offers us the unique ability to completely mediate interactions between the host software and the underlying hardware. We present a detailed study of our architecture, including Livewire, a prototype implementation. We demonstrate Livewire by implementing a suite of simple intrusion detection policies and using them to detect real attacks.

Finally, Application security is ultimately limited by to the assurance of operating system running it, and other applications it is colocated with. We introduce a virtual-machine-based system called Overshadow that protects the privacy and integrity of application data, even in the event of a total OS compromise. Overshadow presents an application with a normal view of its resources, but the OS with an encrypted view. This allows the operating system to carry out the complex task of managing an application's resources, without allowing it to read or modify them. Thus, Overshadow offers a last line of defense for application data. Overshadow builds on multi-shadowing, a novel mechanism that presents different views of ”physical” memory, depending on the context performing the access. This primitive offers an additional dimension of protection beyond the hierarchical protection domains implemented by traditional operating systems and processors. We present the design and implementation of Overshadow and show how its new protection semantics can be integrated with existing systems. Our design has been fully implemented and used to protect a wide range of unmodified legacy applications running on an unmodified Linux operating system. We evaluate the performance of our implementation, demonstrating that this approach is practical.

Chapter 2

Virtual Machine Based Trusted Computing

2.1 Introduction

We present a flexible architecture for trusted computing, called Terra, that allows applications with a wide range of security requirements to run simultaneously on commodity hardware. Applications on Terra enjoy the semantics of running on a separate, dedicated, tamper-resistant hardware platform, while retaining the ability to run side-by-side with normal applications on a general-purpose computing platform. Terra achieves this synthesis by use of a *trusted virtual machine monitor* (TVMM) that partitions a tamper-resistant hardware platform into multiple, isolated virtual machines (VM), providing the appearance of multiple boxes on a single, general-purpose platform. To each VM, the TVMM provides the semantics of either an "open box," i.e. a general-purpose hardware platform like today's PCs and workstations, or a "closed box," an opaque special-purpose platform that protects the privacy and integrity of its contents like today's game consoles and cellular phones. The software stack in each VM can be tailored from the hardware interface up to meet the security requirements of its application(s). The hardware and TVMM can act as a trusted party to allow closed-box VMs to cryptographically identify the software they run, i.e. what is in the box, to remote parties. We explore the strengths and limitations of this architecture by describing our prototype implementation and several applications that we developed for it.

2.2 Trusted Computing: What and Why

Commodity computing systems have reached an impasse. There is an increasing need to deploy systems with diverse security requirements in enterprise, government, and consumer applications. However, current hardware and operating systems impose fundamental limitations on the security these platforms can provide.

First, commodity operating systems are complex programs that often contain millions of lines of code, thus they inherently offer low assurance. Building simple, high-assurance applications on top of these operating systems is impossible because applications ultimately depend on the operating system as part of their trusted computing base.

Next, commodity operating systems poorly isolate applications from one another. As a result, the compromise of almost any application on a platform often compromises the entire platform. Thus, applications with diverse security requirements cannot be run concurrently, because the platform's security level is reduced to that of its most vulnerable application.

Further, current platforms provide only weak mechanisms for applications to authenticate themselves to their peers. There is no complete and ubiquitous mechanism for distributed applications to verify the identities of programs they interact with. This makes building robust and secure distributed applications extremely difficult, as remote peers must be assumed to be malicious. It also significantly limits the threat models that can be addressed. For example, an online game server cannot tell whether it is interacting with a game client that will play fairly or one which has been subjected to tampering that will allow users to cheat.

Finally, current platforms provide no way to establish a trusted path between users and applications. For example, an application for trading on financial markets has no way of establishing if its inputs are coming from a human user or a malicious program. Conversely, human users have no way of establishing whether they are interacting with a trusted financial application or with a malicious program impersonating that application.

To address these problems, some systems resort to specialized closed platforms, e.g. cellular phones, game consoles, and ATMs. Closed platforms give developers complete control over the structure and complexity of the software stack, thus they can tailor it to their security requirements. These platforms can provide hardware tamper resistance to ensure that the platform's software stack is not easily modified to make it misbehave. Embedded cryptographic keys permit these systems to identify their own software to remote systems, allowing them to make assumptions about the software's behavior. These capabilities allow closed platforms to offer higher assurance and address a wider range of threat models than current general-purpose platforms.

The security benefits of starting from scratch on a "closed box" special-purpose platform can be significant. However, for most applications these benefits do not outweigh the advantages of general-purpose open platforms that run many applications including a huge body of existing code and that take advantage of commodity hardware (CPU, storage, peripherals, etc.) that offers rich functionality and significant economies of scale. In this work, we describe a software architecture that attempts to resolve the conflict between these two approaches by supporting the capabilities of closed platforms on general-purpose computing hardware through a combination of hardware and operating system mechanisms.

Our architecture, called Terra, provides a simple and flexible programming model that allows application designers to build secure applications in the same way they would on a dedicated closed platform. At the same time, Terra supports today's operating systems and applications. Terra realizes this union with a *trusted virtual machine monitor* (TVMM), that is, a high-assurance virtual machine monitor that partitions a single tamper-resistant, general-purpose platform into multiple isolated virtual machines. Using a TVMM, existing applications and operating systems can each run in a standard virtual machine ("open-box VM") that provides the semantics of today's open platforms. Applications can also run in their own closed-box virtual machines ("closed-box VMs") that provide the functionality of running on a dedicated closed platform. The TVMM protects the privacy and integrity of a closed-box

VM's contents. Applications running inside a closed-box VM can tailor their software stacks to their security requirements. Finally, the TVMM allows applications to cryptographically authenticate the running software stack to remote parties in a process called *attestation*.

Both open- and closed-box VMs provide a raw hardware interface that is practically identical to the underlying physical machine. Thus, VMs can run all existing commodity software that would normally run on the hardware. Because a hardware-level interface is provided, application designers can completely specify what software runs inside a VM, allowing them to tailor an application's software stack to its security, compatibility, and performance needs. Closed-box VMs are isolated from the rest of the platform. Through hardware memory protection and cryptographic protection of storage, their contents are protected from observation and tampering by the platform owner and malicious parties.

The next section presents the Terra architecture and describes the basic properties of trusted virtual machine monitors, the mechanism that allows applications with open-box and closed-box semantics to run side-by-side. It describes the process of attestation that Terra uses to identify the contents of VMs to remote parties and presents several models for user interaction with the TVMM. Section 2.4 describes Terra's local security model, Terra's impact on application assurance, and how remote parties can take advantage of Terra's security model. In Section 2.5 we describe the design of the Terra TVMM. Section 2.6 describes a prototype implementation of this design and the implementation of closed-box applications that utilize our prototype. These applications include a "cheat-resistant" closed-box version of the popular multiplayer game Quake and trusted access points (TAPs), a system of closed-box VMs that can be used to regulate access to a private network at its endpoints. We discuss related work in section 2.7.

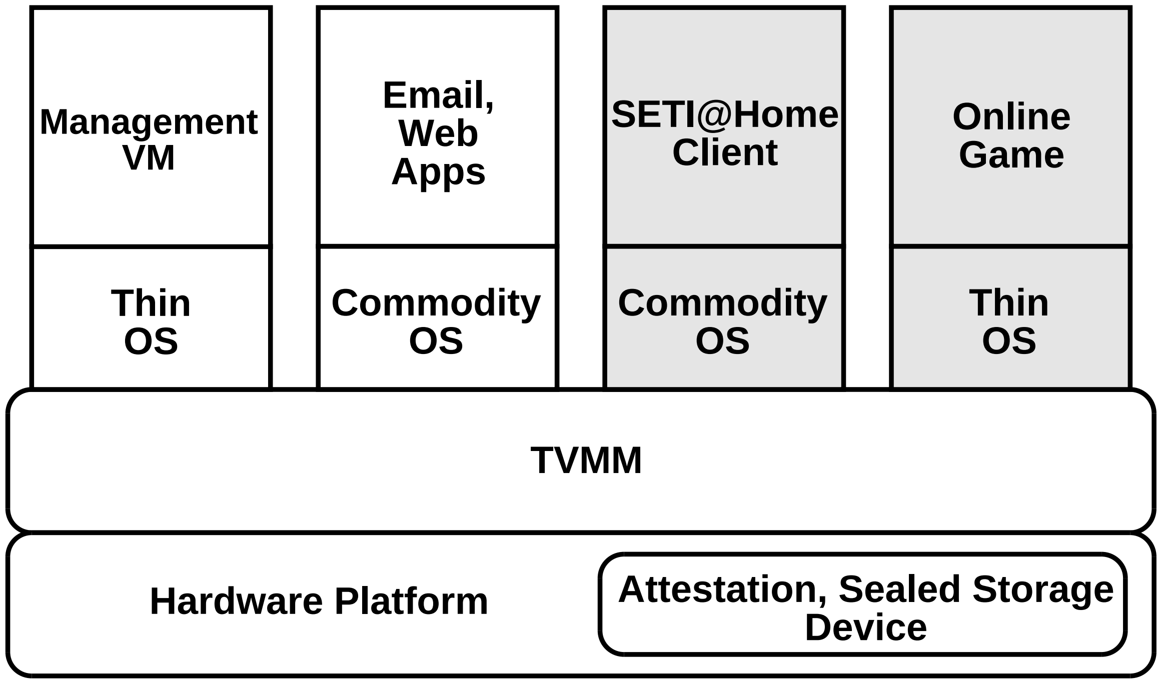

Figure 2.1: **Terra Architecture.** A trusted virtual machine monitor (TVMM) isolates and protects independent virtual machines (VMs). Closed box VMs, shown in gray, are protected from eavesdropping or modification by anyone but the remote party who has supplied the box. Here, the SETI@Home client is in a closed box so that its server can verify that it has not been modified to claim it has run checks that it actually has not, and an online game is in another to deter cheating (see section 2.6.2 for more information). The TVMM can identify the contents of the closed box to remote parties, allowing them to trust it. Also shown here are the management VM and an open-box VM running a commodity operating system.

2.3 Terra Architecture

At the heart of Terra is a virtual machine monitor (VMM). Like any VMM, Terra virtualizes machine resources to allow many virtual machines (VMs) to run independently and concurrently. Terra also provides additional security capabilities including acting as a trusted party to authenticate the software running in a VM to remote parties. Because of this property we refer to it as a "trusted VMM" (TVMM).

At a high level, the TVMM exports two VM abstractions. *Open-box VMs* provide the semantics of today's open platforms. These can run commodity operating systems and provide the appearance of today's general-purpose platforms. *Closed-box VMs* implement the semantics of a closed-box platform. Their content cannot be inspected or manipulated by the platform owner. Thus, their content is secure, neither inspectable nor modifiable by any but those who constructed it, who can explicitly provide themselves access. Figure 2.1 depicts an instance of the Terra architecture with an open-box VM, two closed-box VMs, and the management VM (to be described later).

Terra provides a raw virtual machine as the development target for applications, lending great flexibility to application designers. Applications can be designed from the (virtual) hardware up, using the operating systems that best suit their security, portability, and efficiency needs. Operating systems that run in VMs may be as simple as a bootstrap loader plus application code or as complex as a commodity operating system that runs only one application. Applications can completely tailor the OS to their security needs. Instead of running single closed-box applications, a closed-box VM might run a special trusted OS with a selection of applications designed specifically for it, thus providing something similar to the NGSCB [20] model.

VMs on a single physical machine communicate with one another over virtualized standard I/O interfaces such as NICs, serial ports, etc. The VMM can also multiplex the display and input devices. Thus, from the user's perspective, a closed-box VM may take on the appearance of a normal application, a virtual network appliance, or a virtual device (e.g. a USB device).

The responsibility for configuring how these VMs are granted storage and memory, connected, started, stopped, etc. is delegated to a special *management VM*. The TVMM offers the management VM a basic interface to carry out these tasks. Where the TVMM provides resource management mechanisms, the management VM decides policy, providing a higher-level interface to users and other VMs.

2.3.1 The Trusted Virtual Machine Monitor

Terra's architecture is based on a virtual machine monitor [51], a thin software layer that allows multiple virtual machines to be multiplexed on a single physical machine. The virtual machine abstraction that the VMM presents is similar enough to the underlying architecture that programs and operating systems written for the physical hardware can run unmodified on the virtual hardware. Terra takes advantage of the following properties of traditional VMMs:

Isolation A VMM allows multiple applications to run in different virtual machines. Each virtual machine runs in its own hardware protection domain, providing strong isolation between virtual machines. Secure isolation is essential for providing the confidentiality and integrity required by closed-box VMs. Also, the abstraction of separate physical machines provides an intuitive model for understanding the isolation properties of the platform.

Extensibility Any "one size fits all" approach to providing an operating system for a trusted platform greatly limits a platform's flexibility because it ties all applications to one interface. If this interface is too complex, it compromises the simplicity of the system, forcing many applications to deal with an unacceptably low level of assurance. Conversely, if it is too simple, it compromises the performance and functionality of the system, severely limiting the variety of applications that can usefully take advantage of it.

Terra addresses this conflict by allowing application implementers to view a VM as a dedicated hardware platform, allowing an application's software stack to be built from the (virtual) hardware up. This allows application designers to select the OS that best addresses their requirements for security, compatibility, and performance.

For example, simple applications that require very high assurance (e.g. electronic voting) can use a very minimal OS layer that consists of little more than bootstrapping code. Other applications (e.g. the trusted access points covered in section 2.6.3) may require high assurance and a rich set of OS primitives for access control such as the NSA's SELinux [79] or EROS [104]. A third class of applications may need only a modest level of assurance, but require a relatively feature-rich OS offering high performance and compatibility, such as a stripped-down version of Windows or Linux. Many online games likely fall in this category.

Beyond choosing an operating system to meet their needs, designers can tailor the OS to include only the components required for their applications. Modular OSes such as QNX and Windows CE, commonly used in embedded systems, illustrate how an OS can facilitate this type of application-specific customization.

Efficiency Experience with virtual machine monitors over the past 30 years has shown the overhead of virtualization on virtualizable hardware platforms can be made essentially negligible. Even without virtualizable hardware, the overheads can be made very small [56]. Thus, a VMM can provide essentially the same properties as separate devices with more modest resources whose total resources sum to those of the physical machine. An application running under Terra can potentially be more efficient than its standard OS counterpart because it can tailor the OS abstractions it uses to its needs as in exokernels [36]. This is essential to providing a platform flexible enough to run a wide range of applications with differing performance demands.

Compatibility VMMs can run today's operating systems, such as Linux and Windows, and applications without modifications, unlike alternative approaches to secure isolation, such as microkernels [77] and isolation kernels [126]. This allows existing systems to run under Terra, and means that specialized standalone applications targeted to Terra can run side-by-side with legacy applications. The greater isolation of a VM, compared to a process in an ordinary OS, can improve assurance on its own; untrusted applications can be transformed into low-assurance trusted applications in closed boxes with minimal changes (see section 2.6.2 for an example). It also allows new stand-alone applications to leverage existing toolchains, operating systems, etc. for their construction.

Security A VMM can be a relatively simple program (Disco has only 13,000 lines of code [19]), with a narrow, stable, well-defined interface to the software running above it. Unlike traditional operating systems, that must support filesystems, network stacks, etc., a VMM only needs to present relatively simple abstractions, such as a virtual CPU and memory. As a result of these properties, VMMs have been heavily studied as an architecture for building secure operating systems [64, 49]. VMMs have long been a mainstay of mainframe computing [61], where their security has been leveraged for implementing systems for banking and finance, health care, telecommunication [1], defense [86], etc. The isolation properties of real-world VMMs such as that of the IBM zSeries have received intense scrutiny and been certified as conforming to the highest standards for assurance according to Common Criteria requirements [3].

Terra's TVMM provides three additional capabilities not found in traditional VMMs. These capabilities are essential to providing a "closed box" abstraction:

Root Secure Even the platform administrator cannot break the basic privacy and isolation guarantees the TVMM provides to closed-box VMs.

Attestation This feature allows an application running in a closed box to cryptographically identify itself to a remote party, that is, to tell the remote party what is running inside the closed box. This allows that party to put trust in the application, i.e. to have faith that the application will behave as desired. The following section discusses the basics of attestation.

Trusted Path Providing a *trusted path* from the user to the application is essential for building secure applications [80]. In a TVMM, a trusted path allows a user to establish which VM they are interacting with as well as allowing a VM to ensure that it is communicating with a human user. It also ensures the privacy and integrity of communications between users and VMs, thereby preventing snooping or tampering by malicious programs.

2.3.2 Attestation and VM Identity

Attestation enables an application in a VM to authenticate itself to remote parties [69, 127]. Attestation authenticates who built the platform hardware and what software was started at each layer of the software stack, from the firmware up to the VM. Receiving an attestation tells the remote party what program was started on a platform, but it does not confirm that the program has not subsequently been compromised. The party receiving an attestation must judge for itself how strongly it believes in the correctness and security of each of the platform's layers.

Attestation requires building a certificate chain, from the tamper-resistant hardware all the way to an application VM, to identify each component of the software stack. This chain begins with the hardware, whose private key is permanently embedded in a tamper-resistant chip and signed by the vendor providing the machine. The tamper-resistant hardware certifies the system firmware (e.g. PC BIOS). The firmware certifies the system boot loader, which certifies the TVMM, which in turn certifies the VMs that it loads.

At a high level, each certificate in this certificate chain is generated as follows: A component on the software stack that wants to be certified first generates a public/private key pair. Next, the component makes an Endorse API call (see section 2.5.3) to the lower level component, passing its public key and possibly other application data it wants certified. The lower-level component then generates and signs a certificate containing (1) a SHA-1 hash of the attestable parts of the higher-level component, and (2) the higher-level component's public key and application data. This certificate binds the public key to a component whose hash is given in the certificate.

Certification of a VM being loaded by the TVMM involves the TVMM signing a hash of all persistent state that identifies the VM. This includes the BIOS, executable code, and constant data of the VM. This does not include temporary data on persistent storage or NVRAM contents that constantly change over time. The separation between data which does and does not need to be included in the attestation is application-specific, made by the VM's developer. Terra supports these two type of data by providing VMs with both "attested storage" that the TVMM incorporates in the VM's hash and "unattested storage" that it does not (see section 2.5.2).

Example attestation

As an example of how a VM can use an attestation certificate, consider a home banking application VM, such as Quicken, that is attesting its validity to a remote banking server. For simplicity, we assume that the VM and remote server are establishing an authenticated channel using the standard SSL session key exchange protocol. SSL is well suited for this purpose because it allows both parties the opportunity to present a certificate chain.

For the SSL handshake protocol the VM and remote party use their attestation certificate chains and private keys for authentication. At the end of the protocol both parties share a secret session key. During the handshake protocol, the remote server validates the VM's certificate chain as follows:

1. It verifies that the lowest certificate in the chain, certifying the hardware, is from a trusted certificate authority and that the certificate has not been revoked.

2. It verifies that all hashes in the certificate chain are on the remote server's list of authorized software. That is, the remote server trusts the BIOS, the bootloader, and the TVMM.

3. It verifies that the hash of the VM's attested storage, provided in the topmost certificate, is on a list of authorized applications (e.g. the VM is a valid version of Quicken).

If all these checks are satisfied, then the remote server knows that it is communicating with an authorized application VM. It then completes the session-key exchange protocol to establish an authenticated channel. Omitting the key exchange would open up the attestation process to a man-in-the-middle attack. For example, a malicious user could wait for attestation to complete, then reboot the machine into an untrusted state without the remote server's knowledge.

Establishing trust

Validation of the VM's attestation certificate chain at the remote server requires further explanation. In the discussion above we required the remote server to verify that the hash of the VM's attested storage is on the server's list of authorized applications. However, since there are many versions of a given application it is unreasonable to require the remote server (e.g. a bank) to keep track of hashes of all these versions. Instead, the remote server should require the application VM to also send a certificate from its software vendor (e.g. Intuit, in the case of Quicken) certifying that a given VM hash is indeed a valid version of the application. Thus, we see that the attestation certificate chain proves to the remote server the components that were loaded onto the local machine. The remaining certificates prove what these components are.

We can see from the above that two chains of trust are involved in attestation. Both of these start at a CA (either the same CA or different ones) and end at the application VM. The first chain certifies that a particular software binary image is running; the CA certifies the hardware manufacturer, which signs the tamper-resistant hardware, which signs the TVMM, which signs the application VM's hash. The second chain certifies that the binary image is in fact a version of some interesting program, e.g. version 4.3 of Quicken; the CA certifies the software manufacturer, which signs the VM's hash. Taken together, the two certificate chains show that a VM with a particular hash is running and that that hash represents a particular version of a particular software program. Additional chains, provided by the software vendors that shipped these components, can be used to certify the BIOS, boot loader, and TVMM. During attestation, all the certificates of interest are sent to the remote server, which uses them to decide whether it trusts the various software vendors and whether it trusts the applications that these software vendors are certifying.

Software upgrades and patches

The mechanism described above makes the software upgrade and patch process straightforward. Every upgrade to a VM simply includes a new certificate proving that the resulting VM hash is still a valid version of the application. Cumulative patches that supersede all previously released patches can work the same way.

The situation is a bit more complex for vendors that allow any subset of a collection of patches to be applied to a base VM. We speculate that in this case, the vendor could issue a certificate that states that a specified base VM, plus any or all of a list of specified patches, is a valid version of some software program. The TVMM would sign a certificate that identified the variant in use and include both certificates in attestations.

Revocation

A user who could extract the private key from the tamper-resistant hardware could completely undermine the attestation process. Such a user could convince a remote peer that the local machine is running well behaved software, when in fact it is running malicious code. Worse yet, by widely publishing the private key and certificate the user could enable anyone to undermine the attestation process.

This scenario shows the importance of revoking compromised hardware. Revocation information must be propagated to every host that might depend on revoked certificates for attestation using CRLs, OCSP, or CRTs (see survey of certificate revocation methods in [62]). It is much harder to recover from a compromise of a manufacturer's signing key (e.g. Dell's signing key) without recertifying all deployed devices, so it is critical that manufacturers' private keys be protected as carefully as root CA private keys.

Privacy

The attestation process completely identifies the machine doing the attestation, which raises a privacy concern. Given the resistance met by Intel when it introduced processor serial numbers, this concern must be taken seriously.

One option for maintaining user privacy, proposed by the Trusted Computing Group, is to use a special CA, called a Privacy CA (PCA). Periodically, the user's machine sends an attested certificate request to a PCA. The PCA verifies the machine's hardware certificate and then issues a certificate containing a random pseudonym in place of the real identity. From then on, the machine uses this anonymized certificate for attestation. Although the mapping between real identities and pseudonyms is kept secret, the PCA does keep track of the mappings for revocation purposes. Note that the anonymized hardware certificate must be periodically renewed with a fresh pseudonym; otherwise the anonymized certificate functions as a unique processor ID.

Past experience shows that users are generally unwilling to pay for anonymity services such as PCAs. As a result, the PCA incurs significant liability with no income—not a good business model. Consequently, it is unlikely that this PCA mechanism will be used in practice.

Fortunately, practical cryptographic techniques enable private attestation without the need for a third party. The simplest mechanism, due to Chaum [21], is known as *group signatures*. A practical implementation is given in [14]. In our context, group signatures enable private attestation without any extra work from the user. When using group signatures, the hardware manufacturer embeds a different secret signing key in each machine. As in standard attestation, this key is used to sign the firmware (e.g. BIOS) at boot time. However, the signature does not reveal which machine did the signing. In other words, the attestation signature convinces the remote party that the hardware is certified, but does not reveal the hardware identity. Furthermore, in case a machine's private key is exposed, that machine's signing key can be revoked so that attestation messages from that machine will no longer be trusted.

Interoperability and Consumer Protection

Attestation is a valuable primitive for building secure distributed systems. It frequently simplifies system design and reduces protocol complexity [43]. However, attestation also has a variety of potentially ominous implications that bear careful consideration.

In today's open distributed systems, programs from any source can interoperate freely. This has led to a proliferation of clients and servers for a wide variety of protocols, including commercial, free, and open source variants. This has benefited consumers by fueling innovation, encouraging competition, and preventing product lock-in. Attestation would allow software vendors to create software that would only interoperate with other software they had provided. This creates the tremendous risk of stifling innovation and enabling monopoly control [7]. Given this risk it is critical that the deployment of attestation be given careful consideration, and that appropriate technical and legal protections are put in place to minimize abuse.

Another far-reaching implication of attestation is its ability to facilitate digital rights management (DRM). If trusted computing is deployed ubiquitously, media providers could decide to only release their content to platforms that would prevent copying, expire the media after a certain date or number of viewings, etc. A full discussion of technical, commercial, and legal implications are beyond the scope of this work.

2.3.3 Secure User Interface

A secure user interface provides a trusted path to applications. It prevents malicious applications from confusing the user about which VM is in use. This can be achieved by providing unforgeable and unobstructable visual cues that allow the user to identify the current VM. A wide range of options exist for addressing this from a UI design perspective.

One model presented by the NetTop architecture [86] is a virtual KVM (keyboard, video, mouse) switch model. In this model the user is presented with separate virtual consoles which the user can select using a virtual KVM switch. A small amount of space at the top of the screen displays which VM the physical console is currently showing. This space is reserved for exclusive use by the VMM.

Another option for accomplishing the same end has been used in compartmented mode workstation systems [28]. In these systems a secure window manager controls the entire desktop and applications (in our case, VMs) can only write to portions of the display to which they have been granted access. Tags on the frame of each window indicate which VM owns it, and a dedicated space is again reserved to inform the user which VM is in use.

We have not implemented a secure user interface in our Terra prototype. We believe that implementing a secure UI that allows the capabilities of commodity graphics hardware to be utilized will require additional hardware and software support. This is due to problems imposed by the massive complexity and resulting low assurance of today's video drivers. We discuss how to address these problems in 2.5.5 and 2.5.6 respectively.

2.4 Platform Security

2.4.1 Local Security Model

Terra's basic access control model is specified completely by the TVMM and the management VM. It is assumed that the management VM will make a distinction between the *platform owner* and *platform user*, similar to the distinction between system administrator and normal users in a standard OS access control model. We assume the platform owner can choose the TVMM (or OS) that boots, although only certain TVMMs will actually be trusted by third parties.

The trusted virtual machine monitor runs at the highest privilege level. It is "root secure," [109] meaning that it is secure from tampering even by the platform owner who has root level access, from the management VM, etc. The TVMM only dictates policy that is required for attestation; it isolates VMs from each other, it will not falsely attest to a VM's contents, and it will not disclose or allow tampering with the contents of a closed-box VM. The TVMM cannot guarantee availability. All other policy decisions are left to the discretion of the management VM, i.e. the platform owner.

The management VM formulates all platform access control and resource management policies. It grants access to peripherals, divides storage among VMs, and issues CPU and memory limits. It might formulate policies that limit how many VMs can run, which VMs can run (i.e. what software can run in a given VM), which VMs can access network interfaces or removable media, and so on. The management VM also starts, stops, and suspends VMs.

The management VM that runs is determined by the platform owner, so the security guarantees that the TVMM provides must not depend in any way on the management VM. The TVMM enforces these security guarantees, independent of the management VM. The management VM does have the power to deny service to a VM, by failing to provide a required resource. This power is not a security failing because the platform owner and/or user can do the same thing, e.g. by unplugging the device.

2.4.2 Application Assurance

The most important property Terra provides for improving application security is allowing applications to determine their own level of assurance.

In traditional operating systems isolation between applications is extremely poor. The OS kernel itself has poor assurance and is easily compromised, and the great deal of state that is shared between applications makes it difficult to reason about isolation. As a result, compromising a single application often impacts a significant portion of the platform. Thus, the security of the entire platform is often reduced to that of its most vulnerable component. In Terra, applications in different VMs are strongly isolated from one another. This prevents the compromise of any single-application VM from impacting any other applications on the system. Thus, applications with greatly differing assurance requirements may run concurrently, because an application's level of assurance is independent of other applications on the system.

Terra's ability to run an application-specific operating system aids assurance in a variety of ways. Operating systems tailored to an application can be smaller and simpler than general-purpose OSes. Further, an OS tailored to an application can provide the best abstractions for satisfying the security requirements of that application. For example, the fine-grained access controls of SELinux [79] could be used to compartmentalize a more sophisticated application with many components, while a simpler application could reduce its TCB to simple bootstrapping code.

Attestation also has potential benefits for application assurance. Because applications can ensure that they only interact with trusted peers, they add an additional level of depth to their defenses. An attacker wishing to exploit the application must first either exploit its peer or find some means of impersonating its peer, in order to provide a vector for attack.

The assurance of applications running in Terra is still ultimately limited by the assurance of the operating system, in this case the TVMM. However, we believe that with adequate hardware support a VMM can provide isolation at the highest levels of assurance [3].

2.4.3 Trusting Software

Attestation allows a user to authenticate what hardware and software are in use on a remote platform. This is referred to as establishing that the remote platform is "trusted." This can be a valuable capability for gaining confidence in the integrity of a system's components, but correctly using this capability can also be quite subtle. In particular, it is easy to construe an attestation as promising more than it actually can. It is critical that developers not overestimate what this capability provides.

Naively assuming a client is completely trustworthy based on attestation can potentially make applications needlessly fragile and ultimately degrade their security instead of improving it. This can occur when a remote party places too much faith in the client's good behavior, ignoring relevant issues in the threat model, and making overly strong assumptions about software assurance or hardware tamper resistance. This can also occur if the trust provided by attestation is used as a replacement for stronger alternatives such as cryptography.

Attestation cannot make a promise about the future. A trusted node can fail at any time, through many means—hardware failure, power failure, a user unplugging the device—so an attestation cannot reliably guarantee that the node will do anything at a later time. At best, a trusted platform can only ensure the integrity and confidentiality of the software it is running. In this respect it is no worse than a real closed-box platform.

Hardware in the hands of malicious remote users can only be trusted up to the level of hardware tamper resistance. Current inexpensive commodity hardware that offers only modest tamper resistance, such as the minimal amount required to support TCPA [119], should generally be assumed to deter only the least resourceful attacker. It is our hope that as tamper-resistant hardware becomes more widely deployed, in the form of TCPA, high-quality tamper-resistant hardware will become affordable due to economy of scale. Effective means to take hardware tamper resistance and the threat from the local user and physical security into account in the design of trusted systems has been studied extensively. A good starting point on this topic is work by Anderson [8] and Yee [120].

2.5 Trusted Virtual Machine Monitors

Trusted virtual machine monitors provide application developers with the semantics of real closed-box platforms. VMs provide a raw hardware interface equipped with virtual network cards, video cards, secure disks, etc. VMs can attest to their contents by obtaining signed certificates using a direct interface to the TVMM. The TVMM provides the management VM with interfaces to create and manage VMs, and to connect them through virtual devices. In this section we describe these interfaces, and how they are implemented by the TVMM. We also describe hardware required for building TVMMs.

2.5.1 Storage Interface

The TVMM provides an interface for maintaining application security in the face of the threat model presented by mainstream tamper-resistant hardware such as a TCPA-equipped PC. It assumes that the hardware platform will provide tamper resistance for the memory, CPU, etc., but will not protect the disk. Thus, the disk may be removed from the machine, accessed by a different OS, etc. In light of this threat, Terra provides several classes of virtual disks that VMs can use to secure the privacy and integrity of their data. VMs can select what type of disk they are using for any given virtual disk, based on their security, performance, and functionality requirements:

Encrypted disks hold confidential data. The TVMM transparently encrypts/decrypts and HMACs [16] storage owned by a given VM on that VM's behalf, ensuring the storage's privacy and integrity.

Integrity-checked disks store mutable data whose integrity is important but does not require privacy. The TVMM uses a simple HMAC to prevent tampering. Optionally, a secure counter can prevent rollback (see section 2.5.6).

Raw disks provide unchecked storage. These are useful for sharing data with applications outside the VM.

In addition to these basic disk types, disks are also specified as being attested or unattested. Attested disks contain the program binary and other immutable state that make up the identity of the VM for the purpose of attestation. Which disks are attested is specified as part of a VM's metadata i.e. its basic configuration data. Persistent state that will change, e.g. variable configuration state or application data, is not kept on attestable disks, because a hash of its contents would not generally be meaningful to a remote party. Attestable disks may be encrypted or left in the clear at the discretion of the VM's developer.

Any VM that desires attestation must have been booted from an attestable disk. This disk's hash makes up the *primary identity* of the VM, along with the VM firmware and other immutable VM state. Additional disks may also be made attestable. The hash of each of these disks, if any, constitutes a secondary identity for the VM. The reason for this separation is to facilitate the specialization and redistribution of closed-box VMs. For example, suppose Acme firewall company produces a closed-box VM that provides trusted access point functionality (see section 2.6.3). The primary identity of this VM will be given by the VM supplied by Acme. Each company that purchases this box will add a separate disk which stores site-specific configuration data (e.g. firewall rules, VPN keys). The hash of this disk forms a *secondary identity* for the VM. A VM may have only one primary identity, but it may have any number of secondary identities.

Cryptographic keys used for protecting storage are sealed under the TVMM's public key (see section 2.5.6 for information on sealed storage). The hardware will release the TVMM's private key only to the TVMM itself, maintaining the confidentiality of these keys.

2.5.2 Implementing Attestation

In principle, computing the identity of an application for attestation is done by applying a secure hash to the entire executable image of an application before that application is started. In practice many issues must be taken into account. What portions of the VM are hashed? How is the VM decomposed for hashing? When are the hashes actually computed? The answers to all of these questions have important practical implications for security and performance.

A complete VM image consists of a variety of mutable and immutable data. The VM is defined not only by the initial contents of its virtual disks, but also by its NVRAM, system BIOS, PROMs for any BIOS extensions, and so on. Each VM also includes a "descriptor" that lists hashes for attestable parts of the VM, including attestable disks. The TVMM takes responsibility to ensure that loaded data actually matches these hashes.

Verifying an entire entity (e.g. a virtual disk) with a single hash is efficient only if the entity is always processed in its entirety. If subsections of a hashed entity are to be verified independently (e.g. demand paging a disk) then using a single hash is undesirable. So, instead of a single hash, Terra divides attestable entities into fixed-size blocks, each of which is hashed separately. The VM descriptor contains a hash over these hashes.

If the VM is accompanied by a list of the individual block hashes, subsections of the hashed entity can then be verified at a block-sized granularity, e.g. blocks can be verified as they are paged off disk. Whether the list of hashes is available or not, the entity as a whole can still be verified against the hash of hashes.

The problem of efficiency in hashing an entire entity is recursive. Hashing a 4 GB entity into 20-byte SHA-1 hashes with a 4 kB block size yields 20 MB of hashes.

Storing these hashes on disk should not be a problem, since normal filesystems have a small per-block overhead anyway. A possible real problem is memory and time; before any of these hashes is used to verify a block, the entire 20 MB of hashes must themselves be verified against the hash in the VM descriptor. If it is too expensive to verify these 20 MB of hashes at startup or to keep them in memory, use of a Merkle hash tree [85] would trade startup delay for runtime performance. In the current Terra prototype we have not not yet implemented generalized hash trees to verify hashes, because we have not yet encountered space or performance constraints that necessitate their use.

Ahead-of-Time Attestation

Each stage in the boot process is responsible for signing a hash of the next stage before invoking it. All of these stages deal with small amounts of data that are loaded into memory in a single step. Thus, they are hashed in their entirety before they are given control. We call this "ahead-of-time attestation" because the attestation occurs before the code runs.

After boot, ahead-of-time attestation is appropriate for use with small, high-assurance VMs. The TVMM reads in the entire VM, verifies all of its attestable components against the VM's descriptor. It also pins the VM into physical memory to avoid the possibility of corruption due to malicious tampering.

Optimistic Attestation

Ahead-of-time attestation is impractical for larger VMs. The data to be verified must be both read and hashed. Both of these steps can take a significant amount of time. For example, ignoring disk transfer time, hashing 1 GB of data with OpenSSL's SHA-1 implementation takes over 8 seconds on a 2.4 GHz Pentium 4. (Section 2.6.2 measures performance of attestation in a real VM.) Moreover, any part of an attestable disk that is paged out and later read back must be verified again to detect malicious tampering.

To address these issues, we introduce the technique of "optimistic attestation."

With optimistic attestation, the TVMM attests to whatever hashes the VM descriptor claims for its attestable disks, but it does not verify them at startup. Instead, individual blocks of the VM are lazily checked by the TVMM as they are read from disk at runtime. If a block fails to verify at the time it is read from disk, the TVMM halts the VM immediately.

Ahead-of-Time vs. Optimistic Behavior

Ahead-of-time attestation and optimistic attestation exhibit potentially different semantics. If attestation is done in advance, a single corrupted bit in an attestable disk prevents a VM from loading, but if attestation is performed optimistically, the VM will start and run until the first access to the corrupted block. VM designers may take this into account, but they should be aware that many kinds of events, including hardware failures and power outages, can cause a VM to stop suddenly at any time.

2.5.3 Attestation Interface

The TVMM provides a narrow interface to closed-box VMs for supporting attestation. This interface provides the following operations:

cert ← ENDORSE(*cert-req*)

Places the VM's hash in the common name field of a certificate and places the contents of *cert-req* in the certificate. Signs the certificate with the TVMM's private key, and returns it to the VM. The *cert-req* argument contains the VM's public key and any other application data used for authenticating the VM. This function forms the basis of attestation.

hash ← GET-ID()

Retrieves the hash of the calling VM. (The VM image cannot contain its own hash.) Useful for a VM that wishes to check whether the hash in an attestation from a remote party matches its own hash. This is frequently useful as closed boxes often have peers of the same type, e.g. the online game example shown in section 2.6.2.

2.5.4 Management Interface

Terra delegates VM administration duties to a special VM called the management VM. The management VM is responsible for managing the platform's resources on behalf of the platform owner, providing a user interface for starting, stopping, and controlling the execution of VMs, and connecting VMs through virtual device interfaces. The TVMM provides only basic VM abstractions. This simplifies its design as well as providing flexibility as policy can be completely determined by the management VM.

The TVMM provides basic services, such as support for running multiple, isolated VMs concurrently, but the management VM is responsible for higher-level resource allocation and management. In particular, the management VM allocates memory and disk space to VMs, and controls VM access to physical and virtual devices. It uses a function call interface into the TVMM to accomplish its tasks. The most important of these functions are outlined below:

device-id ← CREATE-DEVICE(*type*, *params*)

Creates a new virtual device of a given type with specified parameters, and yields a handle for the new device. The *type* may specify a virtual network interface, a virtual disk, etc. In the case of a virtual disk, *params* is a list of physical disk extents corresponding to the virtual disk's content. Other types of devices require other kinds of additional parameters.

CONNECT(*device-id-1*, *device-id-2*)
DISCONNECT(*device-id-1*, *device-id-2*)

Connects (or disconnects) the specified pair of devices. Each *device-id* is a virtual device id returned from CREATE-DEVICE or the well-known id of a physical device. When a pair of devices is connected, data output from one of them becomes input on the other and vice versa. For example, a virtual network device can be used to read and write network frames on a real network if it is connected to a physical network device.

vm-id ← CREATE-VM(*config*)

Prepares a VM to be run, and produces a handle for it. The parameter is a set of configuration attributes for the new VM. The configuration includes a pointer to the VM's descriptor. The VM by default has no attached devices.

ATTACH(*vm-id*, *device-id*)
DETACH(*vm-id*, *device-id*)

Attaches a given physical or virtual device to a VM, or removes one, respectively.

ON(*vm-id*)
OFF(*vm-id*)

Powers a VM up or down, respectively.

SUSPEND(*vm-id*)
RESUME(*vm-id*)

Temporarily prevents a VM from running or allows it to resume, respectively. The VM must already be on. (Individual VMs may disable this function.)

2.5.5 Device Driver Security

Device drivers pose an important challenge to TVMM security. Most of today's commodity platforms support a huge range of devices. Today's drivers can be very large (e.g. those for high-end video cards, software modems, and wireless cards) which makes gaining a high degree of confidence in their correctness virtually impossible. Further, there are a huge number of device drivers, and drivers frequently change to support new hardware features. Often these are written by relatively unskilled programmers, which makes their quality highly suspect. Empirically, driver code tends to be the worst quality code found in most kernels [23] as well as the greatest source of security bugs [12]. Given these facts, we cannot expect to include device drivers as part of the TVMM's trusted computing base.

The problem of untrusted device drivers has currently not been addressed by our TVMM prototype. However, a variety of solutions exist. Protecting the TVMM from untrusted device drivers requires several problems to be addressed. First, the TVMM must be protected from direct tampering by the driver code. This is achieved by confining drivers via hardware memory protection and restricting their access to sensitive interfaces. A wide variety of systems have addressed this problem, from exotic microkernel [77, 81] and safe language based systems [17] to practical adaptations to existing operating systems, such as Nooks, which provides device driver isolation for fault tolerance in Linux [113].

A further threat that must be addressed is posed by malicious devices using hardware I/O capabilities (e.g. hardware DMA) to modify the kernel. Addressing this requires additional assistance from the I/O MMU or similar chip set. One approach has been demonstrated in a modified version of the Mungi system [71], that runs device drivers at user level, as independent processes, and prevents them from performing DMA outside their own address spaces.

Another approach to this problem is specified by the forthcoming NGSCB architecture. In NGSCB the issue of supporting device drivers is avoided altogether by leveraging the device drivers of an untrusted operating system (e.g. Windows XP) that runs concurrently on the platform. In NGSCB, a trusted operating system such as a TVMM can run in "curtained memory," memory that is protected from tampering by both the untrusted operating system, and from "attacks from below" via DMA. The trusted operating system leverages the device drivers of the untrusted operating system by interfacing with them via an explicit interface in the untrusted OS's kernel. As a side benefit of this approach, the TVMM does not need to provide its own drivers and instead can leverage those of an existing operating system (e.g. Windows). Leveraging the drivers of another operating system to support a TVMM would be very similar to the hosted VMM approach of VMware Workstation [112].

Untrusted device drivers pose another problem. If the TVMM cannot trust device drivers, it cannot rely on them to provide a trusted path. Overcoming this challenge requires additional hardware support, discussed below.

2.5.6 Hardware Support for Trusted VMMs

Terra relies on the presence of a variety of hardware assistance:

Hardware Attestation Minimally, the hardware must be able to attest to the booted operating system.

Sealed Storage Encrypts data under the private key of the tamper-resistant coprocessor that is responsible for attestation etc. (e.g. a TPM in the TCPA architecture). A hash of the booted trusted OS is also included with the encrypted data. The coprocessor will only allow a trusted OS with the same hash that sealed data to unseal it. This functionality is used by the TVMM to store its private key on persistent storage. Using this functionality ensures that hardware will only release a TVMM's private key to it to the same TVMM that stored it.

Both of these features are currently supported by TCPA. Several other forms of hardware support are desirable:

Hardware Support for Virtualization Specialized hardware support for accelerating virtualization has long been available in IBM mainframes [51]. We believe this type of hardware support significantly eases the burden of implementing a virtual machine monitor capable of efficiently handling the operating system diversity of commodity computing platforms. Hardware assistance is especially important for efficient interfacing to complex hardware such as graphics and 3-D accelerators. Additional hardware support can also greatly simplify virtualization, allowing very simple VMMs to be built, which in turn aids security.

Hardware Support for Secure I/O As discussed above, we cannot assume trust in device drivers on commodity platforms. Given this, it is essential to provide some means of establishing a secure connection between the TVMM and devices required to provide a trusted path (e.g. mouse, keyboard, video card, etc.). One way to accomplish this is by use of cryptography to secure communication between hardware devices and the TVMM. This could be supported either through additional support for encryption on new devices, or by way of hardware dongles to support legacy devices. Clearly encrypting all communication with the device would simply necessitate moving the driver into the TVMM. Thus, another step to supporting secure I/O would be splitting device interfaces. For example, on video cards the interface could be split into a simple 2D interface that could run in the TVMM and be used to implement the secure UI. A sophisticated 3D interface could be exposed directly to VMs, enabling high-performance graphics operations.

Secure Counter A secure counter, that is, a counter that can only be incremented, greatly enhances the functionality of a VM [35]. A secure counter is necessary to guarantee freshness, e.g. to prevent filesystem rollback attacks. A secure real-time clock is also useful, e.g. for expiring old session keys, defending against replay attacks, and rate limiting (discussed in section 2.6). Secure clocks are currently difficult to manufacture inexpensively, so for now it may be necessary to make do with secure counters.

Device Isolation The TVMM would like to protect itself and the VMs it runs from attacks from below, i.e. attacks coming from devices that have access to the DMA controller, PCI bus, etc. Hardware support for controlling access to these resources, in particular to shield VMs and the TVMM from attack would greatly increase the platform's security, because we would not have to trust device drivers. We anticipate that support for limiting device access to DMA, etc., will soon be present in commodity PCs to support Microsoft's NGSCB architecture [20, 4].

Real-Time Support Closed-box applications often have real-time requirements (e.g. game consoles, cellular phones) that cannot be satisfied by today's operating systems or VMMs. We believe additional hardware support could aid in addressing this problem as well. How best to accommodate these through a combination of low-level virtualization techniques and resource management is a topic for future work.

2.6 Experience and Applications

In this section we describe the Terra prototype and provide an in-depth discussion of several applications that we built using the prototype. We also look at how these applications demonstrate the capabilities and the limitations of the closed-box abstraction that Terra provides. We also discuss other potential applications.

2.6.1 Prototype Implementation

We built a prototype of the trusted virtual machine monitor using VMware GSX Server 2.0.1 with Debian GNU/Linux as the host operating system. Neither Debian nor VMware GSX Server is suitably high assurance for a real TVMM, but they form a convenient platform for experimentation. In practice the same techniques that we describe here can be applied to a dedicated VMM offering high performance [56] and assurance, such as a hypothetical lightweight client-side version of VMware ESX Server [123].

Communication between VMs and the TVMM's attestation device is implemented with a VMware virtual serial device. A Python program monitors the host end of this device and handles requests specified by the attestation interface (section 2.5.3).

We currently do not attempt to emulate the underlying TCPA hardware that the TVMM would communicate with. We believe that since these interactions are relatively minimal and well understood, adding it to our prototype system would be superfluous.

Secure Storage

To implement optimistic attestation and other changes to the way VMware GSX Server uses storage, we had to modify the way it accesses virtual disks. We achieved this by interposing on the VMM's read and write operations using a dynamic preload library. This allowed us to modify the underlying implementation of virtual disks to support our new disk types without the need to change the VMM's source code, which was not available to us.

Ahead-of-time attestation was implemented by verifying whole file hashes before a VM is started. For optimistic attestation, our shared library verifies hashes as data is read from the files that VMware GSX Server uses to represent a virtual disk. Only aligned, full-size blocks can be verified with hashes, so the preload library extends the start and end positions of each read to the edge of an aligned block boundary. Misaligned or partial block writes also require one or two block reads. The same strategies are applied to accesses to integrity-checked and encrypted storage. We use bounce buffers to prevent the VM from seeing unverified data, although for performance a real implementation might try to avoid them on aligned full-block reads, perhaps by temporarily marking pages inaccessible.

System Management

A Python program implements the management VM utilizing the interface described in section 2.5.4. It currently only provides a simple means of managing VMs for testing purposes. The management interface is a Python wrapper layered over a variety of management and configuration interfaces provided by VMware GSX Server.

For certificate management we relied on the OpenSSL library. Our certificates are in X.509v3 format, with X.509 certificate requests used to request attestation. Currently the "common name" field is used for the attestation hash; an extension field would be more suitable. The prototype uses a single trusted CA, which signs a hardware certificate, which signs the TVMM's certificate. The TVMM in turn signs each application's attestation certificate.

2.6.2 Trusted Quake

Commercial multiplayer online games have soared in popularity since the mid-1990s. As the popularity of these games has increased, so has the incidence of cheating. Cheating in these games most often occurs when a malicious party alters the client, the server, or their data files to unfairly change the rules of the game. Cheaters may also take advantage of insecure communication between clients and servers, either to spy on their opponents or maliciously alter traffic.

To better understand how to combat these problems in a real-world online game, we built "Trusted Quake," a closed-box version of "Quake II" [57], a popular "first-person shooter" with a long and storied history of problems due to cheating.

Trusted Quake runs Quake in a closed-box VM and uses attestation to ensure that all of the hosts it contacts, whether clients or servers, also run the same version of Trusted Quake. The attestation protocol is used to exchange 160-bit SHA-1 HMAC keys [16] and 56-bit DES keys. All normal Quake traffic is then exchanged using the HMAC and DES keys for integrity and confidentiality, respectively. The TVMM will not falsely attest that a different VM is Trusted Quake, and the isolation properties of the TVMM keep the keys from leaking.

Our prototype of trusted Quake uses a VM running a minimal Linux 2.4.20 kernel on top of a minimal installation of Debian GNU/Linux 3.0. The VM boots directly into Quake. No shell or configuration interface is available to users. A dynamic preload library interposes on Quake's network communication to perform attestation and key exchange. It uses a custom user-space implementation of the IPsec Encapsulating Security Payload (ESP) protocol [2] to provide both integrity and confidentiality. DNS traffic is special-cased, with the preload library checking incoming DNS responses for proper formatting to allow interaction with conventional DNS servers.

We measured the time for the Trusted Quake VM to boot with different forms of attestation. Booting without any form of attestation takes 26.6 seconds. Ahead-of-time attestation adds 30.5 seconds, totaling 57.1 seconds. Substituting optimistic attestation, boot totals only 27.3 seconds. Adding encryption to optimistic attestation raises the total boot time to 29.1 seconds. (Times are averaged over five runs.) We conclude that optimistic attestation has significant benefits for VM startup. As for interactive performance after boot, we found it to be subjectively indistinguishable from untrusted Quake running within a VM.

Security in Quake, as in many such games, originally took the form of "security by obscurity." However, given its huge popularity it was not long before its binary, graphics and audio media files, and network protocol were reverse-engineered by those intent on modifying the game. These modifications led to development of a wide variety of well-documented ways to cheat, by observing and modifying the game client, server, and network traffic.

The security properties provided by Trusted Quake prevent many common types of cheating and other security problems in untrusted Quake:

Secure Communication The secrecy provided by the closed-box VM allows Quake to maintain a shared secret that it can use to securely communicate with its peers. This defeats several forms of cheating. First, since Trusted Quake authenticates all of its traffic, traffic cannot be forged from it, nor can its traffic be modified. This defeats active attacks in the form of aiming proxies [29], agents that interpose on game traffic on behalf of a player to improve aiming. It also defeats passive attacks. When users can observe opponents' Quake network traffic, they can find out important information about game state, such as the location of other players.

Client Integrity Edited client 3D models can facilitate cheating, e.g. modified models of opposing players can make them visible from farther away or around corners. Similarly, clients can modify sounds that indicate nearby players, making them louder or more distinctive [38]. Some Quake variants verify weak checksums of models to attempt to prevent this type of cheating, but these can be bypassed using modified clients or modified models that still match the expected checksum [38]. Trusted Quake frustrates these attacks because users cannot edit files in the Trusted Quake VM.

Server Integrity The Quake server coordinates and controls the game. It is often run by one of the players in a game, so incentive to cheat is strong. A trusted server prevents two kinds of problems. First, it prevents cheating by the server itself, in which the server offers advantages to selected players. Second, it allows only trusted clients to connect, preventing cheating by individual players.

Isolation A corollary of isolating Quake is that the rest of the system is protected if Quake is misbehaving due to remote compromise.

Trusted Quake cannot prevent some kinds of cheating:

Bugs and Undesirable Features Quake has some commands that inadvertently allow cheating. For instance, one command displays the number of rendered polygon models on-screen. When this number increases, it can indicate that another player is about to come into view. Another command can be used to simulate network lag, allowing the player to hang in mid-air for a limited time.

Network Denial-of-Service Attacks Trusted Quake does not affect attacks that prevent communication between a client and a server. This can be used to introduce lag into other players' connections, putting them at a disadvantage. This is especially easy for the server's owner, who has direct control over outgoing packets.

Out-of-Band Collusion Multiple players who are physically near each other can gain extra information by watching each others' monitors or talking to one another, which may allow them an unfair advantage over opponents. Similar cheating is possible via telephone or online chat services.

Trusted Quake provides a specific example of a solution to the very general problem of protecting the privacy and integrity of a complex service in the face of a variety of threats. The techniques we applied here could be used to improve the security of a wide variety of online games, as well as other types of multi-user applications. Trusted Quake also illustrates the limitations of this technique. Even given the features that Terra provides, it is no panacea. Applications must still be carefully designed and some forms of attack simply cannot be prevented with the features Terra provides.

2.6.3 Trusted Access Points (TAPs)

Trusted Quake illustrates how a specific application can be hardened to ensure that it acts as a well-behaved peer. However, for many applications it is not necessary to harden the entire application. Rather, we simply want to ensure that its communication is well regulated, e.g. rate limited, monitored, access controlled. This can be achieved with a trusted access point (TAP), that is, a filter for network traffic that runs on each client that wishes to access the network. The TAP examines both incoming and outgoing packets and forwards only those that conform to policy. A TAP system can be used to secure the endpoints of overlay networks such as corporate VPNs, to secure point-to-point connections to access points such as wireless APs, dial-in access, and even standard wired gateways [43], or simply to regulate access to network service.

We implemented a TAP system designed to allow a company (or other entity) to securely grant outside visitors limited access to its internal network. To receive this limited use of the internal network via the TAP system, a machine's owner physically connects the machine to the "restricted network," that is, a network isolated from the internal network, then installs the TAP closed-box VM on it. This VM contains a VPN client and firewall software for filtering packets. At startup, the VPN client connects to a TAP gateway that bridges the internal network to the restricted network. The client attests itself to the TAP gateway, and the client and gateway server exchange secret parameters used for encrypting and integrity-checking data packets between the two machines.

The TAP VM can implement a traditional network policy preventing IP spoofing, unapproved port usage, rate-limiting, etc. Only packets that adhere to policy are permitted to pass between the internal and restricted networks. The TAP VM can also implement more complex network policy, running remote vulnerability scanners like Nessus and network intrusion detection systems like Snort. When applied to large numbers of clients by a single server, these can consume considerable network and computational resources. Pushing these costs to the client significantly eases the burden.

As with our Quake VM, the TAP prototype runs a minimal Linux 2.4.20 kernel sitting on top of a minimal Debian 3.0 installation within a closed-box VM. The VM is single-purpose and has no user interface. We use the popular OpenVPN secure IP tunnel daemon, version 1.3.0, to transmit packets between the TAP VM and the TAP gateway. Key exchange and certificate presentation is carried out over SSL, a built-in feature of OpenVPN. On the TAP gateway we check the client's certificate via OpenVPN's ability to do so using an external program.

Benefits

Use of a TAP system has several benefits:

Prevents Source Forging The TAP VM can reject packets whose source address does not match the address assigned to the machine.

Prevents DoS Attacks The TAP VM can detect denial-of-service attacks on machines in the internal network and throttle service at the source. (Attempts at source forging might be a sign of a DoS attempt.) Self-detection of DoS attacks could be augmented by notification from an authority on the internal network.

Scalability A centralized router can be overloaded relatively easily if each packet must traverse an entire TCP/IP stack, go through a network intrusion detection system, and so on. When the client that wishes to send or receive packets is also responsible for verifying them, scalability is improved.

Network Scalability Vulnerability scans, such as port scans, can consume considerable network bandwidth. Performing scans between VMs within a computer, instead of over a wire, reduces bandwidth costs and may allow the frequency of scans to be increased.

TAP systems do have limitations. In particular, there can be no assumption that all packets on a wire are authenticated using a TAP system. Nothing prevents an untrusted host from physically connecting to the network, and nothing prevents a trusted host from rebooting into an untrusted OS or bypassing the TAP VM. Thus, attacks, such as flooding attacks, on the restricted network cannot be prevented. However, if individual ports on a switch can be limited to pass only properly HMAC'd or encrypted packets, with some provision for initial negotiation of keys, then this issue can be eliminated.

2.6.4 Additional Applications

We have explored just a few of the potential applications of this platform. It can support a wide range of other applications, including:

- High-Assurance Terminals

  Many applications require a trusted platform as a secure platform for sending or receiving relatively basic information from the user. In these situations we leverage three properties: the platform's ability to provide a trusted path to and from the user, its ability to support high-assurance applications that are highly robust in the face of a remote attacker, and the remote host's ability to ensure that the user is following best practice by running a closed-box version of the application.

One example is "feeds" that report current stock prices, news, and other data that financial analysts use to make decisions. Such interfaces must be extremely reliable. Malicious manipulation of these applications could have devastating consequences for individual traders, whole firms, even entire financial markets. This capability could also be used to provide voting stations that attest their integrity to the remote tabulation service.

- Isolated Monitors

  The strong isolation provided by Terra's use of a VMM is by itself extremely useful. This can be used to harden a variety of host security mechanisms against attack, such as key stores, intrusion detection systems [42], secure logging systems [31], and virus scanners [4].

- Virtual Secure Coprocessors

  Many applications studied in the context of secure coprocessors such as the IBM 4758 [33, 108] also lend themselves to implementation in this architecture. Some of these applications include privacy-preserving databases [109, 58], secure auctions [93], and online commerce applications [129]. A key constraint in adapting applications from such architectures to a trusted platform like Terra will be ensuring that the platform provides an adequate level of hardware tamper resistance for the application.

Clearly, the range of specific applications that can benefit from the general mechanisms provided by Terra is far too long to list. More specific mechanisms that could leverage Terra such as desktop separation [86], application sandboxing, and OS authentication [127] have already been explored elsewhere.

2.7 Related Work

The central mechanism in our work is the virtual machine monitor. Extensive discussion of VMMs and their properties is found in seminal work by Goldberg [50, 51] and more contemporary work on Disco [19] and VMware [112, 123]. More recently, Chen [22] argues for routine and extensive use of VMMs for security purposes.

Our primary reason for choosing a VMM based architecture is the flexibility it provides. Our claim is that a trusted operating system best serves developers by providing a hardware abstraction as a typical closed platform would, thereby providing maximum flexibility. A more general argument about the inherently limiting nature of committing to a single OS abstraction has been made by the extensible OS community, perhaps most concisely in arguing for exokernels [36]. Exokernels and VMMs are in many ways quite similar. They are primarily differentiated by the fact that an exokernel's resource abstractions are optimized for performance, whereas those of a VMM are optimized for compatibility.

Computer systems able to cryptographically demonstrate their security properties to other systems are mentioned first in the work on trusted computing systems [115] and security kernels [49, 99] from the late 1970s and early 1980s. These systems took the principle of least privilege to the extreme in a general-purpose operating system, relying on a small kernel to do isolation, while all other operating functions, such as memory management and process scheduling, were pushed upward into less-trusted code. It was found that this led to systems that by and large were extremely inefficient, for diminishing returns in simplicity. Reported experience with these systems, especially those to kernelize the already svelte VM370 [27] in the form of KVM370 [46, 100], led us to believe that the VMM represents a least common denominator for virtualization, simplification beyond which yields little additional benefit [49].

The concept of authenticating a platform's software stack was fully developed in the Distributed System Security Architecture of Gasser et al. [44]. This work had all the essential components found in today's architectures for trusted computing, such as TCPA [119]. Each computer system contained dedicated hardware with a public/private key pair that it could use to authenticate to others the identity of the system it had booted by signing a hash of the boot image. The operating system (VMS) could in turn use its own key pair to sign for applications loaded by the system, etc., allowing a system's software to fully authenticate itself to a remote system. Including the machine as part of the authentication process, explicitly taking its composition into account, was also included in the authentication systems developed in later work on Taos [127]. This approach is treated thoroughly by Lampson et al. in their related treatise on authentication in distributed systems [69]. The more recent IBM 4758 secure coprocessor [33, 108] also allows for authenticating the source of outbound connections. Authentication in Terra differs most prominently from this previous work on platform authentication in that an application's software stack is treated as a single authenticated unit, in contrast with previous solutions which authenticated to individual parts of an applications software stack in a piecemeal fashion. Terra's support for rapid authentication of large applications further distinguishes it from previous systems. On the opposite end of the spectrum, Execute Only Memory (XOM) [75] uses cryptographic hardware in the processor to preserve the privacy and integrity of code running in a process on an untrusted operating system. It provides much less functionality than Terra for building secure applications, such as a trusted path to I/O devices.

Efforts by Yee and Tygar on Dyad [120] explored hardware mechanisms to bootstrap trust in the host with secure coprocessors on standard PC hardware. More importantly, this work brought to light the practical applications of this technology for consumers, such as electronic currency, stamps, and copy protection, and articulated a vision of including such hardware on mainstream PCs. The AEGIS system by Arbaugh [11] provides a practical foundation for implementing secure boot on a PC. AEGIS uses a signed hash to identify each layer in the boot process, as does Terra. Unlike Terra, the primary purpose of AEGIS is to ensure that only a single authorized software stack can be loaded on a machine. Terra's signatures are designed to prove to third parties the software running on the machine, whereas those in AEGIS enforce booting only a single software stack.

Recently, hardware support for sealed storage and attested boot has become available in the form of commodity platforms implementing TCPA. TCPA 1.1b [119] provides all the basic features to support Terra, although the addition of some of the optional features described in section 2.5.6, such as improved support for device isolation, secure counters, etc., are certainly desirable, and may be forthcoming in the as-yet-unreleased TCPA 1.2 specification. TCPA is only a hardware mechanism for trusted computing, lacking a vision for support of trusted computing in operating systems.

In recognition of this need for OS support for trusted computing, Microsoft began development of its NGSCB (formerly Palladium) architecture [20, 4, 35, 91, 37]. This work is the most similar to ours in that it provides a "whole system" solution to the problem of trusted computing. NGSCB works by partitioning the platform into two parts ("trusted" and "untrusted") each of which runs a different operating system. It achieves this through what can be seen as a very special purpose VMM that only supports two VMs. The untrusted part runs one of today's commodity operating systems (e.g. Windows) while the trusted part runs a dedicated trusted operating system (the "nexus" in NGSCB parlance). This dedicated operating system is designed to run small, high-assurance programs called "agents." Agents work in conjunction with code on the untrusted side of the system, providing all of the security-critical functionality that programs on the untrusted side need (e.g. sensitive key storage).

NGSCB differs from Terra most prominently in its programming model and how it supports high-assurance applications. Terra allows application designers to specify any OS they desire for closed-box applications. In contrast, NGSCB requires application designers to target their closed-box applications to a single, specific Microsoft OS. Terra also differs in its attestation model. In Terra an application's entire software stack is attested, while in NGSCB only agents are attested. Superficially it appears that Terra provides a more flexible model for building applications, but making any concrete comparison at this point would be difficult, because the NGSCB software architecture is as yet largely unpublished.

Ultimately, NGSCB's architecture may complement Terra's. It appears that hardware support for NGSCB may be fairly OS neutral, thus allowing other architectures (such as Terra) to take advantage of the trusted path support in devices, hardware support for isolation, etc. that it provides. Likewise, an architecture like Terra that can provide an arbitrary number of compatible VMs should be able to host a software architecture like NGSCB which requires just two VMs.

We presented the initial idea of providing a closed-box abstraction for trusted computing through the use of a virtual machine monitor in a short position paper [43].

Chapter 3

Virtual Machine Introspection

3.1 Introduction

Widespread study and deployment of intrusion detection systems has led to the development of increasingly sophisticated approaches to defeating them. Intrusion detection systems are defeated either through attack or evasion. Evading an IDS is achieved by disguising malicious activity so that the IDS fails to recognize it, while attacking an IDS involves tampering with the IDS or components it trusts to prevent it from detecting or reporting malicious activity.

Countering these two approaches to defeating intrusion detection has produced conflicting requirements. On one hand, directly inspecting the state of monitored systems provides better visibility. Visibility makes evasion more difficult by increasing the range of analyzable events , decreasing the risk of having an incorrect view of system state, and reducing the number of unmonitored avenues of attack. On the other hand, increasing the visibility of the target system to the IDS frequently comes at the cost of weaker isolation between the IDS and attacker. This increases the risk of a direct attack on the IDS. Nowhere is this trade-off more evident than when comparing the dominant IDS architectures: network-based intrusion detection systems (NIDS) that offer high attack resistance at the cost of visibility, and host-based intrusion detection systems (HIDS) that offer high visibility but sacrifice attack resistance.

In this chapter we present a new architecture for building intrusion detection systems that provides good visibility into the state of the monitored host, while still providing strong isolation for the IDS, thus lending significant resistance to both evasion and attack.

Our approach leverages virtual machine monitor (VMM) technology. This mechanism allows us to pull our IDS "outside" of the host it is monitoring, into a completely different hardware protection domain, providing a high-confidence barrier between the IDS and an attacker's malicious code. The VMM also provides the ability to directly inspect the hardware state of the virtual machine that a monitored host is running on. Consequently, we can retain the visibility benefits provided by a host-based intrusion detection system. Finally, the VMM provides the ability to interpose at the architecture interface of the monitored host, yielding even better visibility than normal OS-level mechanisms by enabling monitoring of both hardware and software level events. This ability to interpose at the hardware interface also allows us to mediate interactions between the hardware and the host software, allowing to us to perform both intrusion detection and hardware access control. As we will discuss later, this additional control over the hardware lends our system further attack resistance.

An IDS running outside of a virtual machine only has access to hardware-level state (e.g. physical memory pages and registers) and events (e.g. interrupts and memory accesses), generally not the level of abstraction where we want to reason about IDS policies. We address this problem by using our knowledge of the operating system structures inside the virtual machine to interpret these events in OS-level semantics. This allows us to write our IDS policies as high-level statements about entities in the OS, and thus retain the simplicity of a normal HIDS policy model.

We call this approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it *virtual machine introspection* (VMI). We will provide a detailed examination of a VMI-based architecture for intrusion detection. A key part of our discussion is the presentation of Livewire, a prototype VMI-based intrusion detection system that we have built and evaluated against a variety of real world attacks. Using Livewire, we demonstrate that this architecture is a practical and effective means of implementing intrusion detection policies.

In Section 3.2 we motivate our work with a comparison of its strengths and weaknesses to other intrusion detection architectures. Section 3.3 discusses virtual machine monitors, how they work, their security, and the criteria they must fulfill to support our VMI IDS architecture. Section 3.4 describes our architecture for a VMI-based intrusion detection systems and the design of Livewire, a prototype VMI-based IDS that implements this architecture. Section 3.5 describes the implementation of our prototype, while Section 3.6 describes sample intrusion detection policies we implemented with our prototype. Section 3.7 describes our results applying Livewire and our sample policies to detecting a selection of real world attacks. In section 3.8 we explore some potential attacks on our architecture, and in Section 3.9 we discuss some related work not touched on earlier in the paper. We present directions for future work in 3.10.

3.2 Motivation

Intrusion detection systems attempt to detect and report whether a host has been compromised by monitoring the host's observable properties, such as internal state, state transitions (events), and I/O activity. An architecture that allows more properties to be observed offers better visibility to the IDS. This allows an IDS's policy to consider more aspects of normative host behavior, making it more difficult for a malicious party to mimic normal host behavior and evade the IDS.

A host-based intrusion detection system offers a high degree of visibility as it is integrated into the host it is monitoring, either as an application, or as part of the OS. The excellent visibility afforded by host-based architectures has led to the development of a variety of effective techniques for detecting the influence of an attacker, from complex system call trace analysis [55, 73, 122, 125], to integrity checking [65] and log file analysis, to the esoteric methods employed by commercial anti-virus tools.

A VMI IDS directly observes hardware state and events and uses this information to extrapolate the software state of the host. This offers visibility comparable to that offered by an HIDS. Directly observing hardware state offers a more robust view of the system than that obtained by an HIDS, which traditionally relies on the integrity of the operating system. This view from below provided by a VMI-based IDS allows it to maintain some visibility even in the face of OS compromise.

Network-based intrusion detection systems offer significantly poorer visibility. They cannot monitor internal host state or events, all the information they have must be gleaned from network traffic to and from the host. Limited visibility gives the attacker more room to maneuver outside the view of the IDS. An attacker can also purposefully craft their network traffic to make it difficult or impossible to infer its impact on a host [95]. The NIDS has in its favor that, like a VMI-based IDS, it retains visibility even if the host has been compromised.

VMI and network-based intrusion detection systems are strongly isolated from the host they are monitoring. This gives them a high degree of attack resistance and allows them to continue observing and reporting with integrity even if the host has been corrupted. This property has tremendous value for forensics and secure logging [31]. In contrast, a host-based IDS will often be compromised along with the host OS because of the lack of isolation between the two. Once the HIDS is compromised, it is easily blinded and may even start to report misleading data, or provide the adversary with access to additional resources to leverage for their attack.

Host-based intrusion detection tools frequently operate at user level. These systems are quite susceptible to attack through a variety of techniques [53, 10] once an attacker has gained privileged access to a system. Some systems have sought to make user-level IDSes more attack resistant through "stealth," i.e. by hiding the IDS using techniques similar to those used by attackers to hide their exploits, such as hiding IDS processes by modifying kernel structures and masking the presence of IDS files through the use of steganography and encryption [96]. Current systems that rely on these techniques can be easily defeated.

Some intrusion detection tools have addressed this problem by moving the IDS into the kernel [128, 117, 67]. This approach offers some resilience in the face of a compromise, but is not a panacea. Many OSes offer interfaces for direct kernel memory access from user level. If these interfaces are not disabled, kernel code is no safer from tampering by a privileged user than normal user-level code. On Linux systems, for example, user code can modify the kernel through loadable kernel modules [94],

`/dev/kmem`, [106, 102] and direct writes from I/O devices. Disabling these interfaces results in a loss of functionality, such as the inability to run programs, such as X11, that rely on them. We must also contend with the issue of exploitable bugs in the OS, a serious problem in our world of complex operating systems written in unsafe languages, where new buffer overflows are discovered with disturbing frequency.

In a host-based IDS, an IDS crash will generally cause the system to fail open. In a user-level IDS it is impossible for all system activity to be suspended if the IDS does crash, since the it relies on the operating system to resume its operation. If the IDS is only monitoring a particular application, it may be possible to suspend that application while the IDS is restarted. A critical fault in a kernel-based IDS will often similarly fail open. Since the IDS runs in the same fault domain as the rest of the kernel, this will often cause the entire system to crash or allow the attacker to compromise the kernel [116].

Unfortunately, when NIDSes do fall prey to an attack they often fail open as well. Consider a malfunction in an NIDS that causes the IDS to crash or become overloaded due to a large volume of traffic. This will virtually always cause the system to fail open until such time as the NIDS restarts [95]. Failing closed in an NIDS is often not an option as the network connection being monitored is often shared among many hosts, and thus suspending connectivity while the IDS restarted would amount to a considerable denial-of-service risk.

In a VMI-based IDS the host can be trivially suspended while the IDS restarts in case of a fault, providing an easy model for fail-safe fault recovery. In addition, because a VMI IDS offers complete mediation of access to hardware, it can maintain the constraints imposed by the operating system on hardware access even if the OS has been compromised, e.g. by disallowing the network card to be placed into promiscuous mode.

3.3 VMMs and VMI

The mechanism that facilitates the construction of a VMI IDS is the virtual machine monitor, the software responsible for virtualizing the hardware of a single physical machine and partitioning it into logically separate virtual machines. In this section, we discuss virtual machine monitors, what they do, how they are implemented and their level of assurance. We will also discuss the essential capabilities that a VMM must provide in order to support our VMI IDS architecture: isolation, inspection, and interposition.

3.3.1 Virtual Machine Monitors

A virtual machine monitor (VMM) is a thin layer of software that runs directly on the hardware of a machine. The VMM exports a *virtual machine* abstraction (VM) that resembles the underlying hardware. This abstraction models the hardware closely enough that software which would run on the underlying hardware can also be run in a virtual machine. VMMs virtualize all hardware resources, allowing multiple virtual machines to transparently multiplex the resources of the physical machine[51]. The operating system running inside of a VM is traditionally referred to as the guest OS, and applications running on the guest OS are similarly referred to as guest applications.

Traditionally, the VMM is the only privileged code running on the system. It is essentially a small operating system. This style of VMM has been a standard part of mainframe computers for 30 years, and recently has found its way onto commodity $x86$ PCs. Hosted VMMs like VMware [121, 112] have emerged that run a VMM concurrently with a commodity "host OS" such as Windows or Linux. In this setting, the virtual machine appears as simply another program running on the host operating system. Despite a radical difference from the users perspective, traditional and hosted VMMs differ little in implementation. In a hosted architecture the VMM merely leverages a third-party host OS to provide drivers, bootstrapping code, and other functionality common to VMMs and traditional operating systems, instead of being forced to implement all of its functionality from scratch.

VMMs have traditionally been used for logical server partitioning, and are supported for a wide range of architectures; for example, the IBM xSeries ($x86$ servers), pSeries (Unix), zSeries (mainframes), and iSeries (AS/400) all have VMMs available.

Recently, as hosted VMMs have appeared on the desktop, they have begun to find other applications such as cross-platform development and testing.

3.3.2 VMM Implementation

Although the specifics of a VMM's implementation are architecture-dependent, VMMs tend to rely on similar implementation techniques. Among these techniques is configuring the real machine so that virtual machines can safely and directly execute using the machine's CPU and memory. By doing this, VMMs can efficiently run software in the virtual machines at speeds close to that achieved by running them on the bare hardware [112]. VMMs can also fully isolate the software running in a virtual machine from other virtual machines, and from the virtual machine monitor.

A common way to virtualize the CPU is to run the VMM in the most privileged mode of the processor, while running virtual machines in less privileged modes. All traps and interrupts that occur while a virtual machine is running transfer control to the VMM. Attempts by the virtual machines to access privileged operations trap into the VMM; the VMM emulates privileged operations for the VM. In this architecture, the VMM can always control the virtual machine regardless of what the software in the virtual machine does.

Memory is commonly virtualized by keeping a virtual MMU for each virtual machine that reflects the VM's view of its address space. The VMM retains control of the real MMU, and maps each VM's physical memory in such a way that VMs do not share physical memory with each other, or with the VMM. Through this technique the VMM is able to create the illusion that each VM has its own address space that it fully controls. This also allows the VMM to isolate the VMs from one another and prevents them from accessing the memory of the VMM.

In addition to virtualizing the CPU and memory, the VMM intercepts all input/output requests from VMs to virtual devices and maps them to the correct physical I/O device. For memory-mapped I/O, the VMM only allows a virtual machine to see and access the particular I/O devices it is permitted to use.

3.3.3 VMM Assurance

Our argument for the security of a VMI IDS rests on the assumption that a VMM is difficult for an attacker to compromise. We base this assumption on the claim that a VMM is a simple-enough mechanism that we can reasonably hope to implement it correctly. We have several reasons for this claim. First, the interface to a VMM is significantly simpler, more constrained and well specified than that of a typically modern operating system. While the VMM is responsible for virtualizing all of the architecture, many portions, such as virtualization of the CPU, require little participation on the part of the VMM, since most instructions are unprivileged. Second, the protection model of a VMM is significantly simpler than that of a modern operating system. Everything inside the VMM is completely unprivileged with respect to the VMM, and the VMM has only to provide isolation, with no concerns about providing controlled sharing. Finally, although a VMM is an operating system, it is significantly simpler than standard modern operating systems. VMM's such as Disco [19] and Denali [126], which have both virtualized very complex architectures, have been built in on the order of 30K lines of code. This simplicity is attributable to the lack of a filesystem, network stack, and often, even a full fledged virtual memory system.[1] Some will point out that the small size and simplicity of a VMM do to its lack of a filesystem and network stack is misleading, since these facilities must ultimately be available to perform administrative functions such as logging and remote administration. However, this overlooks the fact that these activities are not part of the core VMM, but run in a completely different protection domain, typically in an administrative VM that is strongly isolated both from other VM's and from the secure kernel of the VMM. While there is a risk that this administrative VM(s) could be compromised, the compartmentalization provided by a VMM does a great deal to limit the extent of a compromise.

The small size and critical functionality of VMMs has led to a significant investment in their testing, validation, etc. Notable projects that have made strong claims for the security of VMMs include the Vax security monitor [64] and the NSA with

[1]This also applies to hosted VMMs as components such as the network stacks will not be utilized, and need not even be included in the host OS.

their Nettop [86] system. Nettop also relies on VMware Workstation for its VMM. Ultimately, since VMware is a closed-source product, it is impossible to verify this claim through open review.

3.3.4 Leveraging the VMM

Our VMI IDS leverages three properties of VMMs:

Isolation Software running in a virtual machine cannot access or modify the software running in the VMM or in a separate VM. Isolation ensures that even if an intruder has completely subverted the monitored host, he still cannot tamper with the IDS.

Inspection The VMM has access to all the state of a virtual machine: CPU state (e.g. registers), all memory, and all I/O device state such as the contents of storage devices and register state of I/O controllers. Being able to directly inspect the virtual machine makes it particularly difficult to evade a VMI IDS since there is no state in the monitored system that the IDS cannot see.

Interposition Fundamentally, VMMs need to interpose on certain virtual machine operations (e.g. executing privileged instructions). A VMI IDS can leverage this functionality for its own purposes. For example, with only minimal modification to the VMM, a VMI IDS can be notified if the code running in the VM attempts to modify a given register.

VMMs offer other properties that are quite useful in a VMI IDS. For example, VMMs completely encapsulate the state of a virtual machine in software. This allows us to easily take a *checkpoint* of the virtual machine. Using this capability we can compare the state of a VM under observation to a suspended VM in a known good state, easily perform analysis off-line, or capture the entire state of a compromised machine for forensic purposes.

3.4 Design

In this section we present an architecture for a VMI IDS system (shown in Fig. 1). First, we present the threat model. Next, we discuss the major components of our architecture and the design issues associated with these components. In the next section we will delve into the particulars of Livewire, a prototype VMI IDS system that implements this architecture.

3.4.1 Threat Model

Ideally, the guest OS will not be compromised, as we make some assumptions about the structure of the guest OS kernel to infer its state. If the guest OS is compromised this may result in some loss of visibility assuming the attacker modifies the guest OS in a way that misleads the VMI IDS about the true state of the host. However, even in this case some visibility will be maintained, and the VMI IDS will still be able to perform checks that make fewer assumptions about memory structure (such as naive signature scans) as well as maintaining access controls on devices, sensitive memory areas, etc.

We assume that the code running inside a monitored host may be totally malicious. We believe this model is quite timely as attackers are increasingly masking their activities and subverting intrusion detection systems through tampering with the OS kernel [53], shared libraries, and applications that are used to report and audit system state [66] (e.g. `tripwire`, `netstat`). We can only assume that if VMI-based IDSes sees wide spread deployment attackers will attempt to develop similar countermeasures.

All information that the IDS obtains from the monitored host must be considered "tainted," that is, containing potentially misleading or even damaging data (e.g. incorrectly formatted data that could induce a buffer overflow).

The VMI IDS may make assumptions about the structure of the guest OS to implement some IDS policies. This reliance should only imply that if OS structures are maliciously modified, it may be possible to evade policies that rely upon those structures, but should not affect the security of the IDS in any other way.

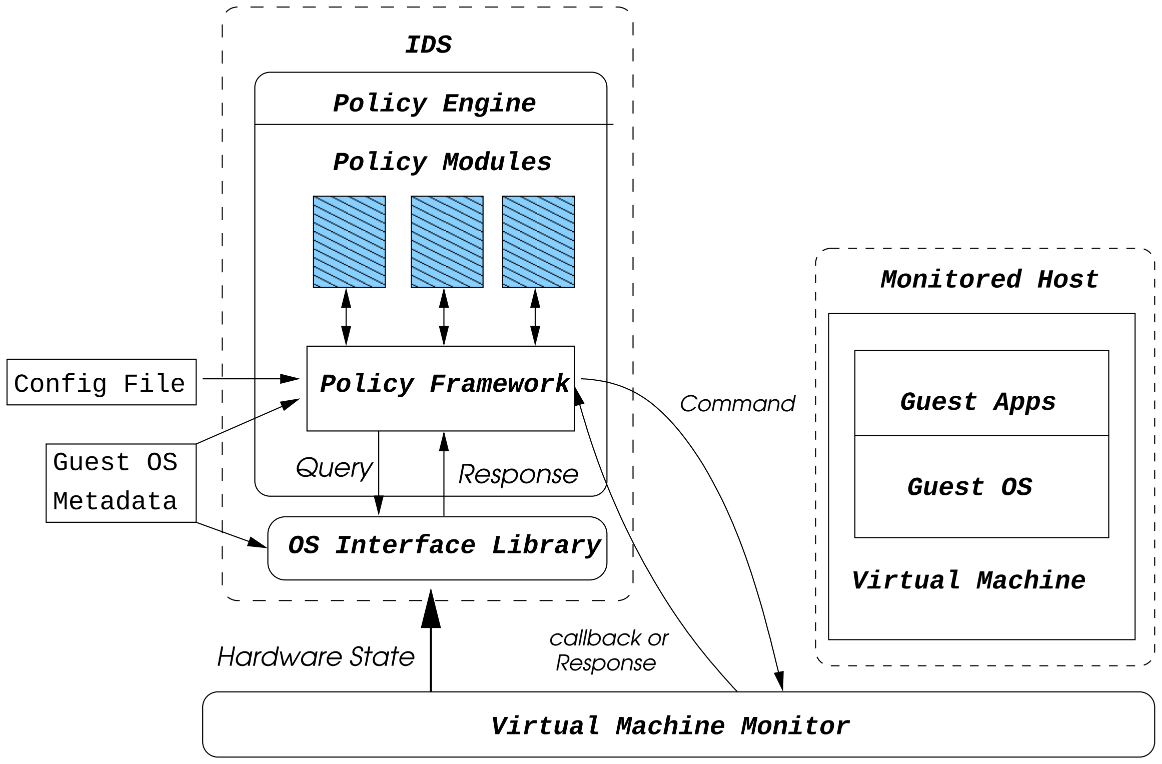

Figure 3.1: A High-Level View of our VMI-Based IDS Architecture: On the right is the virtual machine (VM) that runs the host being monitored. On the left is the VMI-based IDS with its major components: the OS interface library that provides an OS-level view of the VM by interpreting the hardware state exported by the VMM, the policy engine consisting of a common framework for building policies, and policy modules that implement specific intrusion detection policies. The virtual machine monitor provides a substrate that isolates the IDS from the monitored VM and allows the IDS to inspect the state of the VM. The VMM also allows the IDS to interpose on interactions between the guest OS/guest applications and the virtual hardware.

3.4.2 The Virtual Machine Monitor

As explained in section 3.3, the VMM virtualizes the hardware it runs on and provides the essential properties of isolation, inspection, and interposition. VMMs provide isolation by default; however, providing inspection and interposition for a VMI IDS requires some modification of the VMM. When adding these capabilities there are some important design trade-offs to consider:

- *Adding VMI functionality vs. Maintaining VMM simplicity.* We would like to minimize the changes required to the VMM to support a VMI IDS. Implementation bugs in the VMM can compromise its ability to provide secure isolation, and modifying the VMM presents the risk of introducing bugs. However, adding functionality to the VMM can provide significant benefits for the VMI IDS system as well. The ability to efficiently interpose on the MMU and CPU can allow the VMI IDS to monitor events that would otherwise be inaccessible. In confronting this issue in our prototype system, we provided additional functionality by leveraging existing VMM mechanisms. This strategy allowed us to expose a great deal of functionality to the VMI IDS, while minimizing changes to the VMM.

- *Expressiveness vs. Efficiency.* A VMM can allow a VMI IDS to monitor many types of machine events. Some types of events can be monitored with little or no overhead, while others can exact a significant performance penalty. Accessing hardware state typically does not incur any performance penalty in the VMM, so efficiently providing this functionality is purely a matter of making state available to the IDS with minimal copying. Trapping hardware events, such as interrupts and memory accesses can be quite costly because of their frequency. In our prototype system we sought to manage this overhead by only trapping events that would imply definite misuse (e.g. modification of sensitive memory that should never change at runtime). The overhead incurred for monitoring a particular type of event heavily depends on the particular VMM one is using.

A final issue to consider is VMM exposure. The VMI IDS has greater access to the VMM than the code running in a monitored VM. However, since we grant the

IDS access to the internal state of the VM we are potentially exposing the IDS, and by transitivity the VMM to attack. For this reason, it is important to minimize the VMM's exposure to the IDS. For example, communicating with the VMM through an IPC mechanism should be preferred to exporting internal hooks in the VMM and loading the IDS as a shared library. By isolating the IDS from the VMM, we reduce the risk of an IDS compromise leading to a compromise of the VMM. Compromising the IDS should at worst constitute a denial-of-service attack on the monitored VM. A compromise of the VMM is a catastrophic failure in a VMI IDS architecture.

The VMM Interface

The VMM must provide an interface for communication with the VMI IDS. The VMI IDS can send commands to the VMM over this interface, and the VMM will reply in turn. In our architecture, commands are of three types:

INSPECTION COMMANDS are used to directly examine VM state such as memory and register contents, and I/O devices' flags.

MONITOR COMMANDS are used to sense when certain machine events occur and request notification through an event delivery mechanism. For example, it is possible for a VMI to get notified when a certain range of memory changes, a privileged register changes, or a device state change occurs (e.g. Ethernet interface address is changed).

ADMINISTRATIVE COMMANDS allow the VMI IDS to control the execution of a VM. This interface allows the VMI IDS to suspend a VM's execution, resume a suspended VM, checkpoint the VM, and reboot the VM. These commands are primarily useful for bootstrapping the system and for automating response to a compromise. A VMI IDS is only given administrative control over the VM that it is monitoring.

The VMM can reply to commands synchronously (e.g. when the value of a register is queried) or asynchronously (e.g. to notify the VMI IDS that there has been a change to a portion of memory).

3.4.3 The VMI IDS

The VMI IDS is responsible for implementing intrusion detection policies by analyzing machine state and machine events through the VMM interface. The VMI IDS is divided into two parts, the *OS interface library* and the *policy engine*. The OS interface library's job is to provide an OS-level view of the virtual machine's state to facilitate easy policy development and implementation. The policy engine's job is purely to execute IDS policies by using the OS interface library and the VMM interface.

The OS Interface Library

VMMs manage state strictly at the hardware level, but prefer to reason about intrusion detection in terms of OS-level semantics. Consider a situation where we want to detect tampering with our `sshd` process by periodically performing integrity checks on its code segment. A VMM can provide us access to any page of physical memory or disk block in a virtual machine, but discovering the contents of `sshd`'s code segment requires answering queries about machine state in the context of the OS running in the VM: "where in virtual memory does `sshd`'s code segment reside?", "what part of the code segment is in memory?", and "what part is out on disk?"

We need to provide some means of interpreting low-level machine state from the VMM in terms of the higher-level OS structures. We would like to write the code to do this once and provide a common interface to it, instead of having to re implement this functionality for each new policy in our IDS. Our solution must also take into account variations in OS structure such as differences in OS versions, configurations, etc.

The OS interface library solves this problem by using knowledge about the guest OS implementation to interpret the VM's machine state, which is exported by the VMM. The policy engine is provided with an interface for making high-level queries about the OS of the monitored host. The OS interface library must be matched with the guest OS; different guest OSes will have different OS interface libraries.

Some examples of the type of queries that the OS interface library facilitates are:

"give me a list of all the processes currently running on the system," or "tell me all the processes which are currently holding raw sockets." The OS interface library also facilitates queries at the level of kernel code, similar to the queries that one might give to gdb like "show me the contents of virtual memory from x to y in the context of the login process," or "display the contents of task structure for the process with PID 231."

The Policy Engine

At the heart of any intrusion detection system is the policy engine. This component interprets system state and events from the VMM interface and OS interface library, and decides whether or not the system has been compromised. If the system has been compromised, the policy engine is responsible for responding in an appropriate manner. For example, in case of a break-in, the policy engine can suspend or reboot the virtual machine, and report the break-in. Since the focus of our work has been studying VMI as a platform for IDS, we have focused on implementing variations on mainstream HIDS style policies [84] such as burglar alarms, misuse detectors and integrity checkers. A policy engine implementing complex anomaly detection and other, more exotic techniques can also be supported in this architecture.

3.5 Implementation

To better understand the implementation difficulties, performance overhead, usability, and practical effectiveness of our VMI architecture, we built Livewire, a prototype VMI IDS. For our VMM we used a modified version of VMware Workstation [121] for Linux $x86$. Our OS library was built by modifying Mission Critical's `crash` [87] program. Our policy engine consists of a framework and modules written in the Python programming language [52]. Each of these components runs in its own process in Linux, our host OS.

3.5.1 VMM

We used a modified version of VMware Workstation for Linux to provide us with a virtual machine monitor capable of running common $x86$-based operating systems. To support VMI, we added hooks to VMware to allow inspection of memory, registers, and device state. We also added hooks to allow interposition on certain events, such as interrupts and updates to device and memory state.

The virtual machine monitor supports virtual I/O devices that are capable of doing direct memory access (DMA). These virtual devices can use DMA to read any memory location in the virtual machine. We used this virtual DMA capability to support direct physical memory access in the VMM interface. We accomplished this with minimal changes to the VMM.

As part of this virtualization process, the VMM shadows the page tables of the physical machine, allowing the monitor to enforce more restrictive protection of certain memory pages. An example of how this functionality can be applied is the copy-on-write page sharing of the Disco virtual machine monitor [19]. We used this mechanism to write protect pages and provide notification if the VM attempted to modify a protected page.

Interactions with virtual I/O devices such as Ethernet interfaces are intercepted by the VMM and mapped actual hardware devices in the course of normal VMM operation. We easily added hooks to notify us when the VM attempted to change this state. Hooks to inspect the state of virtual devices such as the virtual Ethernet card were also added.

Adding anything to a VMM is worrisome as it means changing low-level code that is critical to both the correctness and performance of the system. However, we found we could support the required interposition and inspection hooks with only minor changes to VMware by leveraging functionality required to support basic virtualization. The functionality that we leveraged is common to most VMMs, thus, we believe that adding interposition support to other VMMs should be straightforward.

3.5.2 VMM Interface

The VMM interface provides a channel for the VMI IDS processes to communicate with the VMware VMM process. This interface is composed of two parts: first, a Unix domain socket that allows the VMI IDS to send commands to, and receive responses and event notifications from, the VMM; and second, a memory-mapped file that supports efficient access to the physical memory of the monitored VM.

In Livewire, when an event occurs, the VM's execution is suspended until the VMI IDS responds with an administrative command to continue. We opted for this model of event notification as our policies only use monitor commands for notification of definite misuse, which we handle by halting as a matter of policy. For other policies, such as monitoring interrupts to do system call pattern-based anomaly detection [73], an event delivery model where the VM does not suspend could also be supported.

3.5.3 OS Interface Library

Our OS interface library was built by modifying the Linux crash dump examination tool `crash` [87] to interpret the machine state exported by the VMM interface. The critical intuition here is that in practice there is very little difference between examining a running kernel through `/dev/kmem` with a crash dump analysis tool from within a guest OS, and running the same tool outside the guest OS. The VMM exports an interface similar to `/dev/kmem` that provides access to the monitored host's memory in the form of a flat file.

Information about the specifics of the kernel being analyzed (the symbol table, data types, etc.) are all derived from the debugging information of the kernel binary by `crash` or `readelf`. All other problems related to dealing with differences in kernel versions were dealt with by `crash`.

The IDS communicates with the OS interface library over a full-duplex pipe, using it both to send and receive their responses. The command set and responses were simply those exported by `crash`.

3.5.4 Policy Engine

The policy engine consists of two pieces: the *policy framework*, a common API for writing security policies, and the *policy modules* that implement actual security policies. The policy engine was built entirely using Python.

Policy Framework

The policy framework allows the policy implementer to interact with the major components of the system with minimal hassle by encapsulating them in simple high level APIs. The policy framework provides the following interfaces:

OS Interface Library: The OS interface library presents a simple request/response to the module writer for sending commands to the OS interface library, and receiving responses that have been marshaled in native data formats. Tables containing key-value pairs that provide information about the current kernel (e.g. the kernel's symbol table) are also provided.

VMM Interface: The VMM interface provides direct access to the VM's physical address space and register state. Physical memory space is accessed as a single large array. This provides an easy way for the programmer to search the VM's memory, or to calculate secure hashes of portions of memory for performing integrity checks.

Monitor commands are used by registering callbacks for events that a policy module wants to be notified of, e.g. a write to a range of memory, or modification of the NIC's MAC address. Callbacks can also be registered for VM-level events, such as the VM rebooting or powering down. Finally, the VM interface exports administrative commands that allow policy modules to suspend, restart, and checkpoint the VM.

Livewire Front End: The front end code is responsible for bootstrapping the system, starting the OS interface library process, loading policy modules, and running policy modules in concert. Interfaces are provided for obtaining configuration information, reporting intrusions, and registering policy modules with a common controller.

Policy Modules

We have implemented six sample security policy modules in Livewire. Four modules are *polling modules*, modules that run periodically and check for signs of an intrusion. The other two are *event-driven modules* that are triggered by a specific event, such as an attempt to write to sensitive memory.

Each policy module is an individual Python module (i.e. a single file) that leverages the policy framework. Policy modules can be run stand-alone or in concert with other policy modules.

We found writing modules using the Livewire policy framework a modest task. Most of the polling modules were written in less than 50 lines of Python, including comments. Only the user program integrity detector (see Section 3.6.1) required more code than this, at 130 lines of Python. The event-driven modules were also quite simple, each one requiring roughly 30 lines of code.

A detailed discussion of the policy modules we implemented is given in the next section.

3.6 Example Policy Modules

In this section we present a variety of policy modules that we have implemented in Livewire. Our goal with these policies was not to provide a complete intrusion detection package, nor was it to experiment with novel policy design. Instead we chose policies as simple examples that illustrate more general paradigms of policy design that can be supported by this architecture.

3.6.1 Polling Policy Modules

Polling modules periodically check the system for signs of malicious activity. All of our polling modules possess close HIDS analogues, as they only leverage the VMM for isolation and inspection. The former is not essential to their function, and the latter can be provided by normal OS mechanisms for accessing low-level system state. In fact, we initially developed some of our polling checkers by running Livewire on the guest OS it was monitoring and inspecting system state through `/dev/kmem`.

Lie Detector

Attackers often achieve stealth by modifying the OS kernel, shared libraries, or user-level services to mask their activities. For example, suppose an intruder wants to modify the system to hide malicious processes. The attacker can modify `ps`, modify shared libraries, or modify the `/proc` interface that `ps` uses to find out about currently running processes. These modification can lead to inconsistencies between the kernel, or hardware view of the system, and the view provided by user-level programs. A variety of HIDS programs detect intruders by noting these inconsistencies [83].

The lie detector module works by directly inspecting hardware and kernel state, and by querying the host system through user-level programs (e.g. `ps`, `ifconfig`, `netstat`) via a remote shell. If it detects conflicts between these two views (i.e. the system is lying), it reports the presence of malicious tampering. This technique has the nice property that it does not matter what part of the system the intruder modified to elicit the malicious behavior. One concern we had when building this checker was ensuring that the views we compared were from the same point in time. In practice, we did not encounter any problems with skew that led to false positives.

User Program Integrity Detector

Checking the integrity of a program binary on disk (ala. `tripwire` [65]) does not ensure that the corresponding in memory image of that program has not been modified (e.g. via `ptrace` [9]). Our integrity checker attempts to detect if a running user-level program has been tampered with by periodically taking a secure hash of the immutable sections (`.text`, etc.) of a running program, and comparing it to a known good hash. This approach is particularly well suited to securing long running programs such as `sshd`, `inetd`, and `syslogd` that are continuously present in memory.

One complication we encountered while implementing this checker was is that portions of large programs may be paged out to disk, or simply never demand-paged into memory in the first place. Our current implementation deals with this issue by taking per-page hashes and only examining the portion of a program that is in memory.

Signature Detector

Scanning the file system for the presence of known malicious program based on a known "signature" substring of the program is a popular intrusion detection technique. It is employed by anti-virus tools as well as root-kit detection tools like `chkrootkit` [88].These tools leverage the fact that most attackers do not write their own tools, but instead rely on a relatively small number of publicly available rootkits, backdoors, Trojan horses and other attack tools. Popular examples include "subseven," "backorifice," and "netbus" Trojan horses for Windows, or the "adore" and "knark" kernel backdoors under Linux. Most Unix HIDS systems that look for signature strings only scan a few selected files for signatures. Our signature detector performs a scan of all of host memory for attack signatures. This more aggressive approach requires a more careful selection of signatures to avoid false positives. It also means that malicious programs that have not yet been installed may also be detected, e.g. in the filesystem buffer cache.

Raw Socket Detector

Raw sockets have legitimate applications in certain network diagnostic tools, but they are also used by a variety of "stealth" backdoors, tools for ARP-spoofing, and other malicious applications that require low-level network access. The raw socket detector is a "burglar alarm" [84] style policy module for detecting the use of raw sockets by user-level programs for the purpose of catching such malicious applications. This is accomplished by querying the kernel about the type of all sockets held by user processes.

3.6.2 Event Driven Policy Modules

Event-driven checkers run when the VMM detects changes to hardware state, such as a write to a sensitive location in memory. At startup, each event-driven checker registers all of the events it would like to be notified of with the policy framework. At runtime, when one of these events occurs, the VMM relays a message to the policy framework. The policy framework runs the checker(s) which have registered to receive the event. In a purely intrusion-detection role, event-driven checkers can simply report the event that has occurred according to their policy, and allow the virtual machine to continue to run. The VMM can also be directed to suspend on events, thus allowing the policy module to also serve as a reference monitor that regulates access to sensitive hardware.

Memory Access Enforcer

Modern computer architectures generally allow programs running in ring 0 (i.e. the kernel) to render certain sections of memory read-only, such as their text segment and read-only data, as a standard part of their the memory protection interface. However, they also allow anything else running in ring 0 to disable these access controls. Thus, while these mechanisms are useful for detecting accidental protection violations due to faulty code, they are relatively useless for protecting the kernel from tampering by other malicious code that is running in ring 0 (for example a kernel backdoor).

Detecting tampering with an OS code segment can be an useful mechanism for discovering the presence of malicious code, and preventing its installation into the kernel proper. Our kernel memory enforcer works by marking the code section, `sys_call_table`, and other sensitive portions of the kernel as read-only through the VMM. If a malicious program, such as a kernel back door tries to modify these sections of memory, the VM will be halted and the kernel memory protection enforcer notified. Several HIDS tools [117, 96] attempt to detect modifications to the `sys_call_table` and system call code through the use of integrity checking. However, this approach is far less attractive due to its lack of immediacy (and inability to prevent attacks) as well as the additional overhead it incurs. Sensitive registers like the `idtr` can also be locked down.

NIC Access Enforcer

The NIC Access Enforcer prevents the Ethernet device entering promiscuous mode, or being configured with a MAC address which has not been pre-specified. Using this module we can prevent variety of common misuses of the NIC to be detected and prevented. In spite of its simple functionality the NIC module provides a useful policy enforcement tool. It is more robust to attack than normal host-based solutions, and not susceptible to evasion, as is a problem with remote promiscuous mode detection solutions [30].

3.7 Experimental Results

In this section we present an experimental evaluation of our Livewire prototype. Our evaluation consists of two parts. First, we test the effectiveness of our security policies against some common attacks. This portion of our evaluation was undertaken to ensure that our policies worked in practice, and to gain experience with utilizing Livewire against real attacks. The second part of our evaluation consisted of testing the performance overhead of Livewire on several sample work loads.

Our target host consisted of virtual machine with a 256 MB allocation of physical memory and a 4 GB virtual disk, running a relatively standard installation of Debian GNU/Linux. The virtual machine monitor (a modified version VMware Workstation for Linux version 3.1) was run on a 1.8 GHz Pentium IV laptop with 1 GB of physical memory, running Debian GNU/Linux as a host OS.

3.7.1 Sample Attacks

Our test suite of sample attacks consisted of kernel- and user-level rootkits and backdoors [102, 40, 118, 25, 111] Trojan horses [78], packet sniffers [110, 78], and a worm [26]. All test attacks were obtained from public sources and were modified only as necessary for configuration, or for adaptation to our kernel. A selection of nine attacks that we feel provides a good representative cross-section has been chosen for our discussion. A summary of the attacks and our results is depicted in Table 3.7.

Name	Description	nic	raw	sig	int	lie	mem
cdoor	Stealth user level remote backdoor		D				
t0rn	Precompiled user level rootkit			D		D	
Ramen	Linux Worm			D			
lrk5	Source based user level rootkit	P		D	D	D	
knark-0.59	LKM based kernel backdoor/rootkit			D		D	P
adore-0.42	LKM based kernel backdoor/rootkit			D		D	P
dsniff 2.4	All-purpose packet sniffer for switched networks	P					
SUCKIT	/dev/kmem patching based kernel backdoor			D		D	P

Table 3.1: Results of Livewire policy modules against common attacks. Within the grid, "P" designates a prevented attack and "D" a detected attack.

3.7.2 Detection Results

To collect our detection results we set up our system under Livewire and then downloaded, installed, and ran each attack. Once an attack had been tested the system was rolled back to a clean state before another attack was attempted.

Our **Signature Scanner** ("sig") was able to detect all of the attacks we provided signatures for. This was a predictable result; the only interesting surprise was that it often detected the presence of malicious code before it had been run, based on the presence of the signature in the buffer cache (when we examined them on disk), in the memory of `ssh` (as they were being downloaded), or in memory once activated. Initially, we encountered several false positives due to overly general signatures. For example, the string "adore" works fine as a signature for a file system-based checker, such as `chkrootkit`, but is too general a signature for a scanner looking at all of memory. Our **Raw Socket Detector** ("raw") raw socket detector detected the presence of `cd00r` and a similar stealth backdoor we included in our test attack suite. Our **Lie Detector** ("lie") modules proved especially effective against rootkits both at kernel and user level since they all provided functionality to hide processes. The **User Program Integrity Checker** ("int") was able to detect the presence of t0rn and lrk5 based on their use of backdoored version of `inetd` and `sshd`. We also found it also effective in detecting backdoored versions of `syslog`. Our **Memory Protection Enforcer** ("mem") was able to detect and prevent the installation of all of our kernel backdoors. **knark** and **adore** were stopped by blocking their attempt to modify

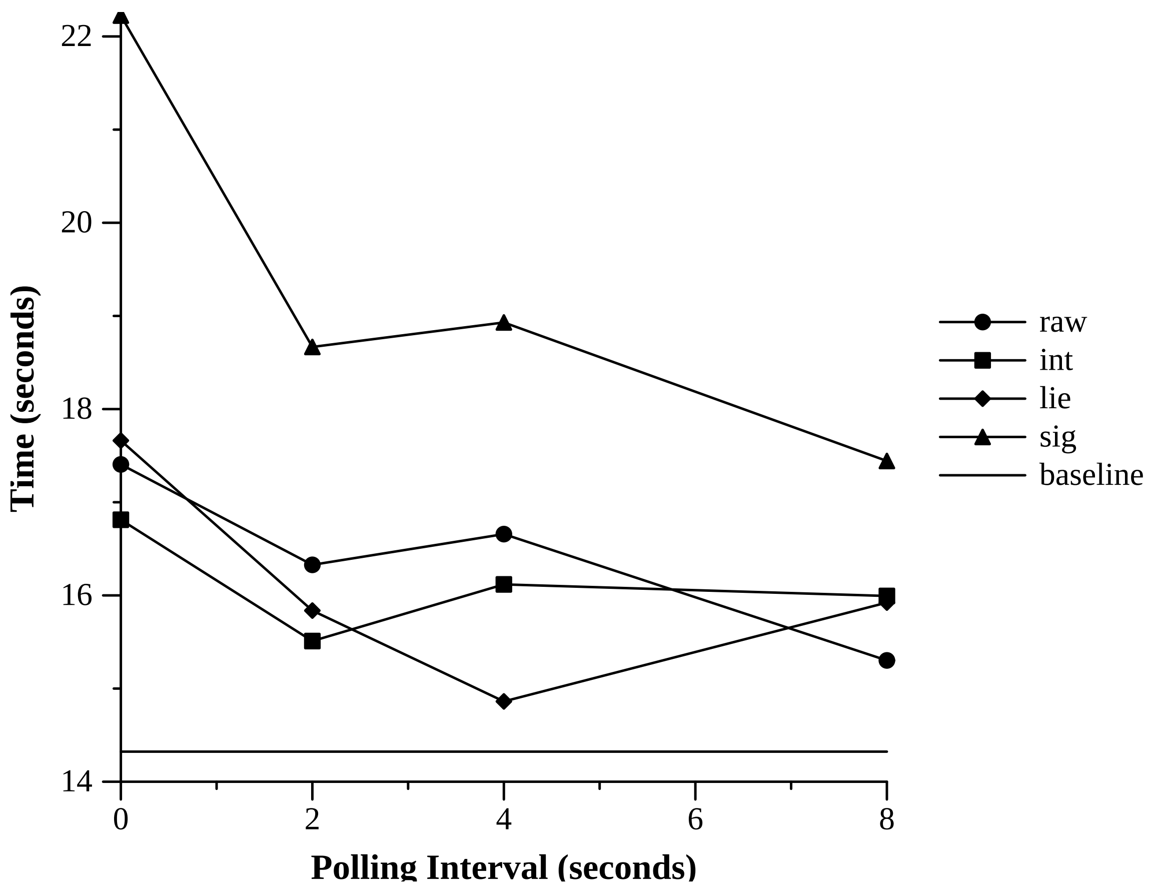

Figure 3.2: Performance of Polling Policy Modules

sys_call_table. SUCKIT was stopped by blocking its attempt to modify the interrupt dispatch table. Our **NIC access enforcer** ("nic") was trivially able to detect and prevent the packet sniffers in our test attack suite from operating, based on their reliance on running the NIC in promiscuous mode.

3.7.3 Performance

To evaluate the performance of our system we considered two sample work loads. First, we unzipped and untarred the Linux 2.4.18 kernel to provide a CPU-intensive task. Second, we copied the kernel from one directory to another using the `cp -r` command to provide a more I/O intensive task.

We used the first workload to evaluate the overhead of running event-driven checkers in the common case when they are not being triggered. As expected, no measurable overhead was imposed on the system.

We used our second workload to evaluate the overhead associated with running our checkers at different polling intervals. The results are shown in figure 2. The baseline measurement shows performance of the workload without Livewire running. Our performance results were somewhat surprising to us. We had expected the time taken by polling modules as a function of the total time to decrease linearly as the cost of checking was amortized over the total running time of the the workload. While this was generally the trend, we found that as the polling interval decreased the interactions with the workload became more erratic.

3.8 Weaknesses and Attacks

In this section we present avenues for attacking and evading VMI-based IDS architectures and explore approaches to addressing these problems. Some of the issues that we present are unique to the problem of building a VMI IDS; other are more general issues that arise in attempts to build secure systems with VMMs.

3.8.1 Attacking the VMM

Indirect Attacks

VMMs may provide interfaces accessible from outside of a VM that provide an avenue of attack. For example, a hosted VMM might be running on a host OS with a remotely exploitable network stack, or application-level network service. In a VMI IDS, the threat of indirect attacks can be minimized by using a traditional VMM that possesses no network stack or by disabling the network stack in a hosted environment.

Detecting the VMM

The first step in evading a VMI IDS is detecting its presence. A significant hint that a VMI IDS may be present is the presence of a VMM. Unfortunately, masking the presence of a VMM is almost impossible due to differences in the relative amount of time that I/O operations, device access, virtualized instructions, and other processes take as compared to a non-virtualized interface [51]. Hiding these disparities is impractical and not worth the little bit of additional stealth it would provide the IDS. Timing can also leak information that could betray the presence of a VMI IDS and its activities.

Directly Subverting the VMM

The VMM may expose the VMI IDS to direct attack in two ways: flaws in the design of the VMM or flaws in its implementation. The former problem can occur when VMMs are not designed with malicious guest code in mind. For example, virtual environments like User-Mode Linux are sometimes designed with debugging or application compatibility as their primary application and do not provide secure isolation. The latter problem occurs when there is an error in the VMM code, or code the VMM relies upon. We conjecture that such errors would most likely be found in device driver code leveraged by virtual devices. While secure VMMs have been built with malicious users in mind, device drivers are often less paranoid about sanitizing their inputs, and thus can be subject to attack [13]. The VMM can attempt to deal with this issue defensively by judiciously checking and sanitizing data flowing from virtual devices to device drivers. This helps to minimize the risk of these inputs compromising the device driver. All devices drivers used with a VMM should be carefully screened.

Attacking the VMM through the IDS

The presence of the VMI IDS introduces another avenue for attacking the VMM. Fortunately, the VMI IDS requires minimal privilege beyond its ability to manipulate the guest VM, so that the impact of an IDS compromise on the VMM can be mitigated by running the IDS in its own VM, or by isolating it from the VMM through some other mechanism.

3.8.2 Attacking the IDS

Fooling the OS Interface Library

The OS interface library relies on meta-data gleaned from a kernel binary or other sources to interpret the structure of the OS. If an attacker can modify the structure of the guest OS so that it is inconsistent with the meta-data that the OS interface library possess, he can fool the OS interface library about the true state of the system. This style of attack is used against kernel modules that attempt to detect tampering with the `sys_call_table` through integrity checking [102]. To subvert these modules, attackers modify the interrupt dispatch table so that the kernel uses a different system call table altogether, while the module continues to check a system call table that is no longer in use. The problem of maintaining a consistent view of the system is fundamental to the VMI-based IDS approach. Livewire attempts to counter this type of attack through the memory access enforcer by disabling the attacker's ability to modify memory locations and registers that could allow sensitive kernel structures to be relocated, thus fooling the OS interface library. There are many sensitive mutable kernel data structures that we do not yet protect that could present an avenue for attack. We have simply tried to "raise the bar," and prevent the most obvious of cases of this class of attack. Finding better methods for identifying and enforcing the static and dynamic invariants that a VMI IDS relies upon seems an important area for further study.

Compromising the OS Interface Library

The OS interface library is the VMI IDS's point of greatest exposure to potentially malicious inputs. Because of this it is vital to carefully sanitize inputs, and treat all data gleaned from the virtual machine by direct inspection as tainted. The potential for problems in this part of the system is especially apparent in our Livewire prototype. The OS interface libraries are based on crash dump analysis tools written in C, thus presenting an ideal opportunity for a buffer overflow. Another means of attacking the OS interface library is by modifying kernel data structures to violate invariants that the OS interface library assumes. For example, introducing a loop into a linked list in the kernel that the OS interface library will read (e.g. a list of file descriptors) could induce resource exhaustion, or simply cause the OS interface library to wedge in a loop. The OS interface library must not assume any invariants about the structure of its inputs that are not explicitly enforced through the VMM. Given the potentially complex nature of the OS interface library, it seems advisable to isolate it from the policy engine and give it minimal privilege. In Livewire, this is approximated by running the OS interface library in a separate process, with only enough privilege to inspect the memory and registers of the monitored VM. If the OS interface library hangs, the policy engine can kill and restart it.

Compromising the Policy Engine

The extent to which the policy engine is vulnerable to compromise is dependent on the policies and implementation of the policy engine. We have taken several steps in our Livewire prototype to reduce the risk of a policy engine compromise:

- **Sanitize Inputs:** The need to carefully check and sanitize inputs from the guest OS cannot be emphasized enough. Inputs that come from the VMM interface and OS interface library should also have sanity checks applied to them.
- **A High-Level Policy Language:** Building IDSes that utilize a high-level policy language is a proven technique for building flexible, extensible NIDSes [92]. VMI IDSes also realize these benefits with a high-level policy language. Additionally, high-level policy languages also reduce the possibility of a total compromise due to memory safety problems. A high-level language like Python is especially well suited for doing pattern matching, manipulating complex data types, and other operations that are frequently useful for introspection. This expressiveness and ease of use allows policies to be written in a concise and easy-to-understand manner that minimizes errors.
- **Failing Closed:** In Livewire, the VMM can suspend on the last synchronous event that occurred and will not continue until explicitly instructed by the

IDS. This means that even if the policy engine crashes, protected hardware interfaces will still not be exposed. This type of fail-closed behavior is always recommended when a VMI IDS is also being used as a reference monitor.

- **Event Flow Control:** In the case when Livewire cannot keep up with queued asynchronous events, the VMM can suspend until Livewire can catch up. Unlike an NIDS which cannot necessarily stem the flow of traffic [92], it is easy to stem the flow of events to the VMI IDS.

- **Avoiding Wedging with Timers:** In Livewire, the polling module are run serially by a single thread of control. This introduces the risk that a bug in one policy module could cause the entire IDS to hang. We have tried to address this problem in two ways. First, all of our policy modules are written defensively, attempting to avoid operations that could hang indefinitely, and using timers to break out of these operations when necessary. Second, each policy module is only given a set amount of time to complete its task, and will be interrupted if it exceeds that limit, so that the next module can run.

3.9 Related Work

Classical operating system security issues such as confinement and protection have been studied extensively in traditional VMMs. In previous years thorough studies of these problems have been presented for VM/370 [100, 48, 47, 46] and the Vax Security Kernel [64]. The most recent implementation study of a security kernel can be found in work on the Denali isolation kernel[126]. A recent application of VMMs for pure isolation can be found in the NSA's nettop [86] architecture.

VMMs have also become a popular platform for building honey pots [103]. Often a network of virtual machines on a single platform will be built to form a honey net, providing a low-cost laboratory for studying the behavior of attackers.

The idea of collocating security services with the host that they are monitoring, as we study in this work, has also seen attention in the ReVirt [31] system, which facilitates secure logging and replay by having the guest operating system (the OS running inside the VM) instrumented to work in conjunction with the VMM.

Chen et al. [22] proposed running code in a VM to discover if it is malicious before proceeding with its normal execution. This idea is similar to the application of VMs to fault tolerance explored by Bressoud and Schneider in their Hypervisor [18] work.

Goldberg's work on architectural requirements for VMMs [50] and his survey of VMM research up to 1974 [51] are the standard classic works on VMMs. More recent noteworthy work on VMM architectural issues can be found in Disco [19], and in work on virtualizing I/O [112] and resource management [123] in VMware.

Also relevant to the topic of VM introspection is work on whole-machine simulation in SimOS [97], which also looked at the issues involved in instrumenting virtual hardware and extrapolating guest operating system state from hardware state.

3.10 Future Work

There are still many significant questions to be addressed about how VMI-based intrusion detection systems can best be implemented and used.

Livewire has taken an extremely conservative approach to introspection by primarily engaging in passive checks that incur no visible impact on system performance. This decision allowed Livewire to be implemented with only minimal changes to the virtual machine monitor. However, the cost of this was that monitoring frequent asynchronous events, e.g. all system calls, may be quite performance intensive. Our current architecture could support frequent asynchronous checks, such as monitoring and processing system call, and supporting lightweight data watchpoints with relative efficiency via. hard coding the functionality to log these events directly into the monitor, then offloading the processing of these logs to the policy engine. However, this approach seems somewhat inflexible. We believe a more promising approach would involve support for providing a small, safe and extensible mechanism for efficiently filtering architecture events in the VMM, in much the same fashion that current OSes provides this functionality for filtering packets via BPF.

In Livewire we made the choice to leverage the `crash` program in order to provide us with an OS interface library. This provided the functionality to experiment with a wide range of policies while minimizing implementation time. However, given the OS interface libraries exposure to attack it would be desirable to have a dedicated OS interface library of significantly smaller size, ideally written in a safe language. Another factor deserving further study in the OS interface library is that of concurrency. How can system kernel state be safely observed in the presence of constant updates to kernel state? How should the OS interface library respect OS locking primitives?

Other IDS tools can benefit from the capability of a VMM to allow secure collocation of monitoring on the same machine as the host, even without the use of introspection. HIDS techniques such as filesystem integrity checking could easily be moved outside of the host for better isolation. Conversely, NIDSes could be moved onto the same platform as the host, thereby distributing the load of performing packet analysis to end hosts, and potentially facilitating the use of more complex policies. Finally, the benefits of isolating protection mechanisms from the host has received little attention. Moving distributed firewalls as described by Ioniddis et. al . [60] outside of the host seems like an obvious application for this mechanism. An isolated keystore is another natural application of this mechanism.

Chapter 4

Overshadow Data Protection

4.1 Introduction

Commodity operating systems are ubiquitous in home, commercial, government, and military settings. Consequently, these systems are tasked with handling all manner of sensitive data, from individual passwords and crypto keys, to databases of social security numbers, to sensitive documents and voice traffic.

Unfortunately, the security provided by commodity operating systems is often inadequate. Trusted OS components include not just the kernel but also device drivers and system services that run with privilege (*e.g.*, daemons that run as root in Linux). These components generally comprise a large body of code, with broad attack surfaces that are frequently vulnerable to exploitable bugs or misconfigurations. Once such privileged code is compromised, an attacker gains complete access to sensitive data on a system. While some facets of security in these systems will continue to improve, we believe competitive pressures to provide richer functionality and retain compatibility with existing applications will keep the complexity of such systems high, and their assurance poor.

To ameliorate this problem, many have attempted to retrofit higher-assurance execution environments onto commodity systems. Previous efforts have explored executing applications handling sensitive data in separate virtual machines [41, 114, 37], using secure co-processors [34], or changing the processor architecture to introduce orthogonal protection mechanisms that protect application data from the OS [32, 59, 70, 76, 105]. Unfortunately, these generally demand major changes in the way that applications are written [34, 37, 70, 74, 107] and used [37, 41], and how OS resources are managed [41, 114]. Such radical departures pose a substantial barrier to adoption.

We offer an alternative in a system called *Overshadow*. Overshadow protects legacy applications from the commodity operating systems running them. Unlike other approaches, it requires no changes to existing operating systems or applications, nor any additional hardware support. Instead, it works by extending the isolation capabilities of the virtualization layer to allow protection of entities inside a virtual machine.

Overshadow adds this protection through a novel technique called *multi-shadowing* which leverages the extra level of indirection offered by memory virtualization in a virtual machine monitor (VMM). Conceptually, a typical VMM maintains a one-to-one mapping from guest "physical" addresses to actual machine addresses. Multi-shadowing replaces this with a one-to-many, context-dependent mapping, providing multiple views of guest memory. Overshadow leverages this mechanism to present an application with a cleartext view of its pages, and the OS with an encrypted view, a technique we call *cloaking*. Encryption-based protection allows resources to remain accessible to the OS, yet secure, permitting it to manage resources without compromising application privacy or integrity.

Cloaking is a low-level primitive that operates on basic memory pages. However, nearly all higher-level application resources – including code, data, files, and even IPC streams – are already managed as memory-mapped objects by modern operating systems, or can be adapted as such. As a result, cloaking is sufficiently general to protect all of an application's major resources.

Using cloaking to protect a legacy application running on an unmodified OS requires some changes to the normal execution environment. To accommodate these changes while maintaining compatibility, Overshadow introduces a *shim* at load time into the address space of each cloaked application to mediate all communication with the OS. With assistance from the VMM, the shim interposes on events such as system calls and signal delivery, modifying their semantics to enable safe resource sharing between a cloaked application and an untrusted OS.

The next section presents our design goals and threat model for Overshadow. Section 4.3 reviews virtualized memory systems, and describes extensions to support multi-shadowing and cloaking. Section 4.4 introduces the challenges that arise when applying cloaking to protect real applications, and provides an overview of the Overshadow architecture. Section 4.5 describes how the shim and VMM adapt applications for a cloaked environment, and Section 4.6 explains how particular system calls are mediated. Section 4.7 discusses how cryptographic metadata is managed and stored, to protect against reordering and replay attacks. Section 4.8 evaluates our implementation in the VMware VMM using large unmodified applications running on an unmodified Linux kernel. Section 4.9 discusses future work. Related work is examined in Section 4.10.

4.2 Design Goals

Overshadow offers a last line of defense for application data in the event of an OS compromise. We begin with a discussion of why Overshadow targets whole-application protection, and the threats it attempts to address.

4.2.1 Whole-Application Protection

We were motivated to build a practical system that could be adopted easily, deployed incrementally, and used for diverse applications. As a result, we designed Overshadow to protect entire existing applications *in situ* in existing commodity operating systems. This approach has several advantages:

Ease of Adoption. Previous work on protecting applications requires partitioning an application into protected and unprotected parts – forcing developers to modify their applications heavily [37, 107] or port to a new OS [114]. Changes to how software is packaged and used may also be required [41, 114].

Support for Diverse Applications. Solutions for providing higher assurance are often restricted to a limited set of applications or data, such as passwords [37, 34]. However, sensitive data is remarkably diverse, from databases of credit card numbers, to files containing medical patient information. Sensitive data in real applications frequently doesn't lend itself to being placed in a neat separate container, and restructuring applications is often impractical.

Incremental Path to Higher Assurance. Even after taking the operating system out of the application's trusted computing base, large, complex applications will still have significant assurance concerns. Refactoring applications into more-critical and less-critical pieces running in separate protection domains [37, 107] is ultimately a compelling goal. Overshadow provides an incremental path to achieving this, as cloaking can be used for whole application protection as well as fine-grained compartmentalization.

4.2.2 Threat Model

Overshadow prevents the guest operating system from reading or modifying application code, data and registers, but makes no attempt to provide availability in the face of a hostile OS. All non-application access to cloaked data, including DMA from virtual I/O devices, only reveals the data in encrypted form. Data secrecy, integrity, ordering and freshness are protected up to the strength of the cryptography used. If the OS or other hostile code tries to modify encrypted data, the application will be terminated.

Control transfers to and from a cloaked application are permitted only at well-defined entry and exit points through mechanisms such as system calls and signal delivery. Application registers are also protected from the OS, and are saved and restored securely upon entry and exit from an application's execution context. Overshadow can also protect information shared between cloaked applications via the file system, shared memory or other forms of IPC.

A malicious kernel can still observe an application's memory access patterns, and measure the time that application code sections take to complete. In extreme cases, such side channel information can leak private information, including application crypto keys [68]. Overshadow does not protect against these side-channel attacks. However, most contemporary cryptographic application code (such as OpenSSL) is designed to resist side channel attacks.

Security is ultimately limited by the application being protected. Logical or semantic weaknesses in the application, such as an exploitable buffer overflow, or a DumpMyMemory

c

ommand, could allow a malicious OS to fool it into revealing its data, or otherwise exploit it. The implications of maliciously changing the behavior of seemingly innocuous parts of the system call API, such as those for managing identity and concurrency, are still largely unstudied.

Assurance in Overshadow is ultimately limited by the VMM. While our current implementation uses the VMware VMM, a much simpler, high-assurance hypervisor could be used for running a single VM securely. Regardless, Overshadow offers a valuable additional layer of defense-in-depth. As its protection model is orthogonal to that of the guest OS, protected applications require no additional privileges within the guest.

We make no attempt to protect network I/O, as this is addressed by existing technologies such as SSL. Although a trusted path for user input and secure display is also desirable [37], and could be facilitated by Overshadow, we have not tried to support this in the current system.

4.3 Multi-Shadowed Cloaking

In this section we review how traditional virtualized memory systems work, and explain how they can be extended to support multi-shadowing. Multi-shadowing is then coupled with encryption to implement cloaking, providing both encrypted and unencrypted views of memory.

4.3.1 Classical Memory Virtualization

Conventional operating systems use page tables to map virtual addresses to physical addresses with page granularity. A virtual page number (VPN) is mapped to a physical page number (PPN), and VPN-to-PPN translations are cached by a hardware TLB.

A classical virtual machine monitor (VMM) provides each virtual machine (VM) with the illusion of being a dedicated physical machine that is fully protected and isolated from other virtual machines [98]. To support this illusion, physical memory is virtualized by adding an extra level of address translation. The terms *machine address* and *machine page number* (MPN) are commonly used to refer to actual hardware memory [19, 124]. In contrast, "physical" memory is a software abstraction that presents the illusion of hardware memory to a VM. We refer to address translation performed by a guest operating system in a VM as mapping a *guest virtual page number* (GVPN) to a *guest physical page number* (GPPN).

The VMM maintains a *pmap* data structure for each VM to store GPPN-to-MPN translations. The VMM also typically manages separate *shadow page tables*, which contain GVPN-to-MPN mappings, and keeps them consistent with the GVPN-to-GPPN mappings managed by the guest OS [5]. Since the hardware TLB caches direct GVPN-to-MPN mappings, ordinary memory references execute without incurring virtualization overhead.

4.3.2 Multi-Shadowing

Existing virtualization systems present a single view of guest "physical" memory, faithfully emulating the properties of real hardware. One-to-one GPPN-to-MPN mappings are typically employed, backing each guest physical page with a distinct machine page. Some systems implement many-to-one mappings to support shared memory; *e.g.*, transparent page sharing maps multiple GPPNs copy-on-write to a single MPN [19, 124]. However, existing virtualization systems do not provide flexible support for mapping a single GPPN to multiple MPNs.[1]

Multi-shadowing is a novel mechanism that supports context-dependent, one-to-many GPPN-to-MPN mappings. Conceptually, multiple shadow page tables are used to provide different views of guest physical memory to different *shadow contexts*. The "context" that determines which view (shadow page table) to use for a particular memory access can be defined in terms of any state accessible to the VMM, such as the current protection ring, page table, instruction pointer, or some other criteria.

Traditional operating systems and processor architectures implement hierarchical protection domains, such as protection rings [101]. Multi-shadowing offers an additional dimension of protection orthogonal to existing hierarchies, enabling a wide range of unconventional protection policies.

4.3.3 Memory Cloaking

Cloaking combines multi-shadowing with encryption, presenting different views of memory – plaintext and encrypted – to different guest contexts. Our use of encryption is similar to XOM [76, 74], which modified both the processor architecture and operating system to encrypt and isolate application memory. The term "cloaking" has also been used by Intel's LaGrande Technology (LT) [59], which introduced a different architectural mechanism for creating orthogonal protection domains.

In contrast to XOM and LT, our virtualization-based cloaking does not require any changes to the processor architecture, OS, or applications. In fact, cloaking based on multi-shadowing represents a relatively small change to the core MMU functionality already implemented by a VMM. We initially describe cloaking using a high-level model. Details concerning metadata management and integration with existing systems are presented in later sections.

Single Page, Encrypted/Unencrypted Views. We represent each GPPN using only a single MPN, and dynamically encrypt and decrypt its contents depending on

[1]Some x86 VMMs do statically map a single GPPN to multiple MPNs to emulate the legacy `A20` line, for compatibility with real-mode applications. The `A20` line forces physical address bit 20 to zero, aliasing adjacent 1MB regions of memory.

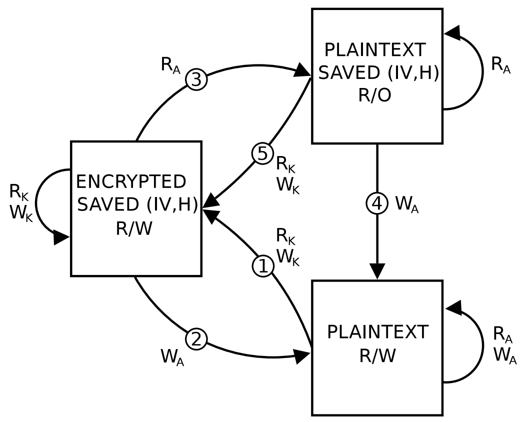

Figure 4.1: **Basic Cloaking Protocol.** State transition diagram for maintaining the secrecy and integrity of a single cloaked page. Application reads R_A and writes W_A manipulate plaintext page contents, while kernel reads R_K and writes W_K use an encrypted version of the page. A secure hash H is computed and stored immediately after page encryption, and verified immediately prior to page decryption.

the view currently accessing the page. This works well, since few pages are accessed simultaneously by both the application and the kernel in practice. As an optimization, the system could keep two read-only copies of the page, one encrypted, and one plaintext, for pages that are read concurrently from both views.

When a cloaked page is accessed from outside the shadow context to which it belongs, the VMM first encrypts the page, using a fresh, randomly-generated initialization vector (IV), then takes a secure hash H of this ciphertext. The pair (IV, H) is stored securely for future use. During decryption, the correct hash is first verified. If verification fails, the application is terminated. If it succeeds, the cloaked page is decrypted, and execution proceeds as normal. By checking the hash before decryption, any attempts to corrupt cloaked pages will be detected.

Overshadow currently uses a single secret key K_{VMM} managed by the VMM to encrypt all pages; see Section 4.7.7 for details. Encryption uses AES-128 in CBC mode, and hashing uses SHA-256; both are standard constructions. An integrity-only mode could be supported easily, but is not part of the current implementation.

Basic Cloaking Protocol. Consider a single guest "physical" page (GPPN). At any point in time, the page is mapped into only one shadow page table – either a protected *application shadow* used by a cloaked user-space process, or the *system shadow* used for all other accesses. When the page is mapped into the application shadow, its contents are ordinary plaintext, and application reads and writes proceed normally.

Figure 4.1 presents the basic state transition diagram for managing cloaked pages. When the cloaked page is accessed via the system shadow (transition 1), the VMM unmaps the page from the application shadow, encrypts the page, generates an integrity hash, and maps the page into the system shadow. The kernel may then read the encrypted contents, *e.g.*, to swap the page to disk, and may also overwrite its contents, *e.g.*, to swap in a previously-encrypted page from disk.

When the encrypted page is subsequently accessed via the application shadow (transitions 2 or 3), the VMM unmaps the page from the system shadow, verifies its integrity hash, decrypts the page, and maps the page into the application shadow. For an application read (transition 3), the page is mapped read-only and its (IV, H) is retained. If the page is later written by the application (transition 4), the (IV, H) is discarded, and the page protection is changed to read/write. If the page is instead accessed by the kernel (transition 5), the VMM proceeds as in transition 1, except that the hash for the (unmodified) page is not recomputed.

The read-only plaintext state, where the (IV, H) is retained, is required to correctly handle the case where the kernel legitimately caches a copy of the encrypted page contents. For example, this could occur if the kernel swaps a cloaked page to disk, which is later paged in due to an application read, and then swapped out again before the application modifies it. The kernel can optimize the second page-out by noticing that the page is not dirty, and simply unmap the page without reading it, since the on-disk swapped copy is still valid. If the (IV, H) had been discarded, it would not be possible to decrypt the page after it is swapped back in.

Cloaking is compatible with copy-on-write (COW) techniques for sharing identical pages within or between VMs. Plaintext pages can be shared transparently, and page encryption handled like a COW fault.

Virtual DMA. Cloaking is also compatible with virtual devices that access guest memory using DMA. For example, suppose the guest kernel performs disk I/O on a cloaked memory page via a virtual SCSI adapter. For a disk read, the cloaked page contents are already encrypted on disk, and the VMM simply permits the kernel to issue a DMA request to read the page.

For a disk write, the action taken by the VMM depends on the current state of the cloaked page. If the page is already encrypted, the VMM allows the DMA to be performed directly. When the page is in the plaintext read-only state, the VMM first encrypts the page contents with its existing (IV, H) into a separate page that is used for the DMA operation. Similarly, if the page is in the plaintext read-write state, the VMM encrypts its contents into a separate page used for the DMA operation. The cloaked page then transitions to the read-only plaintext state, and is associated with the newly-generated (IV, H). Note that in both plaintext states, the original guest page is still accessible in plaintext form to the application, since a transient encrypted copy is used during the actual DMA.

4.4 Overshadow Overview

Cloaking is a low-level primitive that protects the privacy and integrity of individual memory pages. Overshadow leverages this basic mechanism to cloak whole applications, cryptographically isolating application resources from the operating system.

Figure 4.2 provides an overview of the Overshadow architecture. A single VM is depicted, consisting of a guest OS together with multiple applications, one of which is cloaked. The VMM enforces a virtualization barrier between the cloaked application and the OS, similar to the barrier it enforces between the guest OS and host hardware. Overshadow introduces a *shim* into the address space of the cloaked application, which cooperates with the VMM to mediate all interactions with the OS.

Realizing the Overshadow design goal of whole-application protection for unmodified applications running on unmodified commodity operating systems has proved challenging. In this section, we describe several key challenges, sketch high-level solutions, and explain where more complete technical details can be found in subsequent

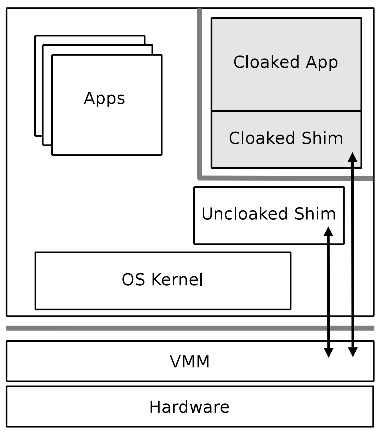

Figure 4.2: **Overshadow Architecture.** The VMM enforces two virtualization barriers (gray lines). One isolates the guest from the host, and the other cryptographically isolates cloaked applications from the guest OS. The shim cooperates with the VMM to interpose on all control flow between the cloaked application and OS.

sections.

Context Identification. The VMM must identify the guest context accessing a cloaked resource precisely and securely, in order to use the shadow page table with the correct GPPN-to-MPN view. Section 4.5 explains how Overshadow leverages the shim to help identify application contexts, without relying on an untrusted OS.

Secure Control Transfer. Applications must interact with the OS to perform useful work, and need to be adapted for cloaked execution. Overshadow performs this adaptation by injecting a *shim* into the address space of each cloaked application. The VMM cooperates with the shim to implement a transparent trampoline that interposes on all control transfers between the application and OS. The detailed mechanics of shim-based interposition for interrupts, faults, and system calls are discussed in Section 4.5.

System Call Adaptation. Most system calls require only simple argument marshalling between cloaked and uncloaked memory. Others, such as file I/O operations, need more complex emulation. For example, `read` and `write` system calls are implemented using `mmap` for encrypted I/O. Section 4.6 explains how particular system calls are adapted for cloaked execution.

Mapping Cloaked Resources. Overshadow must track the correspondence between application virtual addresses and cloaked resources. The shim is responsible for keeping a complete list of mappings, which is cached by the VMM. The shim resides in the same guest virtual address as the application, and interposes on all calls that modify it, such as `mmap` and `mremap`. A more detailed discussion is presented in Section 4.7.

Managing Protection Metadata. The VMM must maintain protection metadata, such as (IV, H) pairs, for each encrypted page, to ensure privacy and integrity. For active mappings, the VMM maintains an in-memory metadata cache that is not accessible to the guest. Metadata associated with persistent cloaked resources, such as file-backed memory regions, is stored securely within the guest filesystem. Section 4.7 contains a detailed treatment of Overshadow metadata management.

4.5 OS Integration with Cloaking

The VMM interposes on transitions between the cloaked user-mode application and the guest kernel, using distinct shadow page tables for each. Privilege-mode transitions include asynchronous interrupts, faults, and signals, and system calls issued by the cloaked application. Mediating these interactions in a secure, backwards-compatible manner requires adapting the protocols used to interact with the operating system, as well as some system calls. This is facilitated by a small *shim* that is loaded into a cloaked application's address space on startup.

We describe the shim in the context of our Linux implementation, although we believe this approach could be applied to other operating systems, including Microsoft Windows. While the system call interface varies across kernels, low-level mechanisms for system call vectoring, fault handling, and memory sharing are tied more closely to the processor architecture than to a particular OS.

We begin by discussing the basic operation of the shim, how it helps the VMM manage identity, and its interaction with the kernel and VMM to adapt the application for cloaked execution. Support for handling faults, interrupts, and system calls is presented in detail. A discussion of how particular system calls are mediated is deferred until the next section.

4.5.1 Shim Overview

The shim is responsible for managing transitions between the cloaked application and the operating system. It uses an explicit *hypercall* interface for interacting with the VMM, *i.e.,* a secure communication mechanism between the guest and the VMM. This arrangement allows relatively complex operations, such as OS-specific system call proxying, to be located in user-mode shim code, instead of the VMM. It also facilitates extensibility, providing a convenient place to add custom or OS-specific functionality without modifying the VMM.

Shim Memory. In memory, the shim consists of both cloaked and uncloaked regions, each with its own distinct code, data and stack space. Each application thread has its own shim instance, and all thread-specific data used by the shim is kept in thread-local storage, preventing conflicts between different instances.

The cloaked shim is multi-shadowed like the rest of the application. It is responsible for tasks where trust is required to maintain protection, such as providing well-defined entry and exit points for control transfers, and moving data between cloaked and uncloaked memory securely. The cloaked shim also includes a *cloaked thread context* (CTC) page, which is set aside for the VMM to store sensitive data used for control transfers. This includes areas for saving register contents, a table of entry points to shim functions, and the identity of the shadow context containing the shim.

The uncloaked shim contains buffer space that provides a neutral area for the kernel and application to exchange uncloaked data. It also contains simple trampoline code to facilitate transitions from the kernel to cloaked code. Nothing in the uncloaked shim is trusted or necessary for protection. If its code or data is corrupted, it will merely cause the application to crash.

Hypercall Interface. The VMM exports a small hypercall interface to the shim. Uncloaked code is allowed to invoke operations to initialize a new cloaked context (used to bootstrap). It can also make calls to enter and resume cloaked execution. Since control can be transferred only to an existing cloaked context, these calls can be initiated safely by untrusted code. Cloaked code can make hypercalls to cloak new memory regions, unseal existing cloaked data, create new shadow contexts, and access other useful interfaces, such as metadata cache operations.

Loading Cloaked Applications. To start a cloaked application, a minimal *loader* program is run with the shim linked into a distinct portion of its address space. The actual loader is part of the shim; before taking steps to load the program, the shim must bootstrap into a cloaked context.

To create a new shadow context, the shim issues a hypercall with a pointer to itself and protection metadata containing hashes for all pages associated with cloaked code and data; see Section 4.7 for details. The VMM uses this metadata to verify its integrity, as the cloaked shim will have access to the address space of the cloaked application. Thus, to bootstrap a secure protection domain for the application, the shim must be trusted; *i.e.*, not malicious to the application. The call to create a new context also takes a pointer to a portion of thread-local storage in which the VMM can setup a new CTC. Once this setup is complete, the VMM transfers control to start execution in the cloaked shim.

The cloaked shim then runs its loading routine, which reads the application binary, and maps appropriate sections into memory. When creating anonymous memory regions or memory-mapping protected files, the shim performs hypercalls to cloak their corresponding virtual memory ranges. After the cloaked application has been loaded, it may launch additional programs. On a subsequent `execve`, if the target program is cloaked, the loader program is prepended to the `exec` call so that the new program will also be cloaked.

Identity Management. To switch between shadow page tables appropriately, the VMM must employ some reliable procedure for identifying shadow contexts uniquely. Precise identification is challenging – contexts are associated with guest-level process abstractions, and scheduling is controlled by the OS, not the VMM. For example, the guest kernel may switch contexts while handling a fault or system call.

Existing approaches for VMM tracking of guest-level processes, such as monitoring assignments to the current page table root in Antfarm [63], work fairly well, but are not foolproof. Other schemes, such as accessing guest OS state at fixed kernel addresses (*e.g.*, Linux `current` pointer), or having the VMM store identifying information at some fixed virtual address, are generally fragile, or assume application pages can be pinned in physical memory. Most importantly, these approaches cannot be guaranteed to work in the presence of an adversarial OS. Overshadow takes an alternative shim-based approach that avoids these problems.

The VMM maintains a separate shadow context for each application address space, for which it assigns a unique *address space identifier* (ASID). Each address space may contain multiple threads, each with its own distinct cloaked thread context. When the shim begins execution, it makes a hypercall to initialize its CTC. During this initialization, the VMM writes the ASID and a random value into the CTC, and returns the ASID to the caller. The ASID value is not protected, and can be used by the uncloaked shim. However, since the CTC is cloaked, the random value is protected, and cannot be read by the uncloaked shim.

Shim hypercalls that transition from uncloaked to cloaked execution are self-identifying. The uncloaked shim passes arguments to the VMM containing its ASID, and the address of its CTC. The hypercall handler verifies that the CTC contains the expected random value, and also that its ASID matches the specified value. Note that the CTC resides in ordinary, unpinned application virtual memory. If the hypercall handler finds that the GVPN for the CTC is not currently mapped, it returns a failure code to the uncloaked shim, which simply touches the page to fault it back into physical memory, and then retries the hypercall.

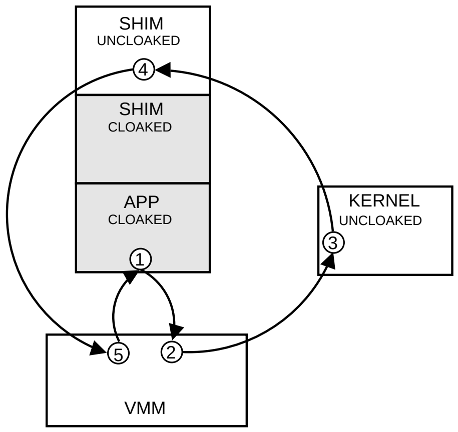

Figure 4.3: **Control Flow for Handling Faults and Interrupts**

4.5.2 Faults and Interrupts

While a cloaked application is executing, OS intervention is required to service faults or interrupts, such as application page faults and virtual timer interrupts. Figure 4.3 illustrates the flow of control for handling a fault from a cloaked application, involving the application, its associated shim, the guest kernel, and the VMM. The procedure for handling a virtual interrupt is essentially identical.

The fault occurs in step 1, and control is transferred to the VMM. In step 2, the VMM saves the contents of all application registers to the CTC in the cloaked shim. The VMM then zeros out the application's general-purpose registers to prevent their contents from being leaked to the OS. Next, the return instruction pointer (`IP`) and stack pointer (`SP`) registers are modified to point to addresses in the uncloaked shim, setting up a simple trampoline handler to which the kernel will return after servicing the fault. Finally, the VMM transfers control to the kernel.

The kernel handles the fault as usual in step 3, and then returns to the trampoline handler in the uncloaked shim setup in step 2. In step 4, this handler performs a self-identifying hypercall into the VMM to resume cloaked execution. In step 5, the VMM restores the registers saved in step 2, and returns control to the faulting instruction in the cloaked application.

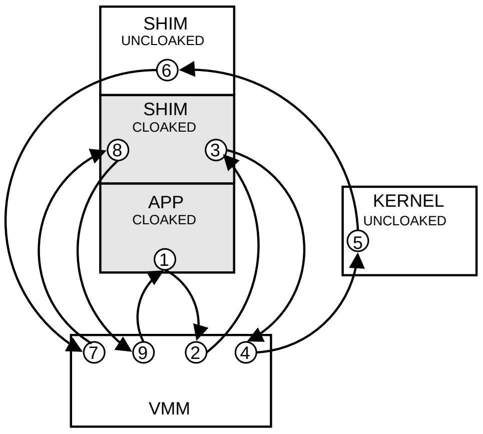

Figure 4.4: **Control Flow for Handling System Calls**

Note that the active shadow page table must be switched when transitioning between uncloaked and cloaked contexts. Two shadow page table switches are required to handle a fault, in steps 2 and 5.

4.5.3 System Call Redirection

Unlike faults and interrupts, which are intended to be transparent to the application, system calls represent an explicit interaction between the cloaked application and the kernel. A system call is issued by the application using the standard OS calling convention. Figure 4.4 depicts the flow of control for handling a system call from a cloaked application, involving the application, its associated shim, the guest kernel, and the VMM. Note that the transitions involved in performing a system call are a strict superset of the transitions presented for handling a fault in Figure 4.3.

In step 1, the cloaked application performs a system call, and control is transferred to the VMM. In step 2, the VMM saves the contents of all application registers to the CTC in the cloaked shim. The `IP` is set to an entry point in the cloaked shim corresponding to a system call dispatch handler; similarly, the `SP` is set to a private stack in the cloaked shim for executing this handler. The VMM then redirects control to the dispatch handler in the cloaked shim.

In step 3, the cloaked dispatch handler performs any operations required to proxy the system call on behalf of the application. For some system calls, this may involve marshalling arguments, copying them to a buffer in the uncloaked shim. The dispatch handler then reissues the system call, substituting the marshalled arguments in place of the original application-specified values. As before, the VMM again intercepts the system call.

In step 4, the VMM saves the contents of all application registers in the CTC. Note that the CTC contains two distinct register save areas: one for the application registers saved earlier in step 2, and one for the shim registers saved in this step. The VMM then scrubs the contents of any application registers that are not required by the kernel system call interface. The return `IP` and `SP` are modified to point to addresses in the uncloaked shim, setting up a simple trampoline handler to which the kernel will return after executing the system call. Finally, the VMM transfers control to the kernel.

The kernel executes the system call as usual in step 5, and then returns to the trampoline handler in the uncloaked shim setup in step 4. In step 6, this handler performs a self-identifying hypercall into the VMM to enter cloaked execution. In step 7, the VMM restores the shim registers saved in step 4, and resumes execution in the cloaked dispatch handler.

The cloaked dispatch handler continues execution in step 8, performing any operations required to finish proxying the system call. For some calls, this may involve unmarshalling result values, and copying them into cloaked application memory. The dispatch handler then performs a hypercall into the VMM, requesting resumption of the cloaked application. In step 9, the VMM restores the application registers saved in step 2, and returns control to the instruction after the original system call in the application.

As in the case of fault handling, only two transitions require shadow page table switches between uncloaked and cloaked contexts, during steps 4 and 7.

4.6 Adapting System Calls

Cloaking necessarily changes the way the OS can manage process memory – it cannot modify it or introduce any sort of sharing without application help. It also changes the way the OS transfers control – it can only branch to well-defined entry and exit points within the application. Accommodating these changes requires adapting the semantics of a variety of system calls.

4.6.1 Pass-through and Marshalling

A majority of system calls can be passed through to the OS with no special handling. These include calls with scalar arguments that have no interesting side effects, such as `getpid`, `nice`, and `sync`. The shim need not alter arguments to these system calls, so the cloaked shim is bypassed altogether, resulting in control flow like that in Figure 4.3. Note that the VMM itself is not aware of system call semantics; during initialization, the shim simply indicates which system call numbers can be bypassed.

Many other calls have non-scalar arguments that normally require the OS to read or modify data in the cloaked application's address space, for example, path names and `struct sockaddr`s. Such arguments are marshalled into a buffer in the uncloaked shim, and registers are modified so the system call uses this buffer as the new source (or destination) for non-scalar data. After the system call completes, results are copied back into the cloaked application, if necessary. Implementing all this manually would be tedious and error prone, so we instead generate this code automatically from a simple specification, and the resulting code is used by the shim.

4.6.2 More Complex Examples

Several system calls require changes to resolve incompatibilities between cloaked semantics and normal OS semantics. We first describe system calls that require non-trivial emulation, and then discuss thread creation and signal handling.

Emulation. We are forced to emulate the semantics of several system calls. For example, `pipe` normally creates a queue in the kernel for communicating bytes. We cannot easily protect this, so instead we emulate a pipe between cloaked applications with a queue in cloaked shared memory. To preserve the normal blocking semantics of calls such as `read`, `write`, and `poll`, reads and writes are performed over the pipe as normal, except that the sender sends zeros instead of actual data. For the receiver, zeros are read, then actual data is copied from the protected queue. Support for `futex` (Linux fast mutex) calls is another example of where emulation is required, as the normal OS implementation involves direct access to process memory.

Thread Creation. Handling the `clone` and `fork` system calls is particularly interesting, since these are intimately related with how the shim manages resources. A `clone` call begins by allocating thread-local storage for the new thread. Next, the child's cloaked thread context (CTC) is setup by making a copy of the parent's CTC, and fixing all thread-local pointers for the child. Finally, it changes the `IP` and `SP` for entering cloaked mode in the child's CTC, arranging for the child to start executing in a child_start

f

unction located in the child's shim, which will complete its initialization.

Normally, the CTC would be modified by the VMM on a switch from cloaked to uncloaked mode. However, in this case, the child's CTC is not currently being used. Thus, on a `clone` system call, only the parent's CTC is modified. We also setup the uncloaked stack that will be used by the cloned thread when returning from the system call, so that it will start running the new cloaked context. After returning from the system call, the parent thread returns to the original execution context. The child thread begins execution in child_start

,

as described above.

Signal Handling. Normal Unix signal-handling semantics are incompatible with cloaking, as we cannot allow the operating system to transfer control into an arbitrary section of cloaked code. Keeping portions of the shim non-preemptible also simplifies its implementation.

When the application registers a signal handler with `signal`, the shim emulates it, registering the handler in its own table. All actual signal handlers (those registered with the kernel) use a single handler located in the uncloaked shim. This signal handler makes a hypercall to the VMM immediately upon receiving a signal, indicating which shadow context received the signal, the signal that occurred, and any additional signal parameters.

The VMM examines the cloaked context and checks the signal status to determine in which context the signal occurred: the cloaked shim, uncloaked shim, cloaked application, or other uncloaked code. If the signal occurred when the cloaked application was executing, the VMM transfers control to a well-defined signal entry point in the shim, with relevant signal information. If the signal occurred while the shim was executing, the VMM further checks a flag in the CTC to determine whether to safely rollback execution to the last application system call entry point, or to defer the signal delivery until shim exit, when execution has effectively returned to the application.

4.6.3 File I/O

Extending Overshadow's cryptographic protection to files on disk requires interposing on I/O related system calls. Unprotected files are handled using simple argument marshalling, while protected files must be adapted to utilize cloaking.

Encrypted file I/O for cloaked applications is implemented in the shim using `mmap`. For example, `read` and `write` system calls are emulated by copying data to/from memory-mapped buffers. File data is always mapped using the `MAP_SHARED` flag, to ensure that other processes that may open the same file obtain a consistent view. By transforming all file I/O into memory-mapped I/O, file data is decrypted automatically when it is read by a cloaked application, and encrypted automatically when it is flushed to disk by the kernel. To allow the VMM to protect integrity and ordering of file data, the shim may need to load protection metadata from disk when the file is opened; this is described in detail in Section 4.7.6. For efficiency, the shim maintains a cache of mapped file regions; our current implementation maps regions using 1MB chunks to amortize the cost of the underlying `mmap` and `munmap` calls.

Using `mmap` for file I/O obviates the need to implement any cryptography in the shim. Also, encryption and decryption are performed only when necessary. An application can `read` and `write` portions of a file repeatedly without causing additional decryptions. Similarly, data is only encrypted when the OS flushes it to disk.

A single-page Overshadow header is prepended to each cloaked file. This header contains the actual file size, which may differ from the current on-disk size due to the 1MB mapping granularity. Each shim using the file maps its header using a shared `mmap`, to properly emulate operations such as `fstat` and `lseek`. The shim also tracks all operations that create or manipulate file descriptors, such as `dup`, and maintains a table of all open files, their offsets, and whether they are cloaked. This table is kept in a shared anonymous region to properly track and share descriptors across process forks.

4.7 Managing Protection Metadata

Overshadow introduces OS-neutral abstractions for cloaking both persistent and non-persistent resources, such as files and private memory regions. For each resource, *protection metadata*, such as (IV, H) pairs, must be managed to enforce privacy and integrity, ordering, and freshness (to prevent rollback). Figure 4.5 provides an overview of the components involved in metadata protection. We begin by examining how metadata is stored and mapped to protected objects, then consider how it is used to enforce protection.

4.7.1 Protected Resources

Each cloaked resource, such as a file or anonymous memory region, is associated with a unique 64-bit *resource identifier* (RID). Each RID has a corresponding resource metadata object (RMD) that stores metadata needed to decrypt, check integrity, and

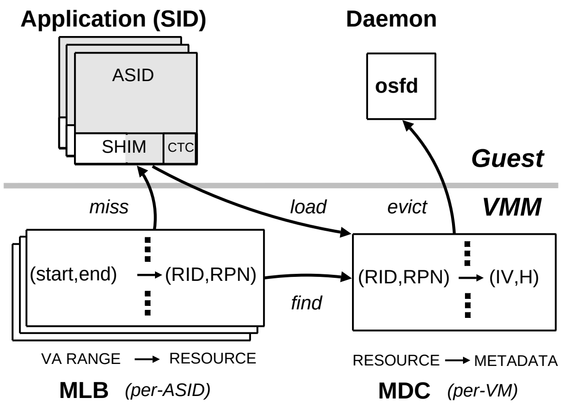

Figure 4.5: **Protection Metadata Management**

preserve ordering. Concretely, an RMD is an ordered set of (IV, H) pairs, one per encrypted page, addressed by a 32-bit *resource page number* (RPN). In our current system, each RMD is implemented with a data structure similar to a three-level page table to efficiently support large, potentially-sparse address spaces, up to 256GB.

When a resource is mapped into memory, its RMD is loaded into the *metadata cache* (MDC) in the VMM. A single MDC caches metadata for all cloaked resources mapped by the guest. This design ensures metadata consistency for shared objects, such as files and shared memory regions. When a resource is not in use by any process, its RMD is stored on disk in a metadata file. The MDC provides primitive operations to get, set, and invalidate metadata entries, as well as higher-level operations for cloning and persisting metadata, described later in this section.

4.7.2 Protected Address Spaces

Access control and sharing for cloaked resources are determined strictly by a unique *security identifier* (SID) that identifies an Overshadow protection domain. In the current implementation, a SID is associated with an application instance, which may contain multiple processes. Processes with the same SID have common access to cloaked resources. The address space for a cloaked process is identified by a unique *address space identifier* (ASID) that defines its shadow context. Portions of multiple cloaked resources are typically mapped into the guest virtual address space associated with a given ASID.

The VMM maintains a per-ASID cache of resource mappings in its virtual address space, called the *metadata lookaside buffer* (MLB). The MLB is used to map a virtual address to a resource. An MLB entry has the form (*start*,*end*) $\mapsto$ (RID, RPN), where *start* and *end* denote the virtual address range into which the resource is mapped, RID denotes the resource being mapped, and RPN denotes the first RPN in the mapping. For example, if file setspace.styfoo.txt has RID 4, and its third page is mapped into the first GVPN in the virtual address space, this is modeled as $(0, 4096) \mapsto (4, 2)$.

The shim is responsible for keeping a complete list of resource mappings for both cloaked and uncloaked memory, updating the MLB on any change. The shim resides in the same guest virtual address space, and interposes on all calls that modify it, such as `mmap`, `munmap`, and `mremap` in Linux; more details appear in Section 4.5. By delegating this responsibility to the user-mode shim, the VMM implementation is kept simple and OS-neutral.

On an MLB miss, the VMM performs an upcall into the shim to obtain the required mapping, and installs it in the MLB, illustrated by the *miss* action in Figure 4.5. The mappings for the shim itself are pinned in the MLB, preventing recursion. Note that even if some bug caused the MLB to have an incorrect mapping, it generally *fails-closed*; the wrong address range or cloaking status will cause decryption to fail, or the application will end up accessing ciphertext, causing it to fail.

4.7.3 Page Decryption

When a process accesses a cloaked page in its shadow context, its ASID and GVPN are known. If the page is unencrypted, then the memory access proceeds normally, without any VMM intervention.

If the page is encrypted, the access will fault into the VMM, since the GVPN is not mapped into the shadow for that ASID. The VMM looks up the faulting address in the MLB, and uses the resulting (RID, RPN) to index into the MDC and fetch the (IV, H) needed to decrypt and integrity check the page contents; see the *find* operation in Figure 4.5. The hash, check, and decrypt steps are performed using the protocol described previously. If the decryption succeeds, and the page is marked writable, (RID, RPN) is invalidated in the MDC. The page is then *zapped*, *i.e.*, removed from all shadows, and mapped into the current shadow for the ASID. The original application access is then allowed to proceed.

There is one special case. Operating systems commonly zero the contents of a page before mapping it into userspace, and applications depend on this initialization. If an access is made to a GVPN that is not mapped in the current shadow, and the (RID, RPN) for that page is not in the MDC, then this must be the first application access to the page, and no decryption is necessary. We check that the page contents are indeed zero-filled, and assuming this succeeds, the page is simply zapped and then mapped into the current shadow, and the original memory access is allowed to proceed.

Finally, the VMM stores the (RID, RPN) used for each decryption with the associated GPPN in the existing VMM pmap structure which stores GPPN-to-MPN translations.

4.7.4 Page Encryption

When the guest kernel (or any context that doesn't match the application SID) accesses a cloaked page, its GPPN is known, but its ASID and GVPN may not be known. The access could originate from any guest context, *e.g.*, during a virtual DMA operation. If the page is already encrypted, then the memory access proceeds normally, without any VMM intervention.

If the page is unencrypted, the access will fault into the VMM, since it is not mapped in the current shadow. If the page is writable, the VMM generates a new random IV; for a read-only page, the existing IV is reused. The VMM then encrypts the page contents, and computes a secure hash H over the encrypted contents. It stores the resulting (IV, H) in the MDC, at the (RID, RPN) previously associated with the GPPN in the pmap during its last decryption. The page is then zapped and mapped into the current shadow, and the original kernel access is allowed to proceed.

4.7.5 Cloning Metadata

The MDC also provides operations to facilitate support for address space cloning, such as `clone` or `fork` in Linux. Suppose a cloaked process forks a child. Immediately after the fork, the parent and child processes share their private memory regions copy-on-write (COW). Overshadow must ensure that the metadata associated with all unmodified COW pages remains accessible and synchronized between the parent and child.

When the fork occurs, each of the parent's private RMDs is cloned eagerly for the child, by copying all of its existing metadata entries, and assigning it a new RID. This ensures that metadata for any pages encrypted prior to the fork remain available to the child, even if the parent later modifies them.

However, suppose the parent encrypts a COW-shared page after the fork; a subsequent access by the child would not find the metadata required for decryption. One approach is to forcibly encrypt all pages in the parent during the fork, but this would be extremely inefficient, since few private pages remain encrypted in practice, unless the system is swapping heavily. Another option is to store a complete backmap for every GPPN, containing all (ASID, GVPN) pairs that map it, but this would be extremely complex.

The solution we implemented is to mirror the application's process tree in the MDC; each RMD has pointers to its parent, first child, and next sibling RMDs, if any. The MDC also maintains a global 64-bit version number, which is incremented on every RMD creation and page decryption. A version is stored with each RMD, set to the global version when it is created. Similarly, a version is stored along with the (RID, RPN) in the pmap for each GPPN, and set to the global version each time it is decrypted. Whenever a page is encrypted, the (IV, H) is stored at the (RID, RPN) associated with the GPPN, and also recursively propagated to any child RMDs with versions greater than the GPPN's version. Thus, metadata is propagated to all children with pages whose contents existed prior to the fork, as desired. A subtle point worth noting is that when the parent modifies a COW page, it will be encrypted (and its metadata propagated to the child) prior to the modification, since the guest OS must first read the page to make a private copy for the parent during the COW fault.

4.7.6 Persisting Metadata

RMDs associated with non-persistent memory regions (*e.g.*, application stack, data, or anonymous shared memory), can be discarded when no longer in use. However, RMDs associated with persistent content, such as file-backed memory regions, must be saved to disk. Each cloaked file has an associated metadata file in the guest for storing its RMD persistently. Metadata file integrity is protected by a message authentication code (MAC) stored in the file, computed using a key derived from the VMM's secret key K_{VMM}. The current implementation uses HMAC with SHA-256.

When a process opens a cloaked file, the shim makes a hypercall to determine if the metadata for its RID is in the MDC. If the metadata is not present, the shim performs a hypercall to allocate a new RMD for that RID, reads the entire metadata file, and passes its contents to the VMM, which verifies its integrity, as illustrated by the *load* action in Figure 4.5. Frequently reloading the RMD or recomputing its MAC might raise efficiency concerns. This can be optimized by keeping RMDs cached longer in the MDC, instead of evicting them eagerly after they have been committed to disk. Another option would be to store MACs in a Merkle hash tree [85], allowing for more efficient verification and updates.

To ensure freshness, a 128-bit generation number is also written to the metadata file, and protected by the MAC. The VMM checks this number against a master list of valid generations when the file is loaded. This number is stored in the MDC as part of the RMD. Just prior to eviction, it is incremented in both the RMD and master list. The master list is stored in the guest, protected by a MAC and its own counter which is stored outside of the guest by the VMM.

RMDs are written to metadata files by the Overshadow file daemon (`osfd`). The `osfd` communicates with the VMM via a simple hypercall interface, polling for metadata that should be evicted from the MDC and persisted to disk. The daemon extracts the metadata for all of its valid RPNs, obtains their MAC as generated by the VMM, commits everything to disk, and finally evicts the RID from the MDC; refer to the *evict* action in Figure 4.5. Notably, the `osfd` daemon is not trusted, and all data it handles is protected cryptographically. Its compromise would sacrifice only system availability, not data privacy or integrity.

4.7.7 Key Management and Access Control

Our usage model during Overshadow development has been to set up a clean VM and cloak an unmodified application in place. However, one could easily use our tools from outside the VM to convert existing Linux packages (*e.g.*, `rpm` files) by encrypting their files, and adding corresponding metadata files to the package.

Given the simple primitives in our architecture, a wide range of access control policies could be supported, as SIDs provide a basic primitive for identifying subjects, and RIDs provide a basic primitive for identifying objects. We currently use a simple model that assumes mutual trust between all parts of an application and dynamically assigns SIDs at startup.

Our current implementation performs all encryption using a single set of encryption and MAC keys. It is important to note that key management and access control in Overshadow are orthogonal. The VMM arbitrates who is allowed to access what resources, regardless of the key with which it was encrypted. Additional keys could be added to support delegation of administrative tasks; *e.g.*, a key per RID would allow different parties to package their own sets of encrypted files outside of the VM.

4.8 Evaluation

The current Overshadow implementation realizes the full system described in earlier sections. It supports cloaking for all application memory regions – private and shared, anonymous and file-backed. We demonstrate that the system is practical by presenting quantitative results for experiments running substantial, unmodified applications on an unmodified Linux operating system.

4.8.1 Implementation

The Overshadow implementation is based on a version of the VMware VMM for 32-bit x86 processors that uses binary translation for guest kernel code [5]. The modified VMM was built as a VMware Workstation binary running in a “hosted” configuration on top of an existing Linux host OS.[2] Since multi-shadowed memory cloaking does not depend on specific features of the VMware VMM, it could also be realized in other virtualization platforms.

Our VMM modifications included approximately 4600 new lines of code, plus 2000 additional lines from publicly-available cryptographic routines. The shim handles nearly all system calls supported by the Linux 2.6 kernel interface, and is sufficiently complete to run large, unmodified Linux programs. The shim consists of 13,100 lines of code, including roughly 8500 lines of new code, and 4600 lines of standard library and utility routines.

Changes would be required to enable hardware-assist for x86 virtualization, such as Intel VT [89] and AMD SVM [6]. For example, system call transitions between guest user-mode and kernel-mode are always trapped by a binary-translating VMM, but are not typically trapped by a hardware-assisted VMM. Forcing system calls to trap for Overshadow interposition would likely introduce additional overhead. Nevertheless, we expect that hardware support for nested page tables will accelerate many Overshadow operations, improving overall performance. Reducing the cost of hardware context switches is also desirable. For Overshadow, the ability to redirect a trap to guest user-mode code would be ideal, making it possible to redirect system calls to handlers in the shim without dynamic VMM intervention.

4.8.2 Performance

All experiments were conducted on a Dell Precision 390 host configured with a 2.66GHz Intel Core2 Duo processor and 4GB RAM. The VM was configured with one CPU and 2GB memory running an unmodified Fedora Core 7 guest OS (Linux 2.6.21-1 kernel).

Microbenchmarks. Figure 4.6 presents the results of microbenchmarks that measure the overhead of system call redirection and cloaking. Each data point plots the

[2]In this configuration, the VMM is not fully protected from the Linux host OS. Secure deployment of Overshadow would require running the VMM directly on hardware, like VMware ESX Server [124], Xen [15], or IBM z/VM.

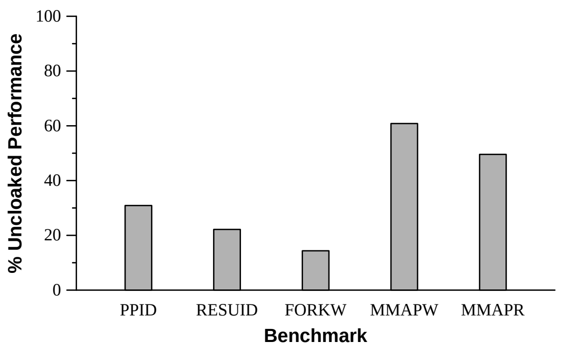

Figure 4.6: **Microbenchmarks.** Percentage of uncloaked performance attained with full cloaking of all memory regions and files.

ratio of cloaked to uncloaked performance, using the mean over the best 5 of 7 trials for both. In these experiments, all the benchmarks exhibited low runtime variability (standard deviation within 2.4% of the mean).

The PPID and RESUID benchmarks measure raw system call overheads, for both `getppid`, implemented using pass-through, and `getresuid`, requiring simple argument marshalling. Cloaking clearly increases system call costs significantly (by a factor of 3 to 5), primarily due to the extra pair of address-space switches on transitions between cloaked and uncloaked guest contexts.

The FORKW benchmark highlights the overhead of process creation, destruction, and synchronization using `fork` and `wait`. Cloaking introduces overhead due to encryption and decryption for copy-on-write (COW) pages, as well as execution in the shim handler and VMM during process creation.

The MMAPW benchmark measures the cost of writing one word to each page in a large file-backed memory region, and flushing the data to disk. The cloaking overhead is dominated by the cost of encryption during disk write operations. MMAPR measures the cost of reading one word from each page of a large memory region backed by the file written by MMAPW. This benchmark incurs page faults, but does not perform disk I/O, as pages containing the file data still reside in the guest buffer cache. In this experiment, cloaking does not cause decryptions because these pages remain accessible to the application in plaintext form during virtual DMA (see Section 4.3.3).

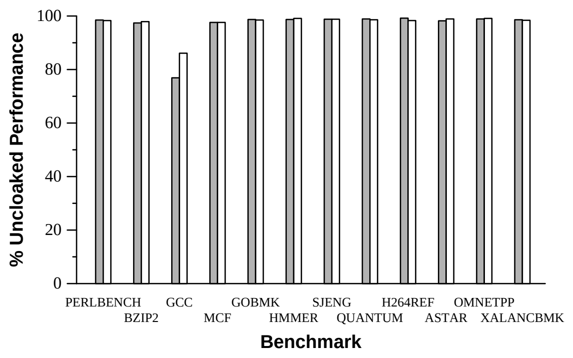

Figure 4.7: **SPEC CINT2006 Benchmarks.** Percentage of uncloaked performance attained with full cloaking of all memory regions and files (gray), and cloaking anonymous memory regions only (white).

The results indicate that cloaking approximately doubles the cost of a minor page fault.

Application Benchmarks. Figures 4.7 and 4.8 present results from the SPEC CPU2006 integer suite and aggressively-loaded Linux web and database servers. All data points are averages over at least three trials. Despite high overheads on some microbenchmarks (Figure 4.6), real applications perform additional work that amortizes these costs.

Figure 4.8 plots the geometric mean for the entire SPEC suite, showing that overall the SPEC benchmarks incur very little overhead from cloaking. When we consider the SPEC benchmarks individually (Figure 4.7), only GCC has non-trivial overhead. This overhead comes from GCC's relatively high system call and page fault rates.

The web server experiment used the standard prefork configuration of APACHE 2.2.4, with caching disabled. A remote host generated client requests for fetching a 28 KB HTML file using the `ab` benchmarking tool with 50 concurrent connections. The client and server were connected by a 100Mbps (APACHE-100M) or 1Gbps (APACHE-1G) switch. We measured the total number of requests served per second. With full cloaking, performance for APACHE-100M was within 1% of uncloaked performance. When using the 1 Gbps switch (APACHE-1G), fully-cloaked performance

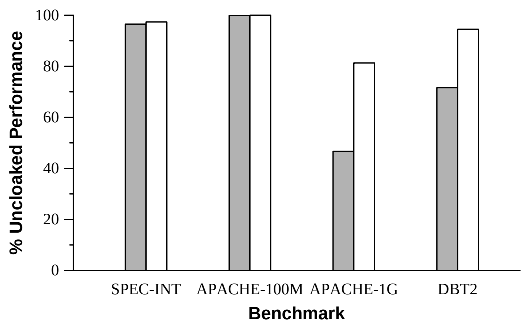

Figure 4.8: **Benchmark Summary.** Percentage of uncloaked performance attained with full cloaking of all memory regions and files (gray), and cloaking anonymous memory regions only (white).

degraded relative to the uncloaked server. These results are explained by the fact that cloaked applications have higher CPU occupancy; for the more realistic load (few web servers saturate 100 Mbps Internet links), the processor was not fully utilized, and cloaking didn't affect performance. With the network bottleneck removed, cloaking scales well until it saturates the CPU. Naturally, this saturation point will occur later for multicore VM configurations (our experiments utilize only one processor).

The database workload uses the DBT2 transactional database performance test suite running with a PostgreSQL 8.2.4 server. This test simulates a wholesale parts supplier with 22 warehouses and 11 concurrent clients at the peak throughput; the clients run uncloaked in the same VM as the server. We measured the number of "new order" transactions per minute during steady-state operation, the standard metric from this suite. With full cloaking enabled for this 8.6 GB database, performance was more than 70% of the uncloaked baseline.

While there are ample opportunities for optimization, the current implementation of full cloaking is practical for many realistic workloads. For applications that require only anonymous regions to be cloaked, performance is uniformly above 80% of the baseline.

4.9 Future Work

A variety of interesting research opportunities remain in the areas of retrofitting protection to legacy operating systems, and leveraging multi-shadowing to define new protection models. We also describe extensions to our current implementation.

Retrofitting Protection. Applications are not designed with the expectation that the operating system can become hostile. A more formal treatment of how an OS might mislead an application – and how such attacks can be mitigated – is an interesting topic for future research. For example, an application might be misled into revealing information if it is run with a particular `uid`. One possible defense is to provide a "reverse sandbox" that filters system calls to prevent such attacks.

We are also investigating a trusted path for user interface devices, as this would enable complete protection of many compelling applications, including web, email, and VOIP clients. In principle, user interaction could be protected in the current implementation if the application uses a remote display system that renders to software frame buffers.

Protecting Device Memory. Many I/O devices present a memory-mapped interface to software. For some devices, multi-shadowing can be employed to protect the contents of "physical" device memory from being inspected or modified by untrusted software. For example, an interactive VM typically provides a virtual high-resolution graphics display that uses a memory-mapped frame buffer. A multi-shadowed frame buffer could help implement a trusted path, by ensuring that a cloaked application's output remains private. While this approach can be used to prevent the operating system from observing raw device memory, additional work is needed to cloak off-screen display images and other memory used by window managers and graphics subsystems.

Fine-Grained Cloaking. Applications can be modified to apply multi-shadowing selectively, cloaking only sensitive pages. For example, two shadow contexts could be defined for each application: a protected shadow containing cloaked code and data, and an unprotected shadow for uncloaked code and data. In this simple model, cloaked memory can be accessed only by cloaked code. A shadow context is identified by the virtual address of the current instruction pointer.

In order to interpose on transitions between these shadow contexts, a VMM can change the execute permission of pages in the shadow page tables (independent of guest PTE permissions). In the unprotected shadow, all protected pages are marked non-executable; similarly, in the protected shadow, all unprotected pages are marked non-executable. When the application branches between protected and unprotected code, the resulting permissions-based page fault will trap into the VMM, allowing it to switch between shadow page tables.

Storage Extensions. Our current implementation has a few storage-related limitations. First, the RID for a file is simply constructed from its device and inode numbers; this is problematic on network file systems where uniqueness can't be expected.

Next, Overshadow currently offers no protection for file system metadata; consequently, the OS could maliciously swap inputs on an application. A simple solution is to provide a secure namespace, associating pathnames with (RID, MAC) pairs. This could be implemented by employing a protected daemon or shared file, which would be updated on file operations such as `rename`, `create`, and `unlink`. More sophisticated approaches have been explored by others [45, 39, 72].

Finally, we currently don't maintain consistency between file system data and metadata in the event of a system crash. If the guest OS crashes before we commit the metadata for a given file, or before the data for a given file is committed to disk, we could end up in a state where data and metadata are out of sync. We believe all of these issues are tractable, but a full treatment remains a topic for future work.

4.10 Related Work

Memory virtualization enables transparent remapping of guest physical pages. Previous systems have leveraged this ability for transparent swapping [19, 124], transparent page sharing [19, 124], transparent page migration across NUMA nodes [19], and transparent VM migration across physical hosts [24, 90]. Overshadow takes advantage of this extra level of indirection to provide isolated, context-dependent views of guest physical memory.

Many prior systems have attempted to tackle the problem of providing a higher-assurance execution environment on commodity platforms. All have aimed to eliminate the need to trust commodity operating systems with the security of sensitive data.

Intel's LaGrande Technology (LT) [59] provides hardware mechanisms for isolating a portion of a machine's address space to create orthogonal protection domains within the guest. The NGSCB [37] (formerly Palladium) architecture proposed using this functionality to split commodity systems into low-assurance and high-assurance partitions. The low-assurance partition would run a commodity operating system (Windows); the other, a simpler trusted operating system (the Nexus). Applications would correspondingly be split into a small trusted part (the agent, run under the Nexus) and the untrusted part, run on the commodity OS.

Proxos [114] takes a similar, but more backwards-compatible approach. It splits the system into multiple VMs, one running an untrusted commodity OS, the other(s) running trusted, application-specific operating systems. Sensitive applications are run in a trusted VM, but still interact with resources in the untrusted VM via a process that proxies system calls for it, manipulating resources on its behalf. The Terra [41] architecture proposes moving the entire application into a separate VM with its own application-specific software stack tailored to its assurance needs. While Overshadow takes an OS-level approach to application protection, unlike all of these earlier systems, it does not introduce any additional resource management mechanisms or new operating systems.

Retrofitting protection via an encryption layer is a familiar concept from networking. In storage, systems like SUNDR [72] and Sirius [45] have examined securing block and file-level storage on untrusted substrates, and similar work exists for databases [82]. However, this approach is rarely encountered in the OS literature, with the exception of architecture-level research such as XOM [74, 76] and SP [32, 70]. XOM and SP also provide a dual encrypted-unencrypted view of memory, like Overshadow, though they achieve this through custom processor architectures, and target a threat model where hardware attacks are possible, *i.e.*, main memory is untrusted.

Overshadow considers only software attacks, but works with off-the-shelf hardware. Compared to architecture-level approaches, Overshadow also gains substantial flexibility by being software-based. XOM requires applications and/or the OS to be substantially modified or rewritten [74]. SP also requires applications to be rewritten, explicitly specifying which code and data to protect. While SP does not need OS modifications, it supports only one protection domain per device [70]. In contrast, Overshadow makes integration with unmodified operating systems and applications feasible, and enables sharing between protection domains. Nevertheless, Overshadow's software mechanisms could be combined with more hardware-centric approaches to provide similar benefits.

We have developed Overshadow as a means of enhancing security in commodity systems, where redesigning applications and using a high-assurance OS [54, 104] is not an option. However, we believe cloaking is useful as a more general OS abstraction, with novel properties not afforded by normal memory protection. In particular, cloaking provides an OS-level analog to end-to-end encryption in networks, eliminating the need to trust those pieces of the system that are merely responsible for moving data from one place to another, versus those that are actively using that data.

Chapter 5

Conclusion

In this thesis we have explored three major paradigms for enhancing operating system security using the new capabilities afforded by virtualization technology.

First, we presented a flexible architecture for trusted computing, called Terra. Terra allows applications to run in an “open box” VM with the semantics of a modern open platform, or in a “closed box” VM with those of dedicated, tamper-resistant hardware. The key primitive that Terra builds on is a trusted virtual machine monitor (TVMM). The TVMM mechanisms allow Terra to partition the platform into multiple, isolated VMs. Each VM can tailor its software stack to its security and compatibility requirements. We examined the primitives the TVMM provides for building closed-box VMs, in particular those required to support “attestation,” the mechanism used to cryptographically identify the contents of closed-box VMs to remote parties. We described how to efficiently implement these primitives. We implemented these primitives in a prototype implementation of Terra and built a selection of applications using this prototype that demonstrate its capabilities. We believe that the closed-box VM abstraction provided in the Terra architecture forms the basis for a truly general-purpose trusted computing platform.

Next, we explored virtual machine introspection, an approach to intrusion detection which co-locates an IDS on the same machine as the host it is monitoring and leverages a virtual machine monitor to isolate the IDS from the monitored host. The activity of the host is analyzed by directly observing hardware state and inferring software state based on a priori knowledge of its structure. This approach allows the IDS to: maintain high visibility, provides high evasion resistance in the face of host compromise, provides high attack resistance due to strong isolation, and provides the unique capability to mediate access to host hardware, allowing hardware access control policies to be enforced in the face of total host compromise. We showed that implementing our architecture is practical and feasible using current technology by implementing a prototype VMI IDS and demonstrating its ability to detect real attacks with acceptable performance. We believe VMI IDS occupies a new and important point in the space of intrusion detection architectures.

Finally, we presented *Overshadow*, a system that cryptographically isolates an application inside a virtual machine from the operating system it is running on, offering another layer of protection for application data, even in the face of total OS compromise. This capability is enabled by *multi-shadowing*, a novel technique for presenting different views of "physical" memory in virtualized systems. This allows memory to be *cloaked*, so that it appears normal to an application, but encrypted to the operating system. Cloaking supports a separation of responsibilities for isolation and resource management, allowing the use of a complex commodity operating systems to manage application virtual memory and other resources, while relying on a much simpler hypervisor to ensure data privacy and integrity. Unlike previous approaches to enhancing assurance in commodity systems, Overshadow is backwards-compatible, protecting a broad range of unmodified legacy applications, managed by unmodified commodity operating systems. While Overshadow is not a panacea, we believe it demonstrates a promising approach to enhancing data security in commodity computing environments.

While the face of the computing landscape will continue to change, and thus the role of virtualization with it, we believe that the paradigms presented here represent fundamental architectural patterns that will continue to have relevance in virtualized platforms for decades to come.

Bibliography

[1] IBM mainframe servers: Case studies. http://www-1.ibm.com/servers/eserver/zseries/library/casestudies/.

[2] IP security protocol (IPsec) charter. http://www.ietf.org/html.charters/ipsec-charter.html.

[3] Security: IBM zSeries partitioning achieves highest certification. http://www-1.ibm.com/servers/eserver/zseries/security/certification.html, December 2002.

[4] Microsoft next-generation secure computing base—technical FAQ. http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/news/NGSCB.asp, February 2003.

[5] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, October 2006.

[6] AMD. *AMD64 Virtualization Technology: Secure Virtual Machine Architecture Reference Manual*, May 2005.

[7] R. Anderson. Cryptography and competition policy: Issues with trusted computing. In *Proc. Workshop on Economics and Info. Sec.*, pages 1–11, May 2003.

[8] R. Anderson and M. Kuhn. Tamper resistance—A cautionary note. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 1–11, November 1996.

[9] Anonymous. Runtime process infection. *Phrack*, 11(59):article 8 of 18, December 2002.

[10] Apk. Interface promiscuity obscurity. *Phrack*, 8(53):article 10 of 15, July 1998.

[11] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Proc. 1997 IEEE Symp. Sec.*, pages 65–71, May 1997.

[12] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symp. Sec. and Privacy*, Oakland, May 2002. IEEE, IEEE Computer Society Press.

[13] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

[14] G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *Proc. CRYPTO'2000*, 2000.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[16] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message authentication using hash functions—the HMAC construction. *CryptoBytes*, 2(1), Spring 1996.

[17] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. 15th ACM Symp. on Operating Sys. Principles*, December 1995.

[18] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[19] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 143–156, October 1997.

[20] Amy Carroll, Mario Juarez, Julia Polk, and Tony Leininger. Microsoft Palladium: A business overview. `http://www.microsoft.com/PressPass/features/2002/jul02/0724palladiumwp.asp`, August 2002.

[21] D. Chaum and E. Van Heyst. Group signatures. *Advances in Cryptology, Eurocrypt '91*, 547:257–265, 1991. Springer-Verlag Lecture Notes on Computer Science.

[22] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proc. 2001 Workshop on Hot Topics in Operating Sys. (HotOS-VIII)*, Schloss Elmau, Germany, May 2001.

[23] A. Chou, J. Yang, B. Chelf, S. Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP '01)*, October 2001.

[24] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation*, pages 273–286, May 2005.

[25] Jack R. Collins. Knark: Linux kernel subversion. Sans Institute IDS FAQ.

[26] Jack R. Collins. RAMEN, a Linux worm. Sans Institute Article, February 2001.

[27] R. J. Creasy. The origin of the VM/370 time-sharing system. *IBM J. Research and Development*, 25(5):483–490, September 1981.

[28] P.T. Cummings, D.A. Fullan, M.J. Goldstien, M.J. Gosse, J. Picciotto, J.P.L. Woodward, and J. Wynn. Compartmented model workstation: Results through prototyping. In *Proc. IEEE Symp. Sec. and Privacy*, pages 27 – 29, April 1987.

[29] DarkNova. Interview with an aimbot coder. `http://www.lamerkatz.com/webvoid/issue7/1.shtml`.

[30] James Downey. Sniffer detection tools and countermeasures.

[31] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. of 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[32] Jeffrey S. Dwoskin and Ruby B. Lee. Hardware-rooted Trust for Secure Key Management and Transient Trust. In *Proceedings of the Fourteenth ACM Conference on Computer and Communications Security*, pages 389–400, October 2007.

[33] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L. van Doorn, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Comp.*, 34:57–66, October 2001.

[34] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L. van Doorn, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34:57–66, October 2001.

[35] Paul England. Personal communication.

[36] D.R. Engler, M.F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource managment. In *Proc. 15th ACM Symp. on Operating Sys. Principles*, December 1995.

[37] P. Englund, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Spectrum*, pages 55–62, July 2003.

[38] Flocutus. The ultimative Quake cheating page: Illegitimate cheats. `http://www.gamescenter.de/uqc/illegal.htm`.

[39] K. Fu, M. Frans Kaashoek, and David Mazières. Fast and Secure Distributed Read-only File System. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 181–196, October 2000.

[40] FX. cdoor.c, packet coded backdoor. `http://www.phenoelit.de/stuff/cd00rdescr.html`.

[41] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 193–206, October 2003.

[42] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Net. and Distributed Sys. Sec. Symp.*, February 2003.

[43] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. A Broader Vision for Trusted Computing. In *9th Workshop on Hot Topics in Operating Sys. (HotOS-IX)*, May 2003.

[44] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proc. 12th NIST-NCSC Nat'l Comp. Sec. Conf.*, pages 305–319, 1989.

[45] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Network and Distributed System Security Symposium*, pages 131–145, February 2003.

[46] B. Gold, R. Linde, R. J. Peller, M. Schaefer, J. Scheid, and P. D. Ward. A security retrofit for VM/370. In *AFIPS Natl. Comp. Conf.*, volume 48, pages 335–344, June 1979.

[47] B. Gold, R. R. Linde, and P. F. Cudney. KVM/370 in retrospect. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 13–23, April 1984.

[48] B. Gold, R. R. Linde, M. Schaefer, and J. F. Scheid. VM/370 security retrofit program. In *Proc. ACM Annual Conference*, pages 411–418, October 1977.

[49] B.D. Gold, R.R. Linde, and P.F. Cudney. KVM/370 in retrospect. In *Proc. IEEE Symp. Security and Privacy*, April 1984.

[50] R.P. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

[51] R.P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7:34–45, June 1974.

[52] Guido van Rossum. Python Reference Manual. `http://www.python.org/doc/current/ref/ref.html`.

[53] halflife. Bypassing integrity checking systems. *Phrack*, 7(51):article 9 of 17, September 1997.

[54] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza Secure-System Architecture. In *Proceedings of the International Conference on Collaborative Computing*, December 2005.

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

[56] Roger Howworth. Virtual servers pay off. IT Week, March 2003.

[57] Inc. id Software. Quake. `http://www.idsoftware.com/games/quake/`.

[58] A. Iliev and S.W. Smith. Prototyping an armored data vault: Rights management on Big Brother's computer. *Privacy-Enhancing Technology*, 2002. Springer-Verlag Lecture Notes on Computer Science.

[59] Intel. *Intel Trusted Execution Technology Preliminary Architecture Specification*, November 2006.

[60] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[61] J. J. Donovan and S. E. Madnick. Hierarchical Approach to Computer System Integrity. *IBM Sys. J.*, 14(2):188–202, 1975.

[62] Gaurav Jain. Certificate revocation: A survey. `http://www.cis.upenn.edu/~jaing/papers/`.

[63] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, June 2006.

[64] P.A. Karger, M.E. Zurko, D.W. Bonin, A.H. Mason, and C.E. Kahn. A retrospective on the VAX VMM security kernel. In *IEEE Trans. Soft. Eng.*, November 1991.

[65] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[66] klog. Backdooring binary objects. *Phrack*, 10(56):article 9 of 16, May 2000.

[67] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[68] Paul Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of Crypto '96*, pages 104–113, 1996.

[69] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Comp. Sys.*, 10(4):265–310, 1992.

[70] Ruby B. Lee, Peter C. S. Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 2–13, June 2005.

[71] Ben Leslie and Gernot Heiser. Towards untrusted device drivers. Technical Report 0303, University of New South Whales, March 2003.

[72] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, pages 121–136, December 2004.

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

[74] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 178–192, October 2003.

[75] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 168–177, 2000.

[76] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000.

[77] Jochen Liedtke. On μ-kernel construction. In *Proc. 15th Symp. on Operating Sys. Principles*, pages 237–250, December 1995.

[78] Lord Somer. lrk5.src.tar.gz, Linux rootkit V. `http://packetstorm.decepticons.org`.

[79] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proc. USENIX Tech. Conf., FREENIX Track*, pages 29–42, June 2001.

[80] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proc. Nat'l Info. Sys. Sec. Conf.*, pages 303–314, October 1998.

[81] M. Accetta et al. Mach: A new kernel foundation for UNIX development. In *Proc. USENIX Summer Conf.*, 1986.

[82] Umesh Maheshwari, Radek Vingralek, and William Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.

[83] Kevin Mandia and Keith J. Jones. Carbonite v1.0 - A Linux Kernel Module to aid in RootKit detection. `http://www.foundstone.com/knowledge/proddesc/carbonite.html`.

[84] Marcus J. Ranum. Intrusion Detection and Network Forensics. USENIX Security Course Notes, 2000.

[85] Ralph Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, April 1980.

[86] R. Meushaw and D. Simard. NetTop: Commercial technology in high assurance applications. `http://www.vmware.com/pdf/TechTrendNotes.pdf`, 2000.

[87] Mission Critical Linux. Core Analysis Suite v3.3. `http://oss.missioncriticallinux.com/projects/crash/`.

[88] Nelson Murilo and Klaus Steding-Jessen. chkrootkit: locally checks for signs of a rootkit. `http://http://www.chkrootkit.org/`.

[89] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), August 2006.

[90] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, pages 391–394, April 2005.

[91] paul england and marcus Peinado. Authenticated operation of open computing devices. In *Proc. 7th Australian Conf. Info. Sec. and Privacy*, pages 346–361, 2002. Springer-Verlag Lecture Notes on Computer Science.

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

[93] A. Perrig, S.W. Smith, D. Song, and J.D. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. *eJETA.org: The Electronic Journal for E-Commerce Tools and Applications*, 1(1), January 2002.

[94] pragmatic. (nearly) Complete Linux Loadable Kernel Modules. `http://www.thehackerschoice.com/papers/LKM_HACKING.html`.

[95] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[96] Rainer Wichmann. Samhain: distributed host monitoring system. `http://samhain.sourceforge.net`.

[97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the simos machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[98] R.P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, June 1974.

[99] S. R. Ames, Jr. Security kernels: A solution or a problem? In *Proc. IEEE Symp. Sec. and Privacy*, April 1981.

[100] M. Schaefer and B. Gold. Program confinement in KVM/370. In *Proc. 1977 Ann. ACM Conf.*, pages 404–410, October 1977.

[101] Michael D. Schroeder and Jerome H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[102] sd. Linux on-the-fly kernel patching without lkm. *Phrack*, 11(58):article 7 of 15, December 2001.

[103] Kurt Seifried. Honeypotting with VMware: Basics. `http://www.seifried.org/security/ids/20020107-honeypot-vmware-basics.html`.

[104] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, December 1999.

[105] Weidong Shi, Joshua B. Fryman, Guofei Gu, Hsien-Hsin Lee, Youtao Zhang, and Jun Yang. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, pages 222–231, February 2006.

[106] Silvio Cesare. Runtime Kernel Kmem Patching. `http://www.big.net.au/~silvio/runtime-kernel-kmem-patching.txt`.

[107] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *Proceedings of the First ACM EuroSys Conference*, pages 161–174, 2006.

[108] Sean W. Smith. Outbound authentication for programmable secure coprocessors. In D. Gollman et al., editor, *ESORICS 2002: 7th European Symp. Research in Comp. Sec.*, volume 2502/2002, pages 72–89, Zurich, Switzerland, October 2002. Springer-Verlag Heidelberg.

[109] Sean W. Smith and David Safford. Practical server privacy with secure coprocessors. *IBM Sys. J.*, 40(3):683–695, 2001.

[110] D. Song. Passwords found on a wireless network. USENIX Technical Conference WIP, June 2000.

[111] Stealth. The adore rootkit version 0.42. `http://teso.scene.at/releases.php`.

[112] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. 2001 Ann. USENIX Tech. Conf.*, Boston, MA, USA, June 2001.

[113] Mike M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proc. 19th Symp. on Operating Sys. Principles*, October 2003.

[114] Richard Ta-Min, Lionel Litty, and David Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, pages 279–292, November 2006.

[115] Peter S. Tasker. Trusted computer systems. In *Proc. IEEE Symp. Sec. and Privacy*, April 1981.

[116] Teso Security Advisory. LIDS Linux Intrusion Detection System vulnerability. `http://www.team-teso.net/advisories/teso-advisory-012.txt`.

[117] Tim Lawless. St Michael: detection of kernel level rootkits. `http://sourceforge.net/projects/stjude`.

[118] Toby Miller. Analysis of the T0rn rootkit. `http://www.sans.org/y2k/t0rn.htm`.

[119] Trusted Computing Platform Alliance. TCPA main specification v. 1.1b. `http://www.trustedcomputing.org/`.

[120] J. D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proc.*, 1994.

[121] VMware, Inc. VMware virtual machine technology. `http://www.vmware.com/`.

[122] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proc. IEEE Symposium on Security and Privacy*, 2001.

[123] Carl A. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. 2002 Symp. Operating Sys. Design and Implementation*, December 2002.

[124] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 181–194, December 2002.

[125] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable length audit trail patterns. In *RAID 2000*, pages 110–129, 2000.

[126] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th USENIX Symp. on Operating Sys. Design and Implementation*, December 2002.

[127] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Trans. Comp. Sys.*, 12(1):3–32, 1994.

[128] Xie Huangang. Building a secure system with LIDS. `http://www.de.lids.org/document/build_lids-0.2.html`.

[129] Bennet Yee and Doug Tygar. Secure coprocessors in electronic commerce applications. In *Proc. 1st USENIX Workshop on Elec. Commerce*, New York, New York, July 1995.