

Protecting Xen hypercalls
Intrusion Detection/ Prevention in a Virtualization
Environment

by

Cuong Hoang H. Le

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

July, 2009

© Cuong Hoang H. Le 2009

Abstract

During the past few years virtualization has strongly reemerged from the shadow of the mainframe generation as a promising technology for the new generation of computers. Both the research and industry communities have recently looked at virtualization as a solution for security and reliability. With the increased usage and dependence on this technology, security issues of virtualization are becoming more and more relevant. This thesis looks at the challenge of securing Xen, a popular open source virtualization technology. We analyze security properties of the Xen architecture, propose and implement different security schemes including authenticated hypercalls, hypercall access table and hypercall stack trace verification to secure Xen hypercalls (which are analogous to system calls in the OS world). The security analysis shows that hypercall attacks could be a real threat to the Xen virtualization architecture (i.e., hypercalls could be exploited to inject malicious code into the virtual machine monitor (VMM) by a compromised guest OS), and effective hypercall protection measures can prevent this threat. The initial performance analysis shows that our security measures are efficient in terms of execution time and space.

Table of Contents

Abstract	ii
Table of Contents	iii
List of Tables	vi
List of Figures	vii
Acknowledgments	viii
1 Introduction	1
1.1 Virtual Machine Monitors and Xen Overview	1
1.2 Virtualization and Security	2
1.3 Xen VMM and Security	3
1.4 Hypercalls Security Model	4
1.4.1 Hypercalls	4
1.4.2 Hypercall Security Model	4
1.5 Approach to Prevent Hypercall Attacks	5
1.6 Challenges	6
1.7 Constraints	6
1.8 Contributions	6
1.9 Summary of The Thesis	6
2 Background	8
2.1 Virtualization Overview	8
2.1.1 The Development of Hardware Virtualization	8
2.1.2 Virtualization Architectures	10
2.2 Xen	12
2.2.1 Overview	12
2.2.2 Design	12
2.2.3 Control and Driver Domains	15
2.2.4 Xen Hypercall Internals	15

Table of Contents

2.2.5	Hypercall Invocation	16
2.3	A Security Model for Xen	16
2.3.1	Xen Security Model Overview	16
2.3.2	Security Boundary and Points of Attack	19
2.3.3	Assets to Protect	20
2.3.4	Specific Threats:	21
2.3.5	Current Security Mechanisms in Xen	23
2.3.6	Discussion about Protecting Xen	25
3	Related Works	26
3.1	System Calls	26
3.2	Syscall Attack Overview	26
3.3	Classes of Syscall Attacks	27
3.3.1	Attack Targets:	27
3.3.2	Exploitation Strategies Based on Modification of Control Flow:	28
3.4	Syscall Defense Challenges	30
3.4.1	Classification of Syscall Defenses	31
3.4.2	Proactive Mechanisms	32
3.4.3	Reactive Mechanisms	34
3.5	Syscalls vs. Hypercalls	36
4	Prototype Design and Implementation	38
4.0.1	Hypercall Protection Considerations	38
4.1	High Level Overview	39
4.2	Authenticated Hypercalls and Hypercall Access Table	39
4.2.1	Authenticated Hypercalls	39
4.2.2	Transformation of a Guest Kernel	41
4.2.3	Hypercall Address Table	42
4.2.4	Constructing Hypercall Address Table	43
4.3	Hypercall Verification	43
4.3.1	Verifying Authenticated Hypercalls	43
4.3.2	Checking Hypercall Address Table	44
4.4	Hypercall Chain Verification	44
4.5	Discussion	45
4.5.1	Remaining Issues	46
5	Evaluation	47
5.1	Security Evaluation	47
5.1.1	Experimental Setup	47

Table of Contents

5.1.2	Attack Experimental Issues	47
5.2	Performance	48
5.2.1	Micro Benchmark	48
5.2.2	Macro Benchmark	49
5.3	Correctness	51
6	Conclusion and Future work	54
	Bibliography	57

List of Tables

5.1	Hypercall Micro-benchmark	49
5.2	Macro-benchmark.	53

List of Figures

2.1	Non-virtualized Architecture	8
2.2	Virtualized Architecture	9
2.3	Types of Virtualization	11
2.4	Xen Architecture	13
2.5	Hypercall Diagram	17
2.6	Xen Security Boundary	18
5.1	Micro Benchmarks	50
5.2	Apache Performance	51

Acknowledgments

I would like to thank Norm Hutchinson, Bill Aiello and Andrew Warfield for valuable guidance and discussion. This research would not have been possible without them.

I would also like to thank my parents for their love, support and encouragement.

Chapter 1

Introduction

In the last few years, virtualization technology has been becoming increasingly popular. Virtualization platforms such as Xen and VmWare are widely used by both research and industry communities for a variety of applications. With the increased usage and dependence on this technology, its security issues become more and more relevant.

Virtualization merits such as isolation and security are usually taken for granted. There is a tendency to assume that given a much narrower low-level interface compared to that of conventional operating systems, virtualization platforms are inherently more secure. Although such an assumption seems plausible, recent studies [17, 47] show that the virtualization interface can be vulnerable to randomized attacks and is potentially exposed to other attacks.

Enhancing security for a virtualization is a practical need to make this technology more robust. For instance, a more secured Xen is beneficial for a variety of services built on top of Xen and would improve both its reliability and usability. There are several aspects regarding to security of Xen. Among these the security of the Xen hypercall interface is critically important because of its role as a sole means to communicate to the VMM. Securing Xen hypercalls would prevent certain kinds of attacks against the virtualization environment; thus enhance its security overall.

1.1 Virtual Machine Monitors and Xen Overview

Virtual machine monitors (also known as **hypervisors**) are often used by IBM, HP, and others on mainframes and large servers. They are also used by Internet hosting service companies to provide virtual dedicated servers. The primary benefits of virtualization are consolidation, increased utilization, an ability of rapid provision of a virtual machine, and increased ability to dynamically respond to faults by re-booting a virtual machine or moving a virtual machine to different hardware [51]. Moreover, virtualization also can separate virtual operating systems, and is able to support legacy soft-

ware as well as new OS instances on the same computer.

Although modern computers are sufficiently powerful to use virtualization, there are several challenges to successfully virtualize a machine hardware to support the concurrent execution of multiple operating systems. First of all, virtual machines must be safely isolated from one another. It is unacceptable for the execution of one to adversely affect the performance or security of another, especially when virtual machines are owned by mutually untrusted users. In addition, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Finally, the performance overhead introduced by virtualization must be small [19].

Xen is an open-source software virtual machine monitor supporting IA-32, IA-64 and PowerPC architectures. It runs on bare hardware and provides hardware virtualization so that several guest operating systems can run on top of Xen on the same computer hardware at the same time. Modified versions of Linux and NetBSD can be used as hosts. Several modified Unix-like operating systems may be employed as guests. On certain hardware, unmodified versions of Microsoft Windows and other proprietary operating systems can also be used as guests [19]. Xen's major strengths are the ability to provide close-to-native performance and its support for efficient virtual machine live migration.

1.2 Virtualization and Security

The VMM or hypervisor can fully mediate all interactions between guest operating systems and underlying hardware, thus allowing strong isolation between virtual machines and supporting the multiplexing of many guests on a single hardware platform. This property is valuable for security. According to Rosenblum [51], VMMs offer the potential to restructure existing software systems to provide greater security, while also facilitating new approaches to building secure systems.

Current operating systems provide poor isolation, leaving host-based security mechanisms subject to attack. Moving these capabilities outside a guest so that they run alongside the guest OS but are isolated from it offers the same functionality but with much stronger resistance to attack. One popular example of such system is ReVirt [20], which uses the VMM layer

to analyze the damage hackers might have caused during the break-in. It not only gains greater attack resistance from operating outside the virtual machine, but also benefit from the ability to interpose and monitor the system inside the virtual machine at a hardware level.

Placing security outside a virtual machine also provides an attractive way to quarantine the network limiting a virtual machine's access to a network to ensure that it is neither malicious nor vulnerable to attack. In addition, virtual machines are also particularly well suited as a building block for constructing high-assurance systems. The US National Security Administration's NetTop architecture, for example, uses VMware's VMM to isolate multiple environments, each of which has access to separate networks with varying security classifications [51].

One interesting feature of VMMs is that they support the ability to run multiple software stacks with different security levels. The ability to specify the software stack from the hardware up and specify an application's complete software stack simplifies reasoning about its security. In contrast, it is almost impossible to reason about the security of a single application in today's operating systems because processes are poorly isolated from one another. Hence, an application's security depends on the security of every other application on the machine. The VMM also can authenticate software running inside a virtual machine to remote parties, in a process called *attestation*. Systems such as Terra [24] utilize these capabilities demonstrate that VMMs are particularly well suited for building trusted computing.

These capabilities are promising for improving system security. However, providing secure isolation requires that the VMM be free of bugs in both design and implementation that attackers could use to subvert the system. A VMM subversion is likely damaging to all virtual machines running on the system and creating even more complicated security issues.

1.3 Xen VMM and Security

The Xen VMM or hypervisor is designed to achieve security. According to its design goal, Xen should be a relatively simple program compared to the conventional OS kernel, with a very narrow, stable and well-defined interface to the software running above it. For instance, Xen has only 35 hypercalls compared to 324 syscalls of the current Linux 2.6.22 kernel. Unlike tradi-

tional operating systems, which must support file systems, 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, Xen (and other VMMs in general) have been widely considered as an architecture for building secure operating systems [51].

Unlike the cases of VMMs for mainframe computing and the IBM zSeries whose isolation properties have received intense scrutiny or been certified as conforming to the common criteria requirements for assurance [3], the security property of Xen has not been verified yet despite the fact that the Xen VMM is a critical security component in Xen vitalization.

1.4 Hypercalls Security Model

1.4.1 Hypercalls

Guest OSes communicate with Xen and make resource requests via the XEN API or *hypercalls*. The hypercall interface serves as the sole mechanism for cross-layer interaction between VMs and Xen. The role of hypercalls in the Xen VMM is similar to the role of system calls in an OS. Basically, a hypercall is a software trap from a domain or virtual machine to Xen, just as a syscall is a software trap from an application to the kernel. A hypercall transfers control to a more privileged state in the VMM. Domains use hypercalls to request high privileged operations like low-level memory requests. Like a syscall, the hypercall is synchronous, but the return path from Xen to the domain uses event channels [19].

1.4.2 Hypercall Security Model

Given the similarity with the syscall interface, the Xen hypercall interface plays a very important role to Xen security. In the OS world, the low level OS API (aka. system calls or syscalls) are well-known attack targets; in fact, most security incidents happen in operating systems that involved syscalls either directly or indirectly. It is plausible to conjecture that hypercalls could be easy targets for similar attacks. Attackers could exploit hypercalls to open new attack vectors and create new challenges for the security community. Some recent studies suggest that the hypercall interface can be exploited to inject malicious code into the VMM. Works of Cully [17] and Omandy [47] shows that randomized hypercall attacks can corrupt the vir-

tualization platform in certain fixed intervals which prompts security issues that need to be resolved.

A typical scenario of hypercall attack could happen in the following steps: 1) first, the attacker compromises one of the applications on a guest OS. This is possible even if the guest started in a clean state because guest applications communicate with the outside world, they could be infected with malware; 2) upon the success of the first step, the attacker would escalate his privilege by common syscall attack methods; 3) when the attacker can get inside the guest kernel (equivalent to the escalation to the ring 1 privilege in the x86 protection ring architecture), he can launch attacks to the hypervisor via hypercalls.

Hypercall attacks could be in any form known for syscall attacks such as argument hijacking or mimicry. Since hypercalls are highly privileged, this attack can be very effective. Some security objectives of the attacker could be denying services to certain guests, introducing low-level malware or rootkits that are beyond scanning capability of the current tools to sniff guest's private information or corrupting the entire virtualization system.

1.5 Approach to Prevent Hypercall Attacks

We first study in depth various aspects of the hypercall interface such as implementation, operation and security. Due to the similarity to system calls, we next analyze the syscall attacks and defenses. Finally, based on our analysis, we would select appropriate defense methods and adapt their implementations on Xen.

Our objective is to prevent intrusion via hypercalls from compromised guests in order to protect the VMM and to maintain normal services for other guests. To achieve that objective, we propose a set of comprehensive approaches to authenticate hypercalls. Authentication of hypercalls is carried out by a cryptographic method in which each hypercall has a Message Authentication Code (MAC) added as a parameter to ensure its authenticity and correctness. Alternatively, a caching technique, hypercall access table (HAT), is deployed to achieve some level of security with better performance. Finally, we study the utilization of the existing information from hypercall invocations such as call stack traces as an alternative method to verify the authenticity of hypercalls.

1.6 Challenges

- **Cost:** while enhancing security, we need to make sure that performance overhead added by security measures reasonable
- **Simplicity and Stability:** changes must be minimal and do not affect other components especially guest applications. Modifications should not impede future changes on other parts of the virtualization architecture
- **Correctness:** since hypercalls are highly privileged, correctness should be guaranteed, i.e., no (or minimal) false negatives or false positives

1.7 Constraints

- Protecting against a well-behaved guest OS which may be compromised and used as a platform to launch attacks to the VMM and other guest OSES
- Protecting by detecting (monitoring) to catch potentially malicious hypercalls and proactively blocking them to protect the virtualization assets

1.8 Contributions

This thesis has two main contributions

- The first is an analysis of security threats via hypercalls and the evaluation of security measures which can counter those threats
- The second is the proof-of-concept working implementation prototypes of the two approaches MAC and HAT

1.9 Summary of The Thesis

This thesis studies different approaches to enhance the security of Xen hypercalls. These approaches include hypercall authentication, hypercall access table, and hypercall stack chain analysis. These security measures help to prevent certain attacks on Xen by injecting malicious hypercalls. Benefits of these protection mechanisms are mainly simplicity, performance and transparency to guests. Our initial evaluation shows that these can effectively

1.9. Summary of The Thesis

protect against some imminent hypercall threats to Xen. Our research approaches include performing a security analysis of Xen and studying the existing techniques in the OS world to apply into the virtualization world with Xen as an implementation platform.

Chapter 2

Background

2.1 Virtualization Overview

Virtualization of the hardware allows the physical hardware of a single computer to be shared between multiple OSes in a transparent manner. Virtualization is performed by the special host software namely the virtual machine monitor (VMM) or hypervisor which provides virtualization capabilities and virtual resources such as processor, memory, IOs, and storage for each virtualized OS. This technology offers a number of benefits such as cost saving, resource isolation and security. Virtualization products such as VmWare and Xen are widely used by a wide range of users. New hardware techniques such as Intel Vanderpool and AMD Pacifica designed to support virtualization have improved the virtualizability of commodity hardware, thus making it increasingly popular for both the research and industry communities.

2.1.1 The Development of Hardware Virtualization

From a high level viewpoint, an operating system provides the first level of virtualization which regulates and facilitates the interaction between the *hardware architecture* and the *software architecture*. Figure 2.1 shows the prevailing non-virtualized hardware/software architecture.

A certain hardware architecture provides resources such as processor, memory, hard disks and devices. An OS manages accesses to the hardware

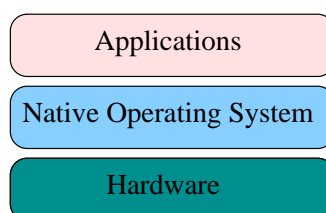


Figure 2.1: Non-virtualized Architecture

2.1. Virtualization Overview

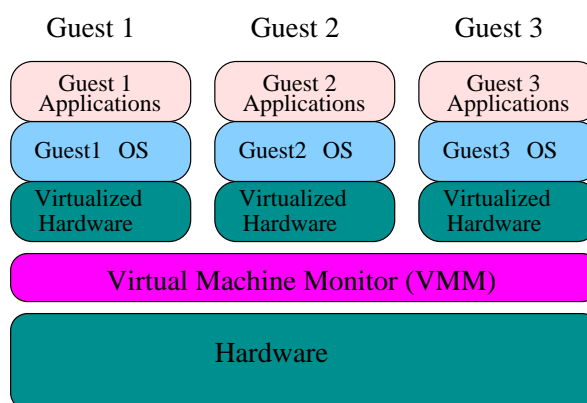


Figure 2.2: Virtualized Architecture

resources, and abstracts the hardware for applications. Except poor application isolation, this abstraction brings many benefits. Firstly, security and reliability are increased because the OS can prevent insecure or damaging access to hardware. Hardware issues such as flaws or version differences can be masked and hidden from applications. Secondly, the OS allows applications to share hardware resources in a concurrent and coordinated fashion. Finally, applications can access hardware resources through the common OS application programming interface.

In the traditional non-virtualized system architecture, only one OS may execute on the hardware platform at a time. This means only applications compatible with the particular OS version can be executed on a piece of hardware. Virtualization of hardware resources overcomes this limitation and allows multiple OSes running on the same hardware by adding another software layer to abstract the hardware. Figure 2.2 illustrates the components of a virtualized architecture, and introduces the central concepts of virtualization. In a virtualized architecture, access to physical hardware and provision of virtual hardware is managed by a *virtual machine monitor* (VMM). As such a *virtual machine* (*VM*) (or *guest*) is provided a virtualized view of hardware to run a *guest OS*.

Figure 2.3 illustrates the two generic models of hardware virtualization. In these models, the guest OS and applications interacts with virtualized hardware. The virtual machine monitor (VMM) performs a similar role to an OS by providing services for guest OSes. There are several kinds of VMM;

two basic ones are type 1 and type 2 VMMs. A type 1 VMM is implemented directly on top of hardware, in which case the term *hypervisor* is often used. A type 2 VMM is implemented on top of another OS (or the *host OS*), as is the case with VMWare. The term VMM covers both approaches.

2.1.2 Virtualization Architectures

All virtualization architectures essentially provide the same services: (1) concurrent sharing of hardware resources between guests, (2) provision of virtual resources based on real resources, and (3) abstraction of hardware details from guests.

Full Virtualization Full virtualization is the traditional virtualization architecture. Guests are provided a view of virtual hardware modeled after actual hardware used by the host. In particular, they are unaware of any virtualization and ideally, recursive virtualization, where a guest provides its own virtual machine monitor, should be possible.

The primary benefit of full virtualization is that existing OSes can run as guests without modification. Although performance appears to be an issue, it can be enhanced by eliminating excessive virtualization operations. In addition, the benefit of not having to modify guest OSes often overrides performance considerations in certain cases. Also, despite the apparent performance disadvantage of full virtualization compared to paravirtualization, full virtualization products such as VmWare have acceptable performance.

Paravirtualization In paravirtualization the guest OS is aware of virtualization and interacts with the VMM through the VMM API. Virtualized resources, such as virtual network and storage devices, can be accessed directly without the need for virtual device drivers. During the process of virtualizing the guest OS, features such as memory management, scheduling, and other core OS features can be designed to work with the support from the VMM.

The primary benefit of paravirtualization is low virtualization overhead. Without the additional overhead of virtualizing certain hardware devices, close-to-native performance is possible even on architectures lacking a full virtualizable instruction set such as the basic x86. The primary drawback of paravirtualization is the need to modify the guest OS kernel. Modifications are required for each supported VM and possibly different versions of

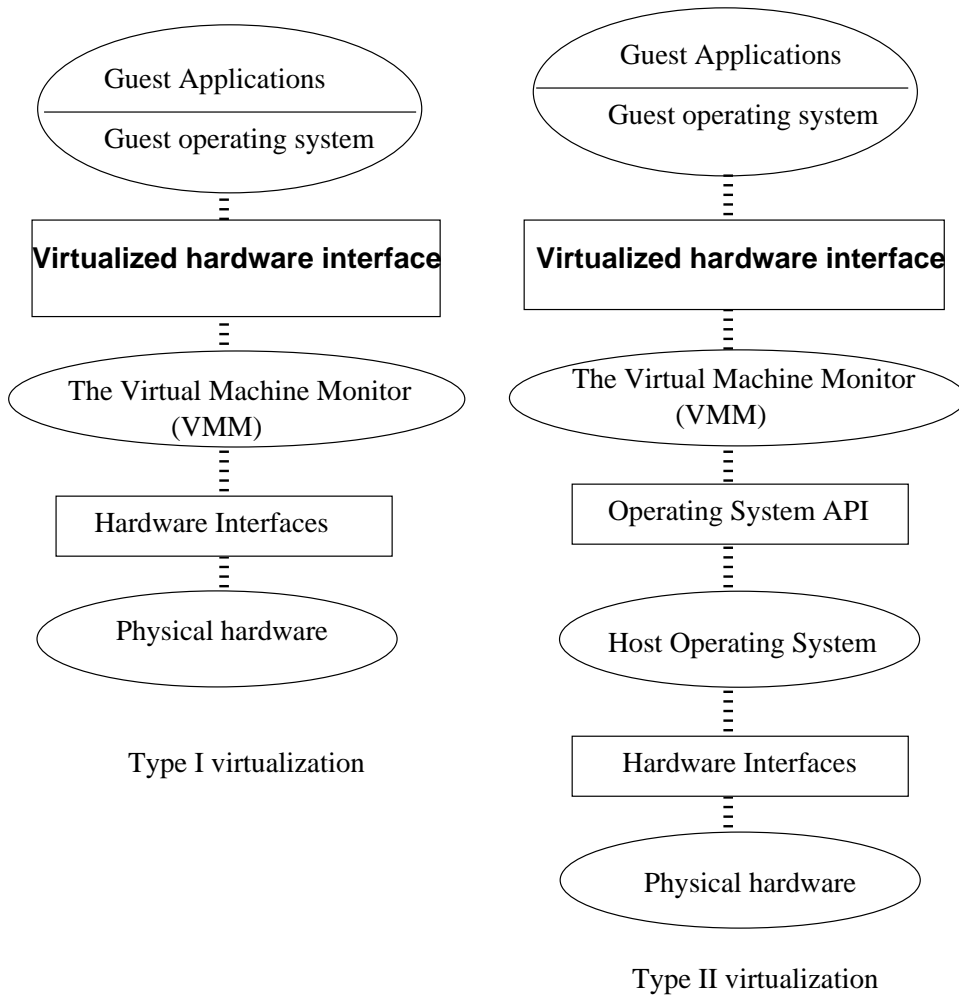


Figure 2.3: Types of Virtualization

each guest OS. To overcome the excessive need of modifying guests, systems like Xen adopt a hybrid approach in which sections critical to performance are paravirtualized manually, while other sections are emulated using trap-driven full virtualization [19].

2.2 Xen

2.2.1 Overview

Xen is a type 1 VMM for paravirtualization and it also supports full virtualization on hardware supported by VTx or Pacifica. Xen paravirtualization exposes idealized virtual hardware for guest OSes i.e., from the view of a guest OS, Xen is simply a normal hardware architecture. The main goals of the Xen architecture are: (1) secure isolation of resources; (2) support for a wide variety of commodity OSes; (3) very low performance overhead of virtualization; and (4) support a large number (>100) of virtual machines on modern hardware [19].

2.2.2 Design

The Xen design was motivated by the desire for high performance virtualization of commodity OSes. Xen strikes a balance between high performance, accurate hardware virtualization, and compatibility with existing software. Although guest OSes need to be modified to run on Xen, guest applications compiled for a specific Application Binary Interface(ABI) continue to work unmodified [19].

As illustrated in Figure 2.4, the Xen VMM runs with full x86 privileges (ring 0), guarding access to all privileged operations and hardware resources. Guest OSes are modified to run with less privilege (ring 1) without direct access to ring 0. Instead, guest OSes request access to privileged operations through Xen. Xen performs checking of all requests to maintain the desired isolation property. The VMM also handles all interfacing with host hardware and device drivers. Xen strives for separation of policy and mechanism: a control domain (namely *domain 0*) manages virtualization policies, while the Xen VMM (or Xen) provides the low level mechanisms implementing these policies.

CPU Virtualization: uses direct execution, which is possible because all guest code executes at x86 ring 1 or lower without access to critical

2.2. Xen

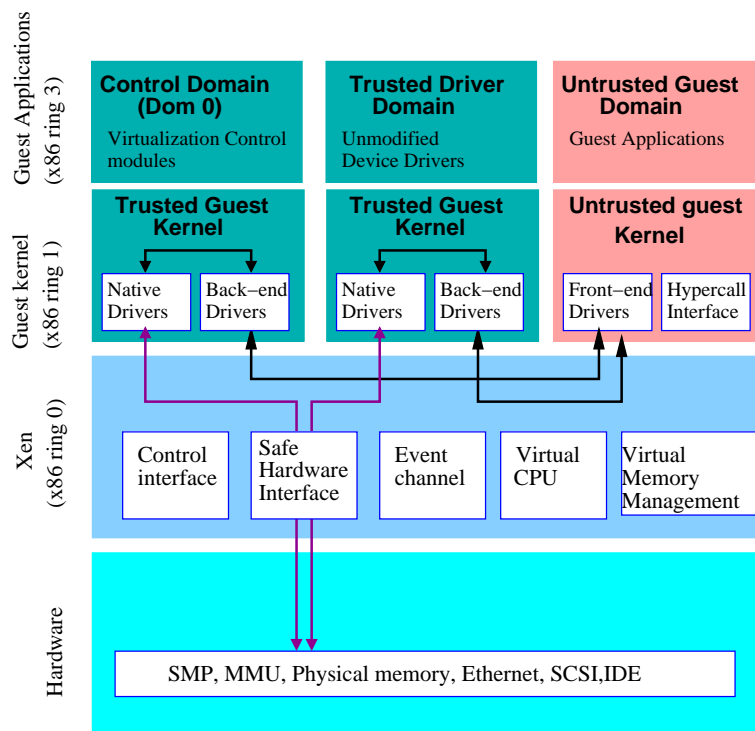


Figure 2.4: Xen Architecture

instructions. All attempts to access such critical instructions are trapped and control transferred to Xen. Guests are modified to request privileged services from Xen through *hypercalls*, and Xen performs appropriate security checks before providing services. To minimize context switch overhead of hypercalls, multiple hypercall operations may be batched and executed using one hypercall [19].

Memory Virtualization: Since guest kernels run in ring 1, all low level memory operations must go through Xen. This allows Xen to partition and manage the memory for guest domains. Xen manages the operations for paging and segmentation. As a result, the guest kernel cannot directly edit the page tables, global descriptor tables (GDT) or local descriptor tables (LDT) because doing so could potentially crash other domains on the same system. These operations must all be requested through hypercalls and mediated by Xen.

Three different types of addresses in Xen virtualization are machine addresses, pseudo-physical addresses and virtual addresses.

- Machine addresses are memory addresses on physical memory.
- Pseudo-Physical addresses are addresses in which the first 20 bits specify a page in the page table, and the last 12 indicate an offset.
- Virtual addresses are offsets within a segment. They must be added to the segment base, then looked up in the page table before the machine address can be found.

Xen provides a domain with a list of machine frames during bootstrapping, and it is the guest kernel's responsibility to create the pseudo-physical address space from this. Finally, it also supports memory ballooning as a measure to minimize guest footprints.

Virtual Resources: provided by Xen are virtual network interfaces (VIFs) and virtual block devices (VBDs), each visible to a guest through an idealized interface. A shared memory *producer-consumer* ring is used for each direction of data transmission. Each entry in the ring contains metadata related to the operation and a reference to page(s) containing related data. The producer entries notifies the consumer of new data through a lightweight asynchronous notification mechanism. Because the notification mechanism is asynchronous and can be used independently of placing data on the ring, efficient batching is possible and happens automatically during high load.

Further, ring entries can be handled out of order by the consumer, which enables *class-of-service* disciplines to be applied.

2.2.3 Control and Driver Domains

To leverage existing drivers, Xen offloads hardware device drivers to the control domain *domain 0* and optionally trusted driver domains. Existing drivers from well tested systems like Linux can then be leveraged, improving Xen's functionality and portability. In addition, by using dedicated driver domains, hardware and device failures can often be contained within the trusted driver domain. This domain can also be restarted when the failure has been detected, with only slight impact on other guest domains.

Domain 0 (Dom0) is a trusted paravirtualized domain. It manages virtualization through the VMM APIs and typically provides hardware device drivers and other functionalities supporting device back-ends. Dom0 covers most aspects of virtualization management such as providing control tools for starting and stopping guests, building initial guest memory images from disk images, and managing virtual networking by bridging virtual and physical network interfaces.

2.2.4 Xen Hypercall Internals

Hypercalls are identified by their numbers defined in `xen/include/public/xen.h`. For instance the number of the call `sched_op` is `__HYPERVISOR_sched_op`, defined as 29 in `xen.h`. Often, the Xen routine that handles the hypercall `acm_op` is called `do_acm_op`. One can find the association between numbers and names in the `hypercall_table` in `arch/x86/x86_32/entry.S`.

For the subroutine or hypercall handler of the hypercall `__HYPERVISOR_xxx`, usually it is `do_xxx`. For example, the handler of `__HYPERVISOR_sched_op` should be `do_sched_op`. The `do_xxx` routines are entries of the `hypercall_table` defined in `xen/arch/x86/x86_{32,64}/entry.S`.

A guest domain invokes an entry in the `hypercall_page` to issue a hypercall. The `hypercall_page` is initialized in `hypercall_page_initialise(void *hypercall_page)` at the time when the control domain creates the guest domain. `Hypercall_page` is actually a code page containing 32 hypercall entries. Every entry includes two instructions:

```
"mov $__HYPERVISOR_xxx,%eax
```

```
int $0x82 "
```

The assembler instruction `INT 0x82` causes an exception which triggers a hypercall entry in *entry.S* which then indexes the hypercall table based on the hypercall number passed the `%eax` register to invoke the corresponding hypercall handler.

As is the case for syscalls, the arguments are passed by *fastcall* (in registers rather than on the stack): the hypercall number is in `%eax`, and up to five arguments (depending on the hypercall) may be passed in `%ebx`, `%ecx`, `%edx`, `%esi` and `%edi`. Xen ensures that the interrupt descriptor table (IDT) only allows the interrupt to be executed from within ring 1, so that guest kernels can make hypercalls but guest userspace applications cannot. Most hypercalls return sensible error codes as defined in `errno.h`.

2.2.5 Hypercall Invocation

Figure 2.5 shows the hypercall invocation trace.

2.3 A Security Model for Xen

Virtualization security can be considered from two perspectives. The first (also more common) regards to how virtualization can improve security of a system. Several projects take advantage of the isolation and supervising properties of the VMM to improve security of operating systems. On the other hand, the presence of the virtualization layer also opens new attack vectors which do not exist in the conventional OS. Since the focal point of this thesis is to improve the security of a virtualization platform, we only focus on the second perspective. Specifically, potential security threats and existing security measures in the context of Xen virtualization Xen will be analyzed. The analysis results form a basis for us to identify security problems and to suggest defense measures.

2.3.1 Xen Security Model Overview

Figure 2.6 illustrates the generic model for Xen security. The virtualization platform consists of the following components:

- Host hardware
- The virtual machine monitor

2.3. A Security Model for Xen

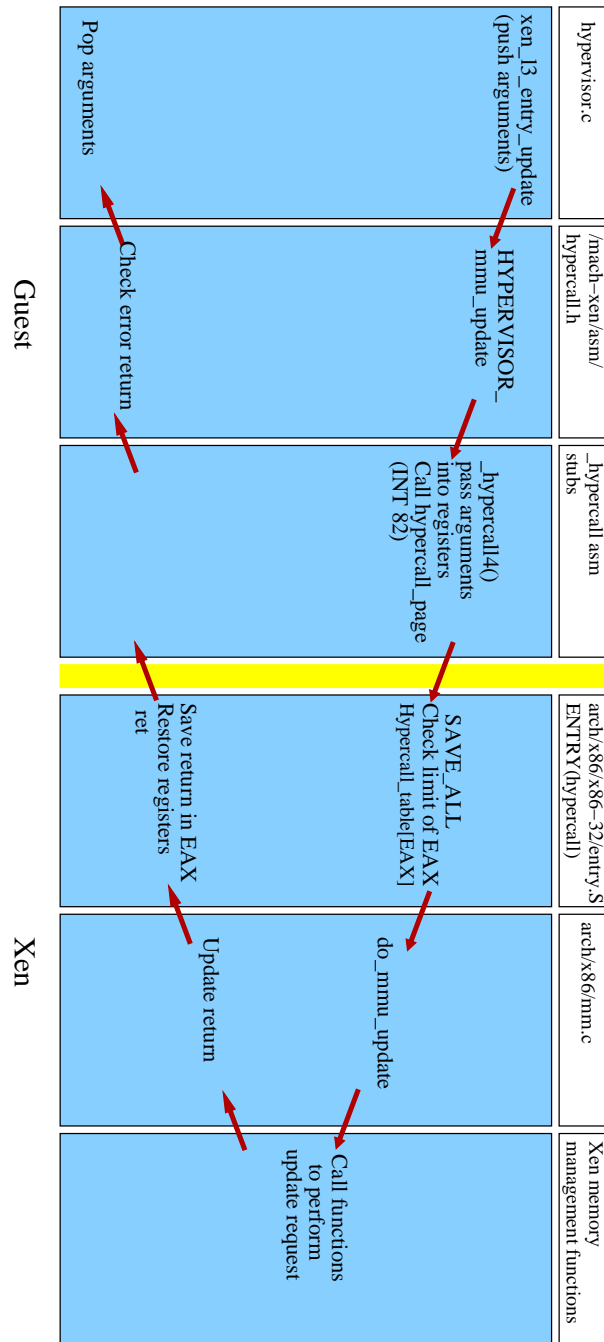


Figure 2.5: Hypercall Diagram

2.3. A Security Model for Xen

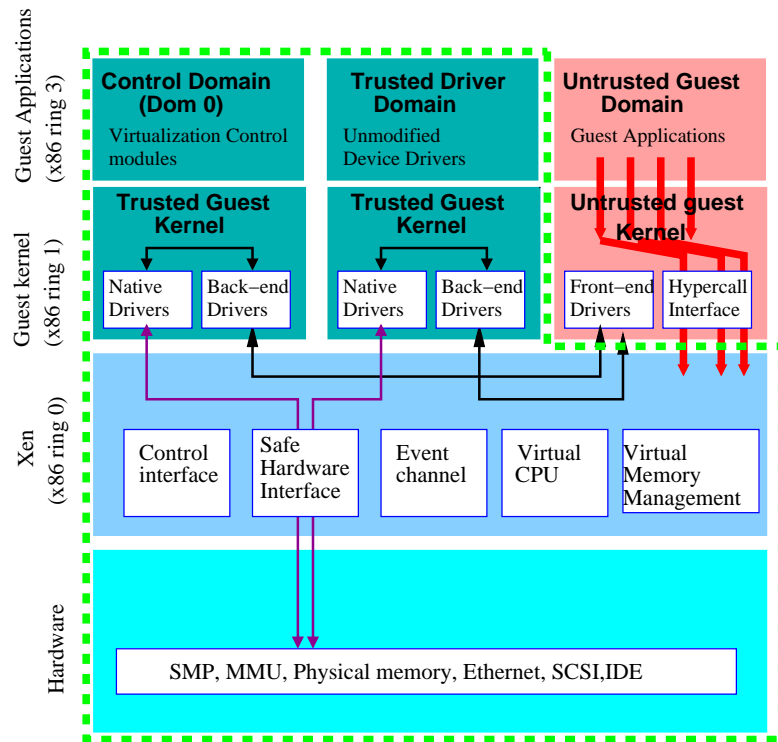


Figure 2.6: Xen Security Boundary

- Trusted or privileged domains providing control and management software and/or hardware drivers (i.e., domain 0 and other driver domains)
- Normal (untrusted) guest domains which use virtualized resources and services provided by the VMM and other trusted domains

2.3.2 Security Boundary and Points of Attack

In our security model, entities inside the security boundary (dashed line in Figure 2.6) are trusted and their correct and secure functioning is critical to the overall system security. Trusted entities inside the security boundary include host hardware, the VMM, privileged domains and virtualization management and control software. A compromise of any of these entities is unacceptable for Xen security. Although there may be file servers and external management hosts in some Xen systems, we do not consider them in our model. Given this, there are four relevant points of attack:

- VMM interface
- Device drivers
- Network traffic
- Hardware

VMM Interface The Xen VMM interface provides entry points where guests can interact with Xen and other guests. A guest usually consists of a normal kernel and software applications. We cannot make any assumption about the security of the kernel and its application codes. As in a normal OS context, an attacker may have full control over both guest kernel and guest application codes, thus being able to attack the VMM and other domains through the VMM interface. The Xen VMM interface available to guests includes virtual resources represented by *front-ends* and *hypercalls*.

The *front-ends* include memory-mapped devices, devices residing in the x86 I/O space, or devices accessed through a virtual bus such as PCI or USB. The *back-ends* are typically provided by privileged domains. Virtual storage may be represented as a file in the domain file system, and a virtual network device may be represented by a virtual back-end network device which is connected to the physical device in the control domain. A guest may also be granted direct hardware access to devices. In such cases the VMM is still

involved and it may hide devices from the guest, or attempts to control DMA parameters (or use an IOMMU) from an unsafe DMA initiated by the guest.

Xen also provides explicit VMM services for domains through *hypercalls* with details mentioned in the above sections. Hypercalls are used by both trusted and untrusted domains for critical and highly privileged requests such as memory management requests. Xen typically performs sanity checks on such requests.

Network Traffic Although, this thesis does not worry about network attacks, it worths to put security threats coming from network traffic into perspective. In theory, an attacker is able to intercept, modify, discard and generate traffic unless specific protection measures like physical protection are taken. The attacker may also be able to break the virtual network infrastructure. There have been research about how to protect virtual network such as [44].

2.3.3 Assets to Protect

Assets are defined from the viewpoint of the administrator and the users or groups on non-hostile guests running on Xen. Assets are defined only if they may be damaged by attacks using the identified points of attack. Categories to protect include *confidentiality*, *integrity* and *availability*.

Virtualization Metadata: Virtualization metadata consist of all types of data required to run guests and maintain the virtualization system such as:

- The VMM configuration information and its operational data structures.
- The virtual storage and file systems
- Global and per-guest configuration data for the virtualization mechanism
- Cryptographic keys and other relevant security data for external communication for management, with other VMMs and external storage
- Virtualization snapshots and replay logs for storage, execution, and memory state

- Event logs related to virtualization

Both confidentiality and integrity are important for virtualization metadata, but their relative importance depends on the type of metadata. For instance, integrity breaches of configuration data are potentially catastrophic, while integrity breaches of event logs cannot typically be used to escalate privileges. Management of cryptographic keys, on the other hand, requires both strong confidentiality and integrity.

Availability of Services and Management: The primary purpose of a virtualization platform is to provide services for guest users. This implies availability of all relevant resources such as CPU time, memory, storage and network bandwidth. Although there may not be a strictly guaranteed service level agreement, guest users would expect some level of availability. From an administrator viewpoint, he needs to be able to manage the virtualization platform at all times.

Guest Data: Guest data consists of storage contents and the current execution states of the guest including memory footprints and the current CPU state. From a guest user viewpoint, guest data includes all data controlled and owned by the user. Both confidentiality and integrity of guest data are important. Integrity breaches of guest data are not as serious as for metadata under the assumption that a guest is potentially hostile, and trust the VMM to combat any attempts to escalate privileges.

Hardware: Host hardware includes the motherboard and all directly connected peripherals. It is possible to damage host hardware through software. Details about host hardware should be minimized to the attacker. Both confidentiality and integrity of host hardware are important.

2.3.4 Specific Threats:

Specific threats mentioned here cannot be exhaustive, but they support a high level view of threats to the identified assets.

Attack Virtualized Memory: Although the guest kernel cannot directly modify memory data structures such as page tables, global descriptor tables or local descriptor tables. These operations can be requested through hypercalls. If an attacker could tamper with hypercalls, he potentially can

cause some damages such as elevation of access rights to pages not intended for this guest. Then he can access memory pages of other guests, or overwrite the VMM code. An attacker could also cause memory DoS against legitimate VM users.

Attack Front-ends and Back-ends: Both virtual resource front-ends and back-ends may be vulnerable. The front-end interface can be quite complex because it represents a real hardware device such as a network interface. For performance efficiency, the front-end interacts with back-ends using low-level memory mechanisms such as page flipping. Although a guest cannot access back-ends directly, the guest may be able to force specific back-end operations to occur by using the front-end in a certain manner. The goal in such attacks is to exploit flaws in the back-end software, and, for instance, to get access to protected information or to execute arbitrary code in the context of the backend device.

Attack Hypercalls: As mentioned above hypercalls are similar to native syscalls, with potentially equivalent flaws. From a security viewpoint, the hypercall interface is a convenient vehicle for an attacker to carry other attacks. If an attacker can compromise a guest OS to launch malicious hypercalls, he can cause serious security problems to the entire virtualization system. In a full virtualization system such as VMWare, there is also the concept of VMM API in the form of *virtualization shortcuts* for guest domains. Flaws in such shortcuts can potentially open security breaches similar to Xen hypercalls.

Other Possible Exploits:

DMA Access: In basic x86 hardware, DMA (Direct Memory Access) initiated by a hardware device cannot be filtered. A guest domain can take advantage of DMA to perform attacks from below. To protect against such attacks, the VMM needs to be able to filter access to hardware, control and eliminate malicious hardware access. Fortunately, the modern x86 hardware architecture includes an IOMMU for controlling DMA accesses.

Compromise Cryptosystems: Replaying a virtual machine may be useful for diagnosis or recovery but this may create problems for some cryptographic algorithms. For instance, it is required that the nonce value of a stream cipher should never be reused. If the VM is reset to a previous

state, it may repeat such values. An attacker may exploit this property to perform attacks against the current cryptosystems.

Side Channel Attacks: Virtualization of CPU and other resources may reveal information about timing, memory usage and computation speed. Such information can be exploited by the attacker to establish side channels or covert channels. The fact that multiple VMs shares the same hardware makes side channel or covert channel attacks easier. Without explicit hardware support for elimination of timing side channels, hardware sharing is unsafe. Fundamentally, a virtualization architecture may minimize bandwidth of side channels, or avoid their impact by ensuring that side channels only exist between mutually trusting entities by access control policies such as Xen MAC [44].

Migration Threats: As a VM is migrated from one host to another, its data such as the kernel, file system and memory images are sent over a potentially unprotected network. Compromising either the confidentiality or integrity of such data is a security problem. In addition, after migration the execution environment of the VM may have changed in several components such as processor, memory and device drivers. Such changes can be detected to some extent and they may affect guest behaviors in performance and computation. A sneaky attacker may be able to observe these subtle changes to carry out some side-channel attacks.

2.3.5 Current Security Mechanisms in Xen

In this section, we discuss specific security mechanisms available to counter threats to virtualization assets. These measures provide concrete examples of approaches for improving virtualization security. The role of a security mechanism is to diminish the possibility or impact of specific threats, thus making them less appealing for an attacker.

Processor Privilege Levels: Processor privilege levels provide basic protection against guest attempts to escape isolation or escalate privilege. In the x86 architecture, there are four privilege rings, from ring 0 to ring 3. Ring 0 has the most privileges, and can access all instructions and memory addresses. The operating system usually runs in ring 0. Ring 1 and 2 have less privileges and are used for operating system services. Ring 3 has the least privileges and is used for user applications.

Xen exploits these privilege rings to allow multiple operating systems to run at the same time. Xen itself runs in ring 0. The guest kernels run in ring 1. Any calls which would previously have required ring 0 to operate are replaced with hypercalls to Xen. Guest applications still run unmodified in ring 3.

Hardware Support for Containment: Hardware may provide mechanisms for enforcing containment. Modern x86 hardware has an IOMMU which allows hardware DMA to be contained, thus preventing *attack from below* attempts. Intel LaGrande (LT) and AMD Secure Execution Mode (SEM) also provide hardware support for containment purposes.

Cryptographic Protocols and Secure Networking: Management, VMM, and storage traffic may be protected using a packet-based encryption system, such as, IPsec, or a stream-based encryption system such as TLS or SSH. Individual data elements inside management and VMM protocols can be protected using an application level mechanism like PGP. Xen virtual networks security could be enhanced by common mechanisms such as fire-walling, VLANS or services classification.

Mandatory Access Control: Mandatory access control (MAC) mechanisms can help to minimize the possibility of configuration errors, which lead to an unintentional violation of isolation goals. For example, a MAC policy may state that VMs with different security labels cannot share any resources. Mandatory access control does not necessarily provide any new security or isolation mechanisms, but it helps ensuring that existing mechanisms are used with minimal risk of human errors. The IBM sHype project has developed mandatory access control within the context of Xen [44].

Trusted Platform Module (TPM): The Trusted Platform Module (TPM) enables new security models and may enhance existing ones by offering facilities for the secure generation of cryptographic keys. It also includes capabilities such as remote attestation and sealed storage [24]. TPM can provide better authentication between VMs and between virtualization platforms. Projects such as Terra [24] utilize the concept of TPM to build a trusted VMM (TVMM) which partitions tamper-resistant hardware platform into multiple, isolated virtual machines (VM) providing the appearance of multiple boxes on a single, general-purpose platform. In this model, the software

stack in each guest VM can be tailored from the hardware interface up to meet the security requirements of its applications.

2.3.6 Discussion about Protecting Xen

We can observe that the threats coming from exploiting the VMM APIs or hypercalls appear to be real and imminent. Other threats are either very hard to carry out (i.e., covert channel attacks) or can be protected against by the current security mechanisms (i.e., cryptographic encryption and securing networking can protect virtualization data and network traffic; MAC may prevent denial of service attacks). On the other hand, existing security measures can not be directly applied to protect hypercalls. For instance, both MAC and TPM can greatly enhance security for inter-VM interactions but do not protect the VMM against internal hypercalls attacks from guest VMs. This motivates us to look for different measures to protect potential hypercall attacks.

Chapter 3

Related works

Hypercalls in the virtualization world are similar to system calls in the OS world not only in terms of semantics but also in terms of implementation. While security incidents of hypercalls are still rare in the public, there is a plethora of publications related to syscall attacks and syscall defenses. This chapter covers a literature survey of the security of syscalls from both attack and defense perspectives. This survey gives us insights into the issues and the approaches to protect hypercalls.

3.1 System Calls

System calls (syscalls) are the sole mechanism to access kernel resources and the sole means for cross-layer communication from user mode to kernel mode. Syscalls are convenient entries where an attacker can exploit to invoke malicious attempts on the host OS. According to recent CERT advisories [1], most OS security violations involve the invocation of system calls at some point. Consequently, syscall attacks pose an immense threat to OSes.

3.2 Syscall Attack Overview

What Is a Syscall Attack? A syscall attack is characterized by an explicit attempt to invoke syscalls to cause damage to the host OS. In these attacks, the invocations of syscalls can be either direct or indirect. Syscalls are essential intermediate means to carry out malicious objectives. An attacker can use syscalls to hijack the program's flow of control, to gather critical system information of resources, usurping and escalating the current privilege, or opening doors for subsequent attacks. The intuition that most attacks could cause real damage to the system by exploiting system calls makes this interface an ideal point to detect or prevent these attacks.

How a Syscall Attack Is Carried Out? A syscall attack is generally carried out in three phases. First, an attacker finds a suitable software bug

to enter a system. These software bugs are usually memory-safety-related (i.e., buffer overflow, heap corruption, integer overflow and format string problems) which lead to certain parts of memory of the victim process susceptible to be overwritten. Subsequently, the attacker tries to take over the program's control logic by overwriting a certain code pointer or a specific memory location, making it point to malicious code of the attacker's choosing. This code is either malicious shell code that has been injected into the victim process' memory, or existing functions in the standard libraries (i.e., *libc*) or elsewhere in the program code. Finally, the attacker is able to compromise the victim process by illegally executing system calls and potentially gains control of the system.

3.3 Classes of Syscall Attacks

Syscall attacks can be classified based on attack targets and exploitation strategies.

3.3.1 Attack Targets:

attack targets are where the attacker can cause memory corruption or to compromise the control logic of the program. There are two kinds of attack targets: control-data (or code pointers) and non-control data.

Control Data Attacks: the attacker targets control data - data which are loaded into the program counter during program execution such as *return addresses* and *jump pointers*. Because control data are transition points where a change of control flow are possible, most syscall attacks try to alter the target program's control data in order to execute injected shell code or out-of-context library code. Subsequently, syscalls are invoked with the privilege of the victim process. Examples of control-data attacks include *stack pointers*, *old base pointer*, *function pointers* and *longjmp pointers* [65].

A survey of the CERT/US-CERT security advisories [1][58] and the Microsoft Security Bulletins [11] shows that control-data attacks are usually considered to be the most critical security threats. According to [2], despite the facts that control-data attacks are well known and even well-studied, new attack instances are continuously crafted to bypass existing defense techniques.

Non-control Data: Non-control data attacks target other types of data instead of code pointers to achieve their malicious objectives. Studies in [14],[57],[60], [67] show that non-control data such as *configuration data*, *user input*, *user identity data*, *decision-making data* can be exploited to compromise the system. Chen’s study [53] provides experimental evidence to prove that non-control-data attacks are realistic and can generally target real-world applications.

Examples in [53] include demonstrated attacks which exploit buffer overflow, heap corruption, format string, and integer overflow vulnerabilities without injecting any new attack code or altering the program control flow. All the non-control-data attacks constructed there resulted in security compromises as severe as those due to traditional control-data attacks.

3.3.2 Exploitation Strategies Based on Modification of Control Flow:

Direct Modification: the most straightforward and commonly utilized approach for the malicious code is to directly invoke syscalls by itself. In these attacks, *shellcode* usually presets syscall number and arguments and then executes the instruction that raises a software interrupt for system calls. These attacks are simple and does not require much skills. Early types of code injection attacks [8] belong to this category.

Since these attacks directly overwrite control pointers and directly alter *the control flow* of the program, they can be easily prevented or detected by simple techniques to protect the return address such as StackGuard [15], PointGuard [14] or to make the stack non-executable [66].

Indirect Modification: another approach is to jump into code available in the targeted software to launch an attack. The two most common are *return-into-library* and *return-into-the Process Linkage Table (PLT)* attacks. In return-to-library attacks, the injected code jumps to an important function provided by a standard library (e.g., libc), it can then obtain access to services of the OS kernel since these library function indirectly lead to system calls.

The attacker can also target returning into the Process Linkage Table (PLT) [41, 46] since the PLT is used to resolve libc and other function addresses automatically. Return-to-library attacks can defeat non-executable memory protection and return-into-PLT can defeat the randomization of the base address. It should be noted that it is much more difficult to carry out these attacks because they require specific understanding of the system and libraries. However, it is also harder to detect and defend against these attacks compared to direct control flow modification.

In general, attacks involving control flow modification (even indirect attacks) will cause certain anomalous behaviors for the running program. Hence, they give away some clues for detection systems based on control flow integrity.

Non-modification: this type of attack produces a legitimate sequence of system calls while performing malicious actions. The attacker usually interleaves the real attacking code with innocuous code, thereby impersonating a legitimate sequence of actions. Mimicry attacks [62] are typical examples of this class.

Mimicry attacks can be further divided into *global mimicry attacks* and *local mimicry attacks* considering the minimum set of system calls necessary for the functionality of an application function. The system call sequence in a global mimicry attack combines the legal system calls of multiple functions, while a local mimicry attack uses the legal system calls of only the running function.

Mimicry attacks are the current challenges of the research community. The major problem is that system-call-based intrusion detection systems are not designed to prevent attacks from occurring. Instead, these systems rely on the assumption that any activity by the attacker appears as an anomaly that can be detected. While the ability to invoke system calls might be significantly limited, arbitrary code can be executed. This includes the possibility to access and modify all writable memory segments.

In general, mimicry attacks require very comprehensive policies which include not only the sequence of system calls but also many other additional information such as function arguments, program counters and runtime call stacks, to detect and to respond. However, although additional

information undoubtedly complicates the task of the intruder, the extent to which the attack becomes more complicated is not clear. In addition, it is usually not practical to include all of this information in the detection reference model for performance and complexity reasons.

3.4 Syscall Defense Challenges

The seriousness of syscall attacks with an increased frequency is evident. The research community also has proposed different defense mechanisms to combat syscall attacks. Although many solutions have been developed, the problem is hardly solved. There are several factors that hinder the development of effective solutions for syscall defense:

Lack of Type-safe Properties Enforcement: most software exploitations violate the type-safety property of programming languages. Software would be undoubtedly much less vulnerable if type-safe languages can be enforced. Unfortunately, type-safe languages and compiler techniques, such as CCured [45], Cyclone [33], and SAFECODE [18], are designed to achieve type safety through reimplementing software or recompiling legacy programs. However, rebuilding existing software with type-safety techniques requires a tremendous amount of effort. In addition, type-safety of an application is usually achieved by hiding type-unsafe behaviors in low-level software components, such as the Java VM, C library, and OS kernel. These components themselves are not free of memory bugs.

Preventing Transferring in The Program's Control Flow Is Not Possible: many entities participate in transferring control in a program execution. Compilers, linkers, loaders, runtime systems, and even hand-crafted assembly code all have legitimate reasons to transfer control. Program addresses are credibly manipulated by most of these entities, e.g., dynamic loaders patch shared object functions; dynamic linkers update relocation tables; and language runtime systems modify dynamic dispatch tables. Generally, these program addresses are intermingled with and indistinguishable from data. In such an environment, preventing a control transfer by malicious code by stopping illegal memory writes is hard.

Existing Solutions Are Only Created to Partially Solve The Problem: previous sections indicate that the attacker can indeed compromise

many real world applications with or without breaking their control flow integrity with variety of different techniques. In addition, an attacker can exploit a vulnerability that gives him random write access to arbitrary addresses in the program address space. Detection and prevention techniques based on the flow integrity simply fail to deal with mimicry or some of the non-control data attacks [50].

Difficulties in Adopting The Existing Solutions: barriers in implementation and deployment, compatibility, performance overhead, complexity, false positives and false negatives are all issues that hinder the development and adoption of a good solution. It is very difficult to devise a satisfactory scheme that can meet all sometimes conflicting requirements.

3.4.1 Classification of Syscall Defenses

Syscall defenses can be classified as *proactive* and *reactive* mechanisms.

Proactive mechanisms attempt either to eliminate the possibility of syscall attacks or to limit the scopes of such attacks. A proactive approach can never be 100% effective. However, effectively preventing certainly decreases the frequency and strength of attacks. Moreover, proactive defense is inherently complementary to or compatible with other defense approaches. Proactive mechanisms can be grouped into proactive preventions of: code pointer overwriting, the execution of injected malicious code and the spreading of attacks.

Reactive mechanisms strive to alleviate impacts of syscall attacks on the system. To attain this goal, reactive methods need to *detect* and *respond* to attacks. The primary focus is to detect every attempted syscall attack as early as possible with a low degree of false positives/ negatives. Then the attack patterns are characterized and response measures are taken. The detection process requires actively monitoring the system. Since detection is the main focus of most reactive approaches, we use detection methods as criteria to classify reactive defense. In terms of detection mechanisms, there are *misuse detection* and *anomaly detection*.

3.4.2 Proactive Mechanisms

i) Protecting Against Code Pointers Overwriting

Overwriting code pointers to alter *the control flow* is a common step in traditional syscall attacks. The preventive methods used to protect the integrity of control information data can be classified into *static prevention* and *dynamic prevention*.

Static Prevention: Static preventive methods try to remove security-related bugs in the source code by using static analysis tools. Removing all security bugs from a program is considered infeasible. Nevertheless, removing notable bugs such as those causing buffer overflow could significantly reduce the number of known attacks.

Dynamic Prevention: Dynamic (run-time) preventive methods aim to change the run-time environment or the system functionalities to make vulnerable programs harmless, or at least less vulnerable. This means that in an ordinary environment the program would still be vulnerable and security bugs may still exist. However, those vulnerabilities cannot be exploited in run-time environment because it is being protected by these dynamic preventive methods.

In this class, *compiler techniques* are used to prevent attacks by verifying and enforcing type safety or by setting extra protection for code pointers. Examples of most popular compiler techniques include StackGuard [15], StackGhost [23], PointGuard [14], safe C compilers such as CCured [45] and [34, 45, 52, 68], Libsafe [59], LibsafePlus [7].

Perhaps the earliest method of control pointer protection comes from StackGuard [15]. It encrypts control information in a stack by XOR-ing it with a random number. StackGuard causes malicious control information to be decrypted into meaningless values, resulting in the failure of the attackers attempt to acquire control of vulnerable software. Several attempts to bypass StackGuard are suggested in [50] and since then several more sophisticated methods have been proposed in StackGhost [23] and PointGuard [14].

ii) Memory Randomization

The principal goal of memory randomization techniques is to rearrange the memory layout so that the actual addresses of program data are different in each execution of the program. Address Space Layout Randomization (ASLR) is a popular randomization technique which involves arranging the positions of key data areas, usually including the base of the executable and position of libraries, heap, and stack, randomly in a process's address space. ASLR relies on the low chance of an attacker guessing where randomly placed areas are located. Security is thus increased by the increase of the search space. Other examples of memory layout randomization are [6, 10, 12].

Preventing of The Execution of Injected Malicious Codes Typically, techniques to prevent the execution of injected malicious codes include randomized instruction sets, non-executable pages, and program address obfuscation. In addition, there are also techniques to prevent the direct invocation of system calls and techniques to prevent library injection such as return-into-library or return-to-PLT attacks.

Randomized Instruction Sets: creates process-specific randomized instruction sets (e.g., machine instructions) of the system installed potentially vulnerable software. By randomizing the underlying system's instructions, *foreign* code introduced by an attack would fail to execute correctly, regardless of the injection approach. Examples of instruction sets randomization techniques can be found in [15, 36].

iii) Containing Scopes of Attacks

Process sandboxing is perhaps the best understood and widely researched area of containing bad code or containing the scope of attacks. *User-level sandboxing* systems such as Janus [28], Consh [5], Mapbox [4] and OpenBSD's systrace [48] operate at user-level and confine applications by filtering access to system calls. To accomplish this, they rely on `ptrace(2)`, the `/proc` file system, and/or special shared libraries. Another category is *kernel-level sandboxing* systems such as Tron [9], SubDomain [16] and others [23, 25, 42, 61]. These systems intercept system calls inside the kernel, and use policy engines to decide whether to permit them or not.

The main problem is that the attack is not prevented in these system. Rather, they try to limit the damage such code can do. Thus, the system does not protect against attacks that use the compromised process' privileges to bypass application specific access control checks (e.g., read all the pages a web server has access to), nor does it protect against attacks that simply use the subverted program as a stepping stone, as is the case with network worms.

3.4.3 Reactive Mechanisms

Reactive mechanisms attempt to detect and respond to attacks. Generally speaking, the detection phase is more important and is the main focus of most research. In this section, we only summarize research on the detection techniques.

i) Misuse Detection

Misuse detection techniques model attacks on a system in specific patterns or signatures, then systematically scan for the occurrences of these patterns. The modeling process involves a specific encoding of previous behaviors and actions that were deemed intrusive or malicious. Misuse detection can be classified by *signature-based* and *specification-based* techniques.

Signature-based it is assumed that intrusive events exhibit certain distinguishing patterns from normal events. These patterns form signatures for the intrusive events. These signatures can be derived by studying past attack activities. One of the limitations of signature-based approaches is that they can only detect known attacks but fail to recognize new attacks. In addition, signatures for some attacks may be too difficult to formalize. Examples of signature-based misuse detection approaches include expert systems [13, 40], model-based reasoning [26, 39], state transition analysis and keystroke dynamics monitoring [30, 31]. The strength of this approach is that it incurs no false positive.

Specification-based similarly to the signature-based approach, data about the system' previous activities are collected and analyzed to detect any deviation from the expected system behaviors. System abnormal behaviors will then be specified. These specifications do not need to be exact matches but

rather approximate definition of attack patterns to distinguish between the benign and the malicious. Ko et al. [37, 38] have designed a program policy specification language based on predicate logic and regular expressions to describe the expected behaviors of privileged programs. Misuse is detected by any deviation from these specifications. Although specification-based systems can detect unknown attacks slightly deviating from the previous attacks, runtime detection demands efficient execution of specifications, and efficiency has been an issue for specification language design.

ii) Anomaly Detection

Anomaly detection schemes focus on *normalcy* and first try to characterize a profile of *normal* behavior. They can directly address the problem of detecting novel attacks against systems. This is possible because anomaly detection techniques do not scan for specific patterns, but instead compare current activities against statistical models of past behaviors. Any activity sufficiently deviant from the model will be flagged as anomalous, and hence considered as a possible attack. Anomaly detection schemes are based on actual user histories and system data to create its internal models rather than predefined patterns. In addition, these statistical models of past normal behavior are computed at run-time which is the key difference between anomaly detection schemes and specification-based ones.

Early syscall anomaly detection methods only model syscall traces or the invocations of syscalls. Typically, the N-gram method characterizes program behavior using fixed-length syscall sequences [22, 29]; data mining based approaches generate rules from syscall sequences [40]; Hidden Markov Model (HMM) and Neural Networks were used in [54, 63]; algorithms originally developed for computational biology were also introduced into this area. Wepi builds variable-length syscall patterns based on the Teiresias algorithm which is originally developed for discovering rigid patterns in unaligned biological sequences [64]. However, there has not been much improvement on modeling syscall traces based methods recently in part because system calls themselves only provide limited amount of information. Invoking syscalls is only one aspect of program behavior.

Other methods use the program counter (PC) to construct states. The PC indicates the current execution point of a program, because each instruction of a program corresponds to a distinct PC, this location information is useful for intrusion detection. The call graph model Wagner et al. pro-

posed models the expected syscall traces using static analysis of the program code [60]. The global control-flow graph is naturally transformed to a non-deterministic finite automaton (N DFA). The N DFA can then be used to monitor the program execution online. If all the non-deterministic paths are blocked at some point, there is an anomaly. Gifn et al. refined the ideas behind the above model [27]. Their approach applies static analysis on binary executables and uses null calls to avoid non-determinism. Instead of statically analyzing the source code or binary, Sekar et al [56] proposes a method to generate a deterministic finite state automata (FSA) by monitoring the normal program executions at runtime.

In addition to the current PC, recent methods also use the call stack and call sites. A lot of information can be obtained about the current status and the history or future of program execution from the call stack, particularly in the form of return addresses. The VtPath method [21] utilizes the return addresses to generate the abstract execution path between two program execution points, and decides whether this path is valid based the normal runs of the program. Gaugav further extends this idea to build an effectively non-executable stack and heap to prevent libc-based code injection attacks [35]. Rajagopalan and Linn utilize binary rewriting tools to embed cryptographic information to protect call sites and call arguments to effectively prevent code injection attacks [43, 49].

All the methods above have advantages and disadvantages. Modeling syscall traces based on AI approaches can provide more exhaustive range of protection as well as capture new attacks. However, they are complicated to build, incur high performance overhead and produce high false alarm rate. Other methods avoiding using complex modeling may have better performance and less false alarm rate but ineffective against certain new attacks and provide less protection coverage.

3.5 Syscalls vs. Hypercalls

From a security viewpoint, hypercalls offer some similarities to syscalls from both attack and defense standpoints. For an attacker, when he can tamper with applications and guest OSes, it is his goal to reach to a higher privilege level to be able to cause some real damage to the system. Both hypercalls and syscalls are convenient entries where privilege escalation may happen. Specifically, a syscall can be exploited to jump from the application level to

3.5. *Syscalls vs. Hypercalls*

the OS level and a hypercall can be used to change from the guest OS level to the VMM level. For a defender, comparably to syscalls, hypercall interface is the ideal place to detect and to respond to an ongoing attack because when there is a hypercall attack, hypercalls are the most likely place where control transfer may happen or where the attacker may reveal his malicious traces. As a result, a study of syscall attack and defense offers insights and necessary background to build hypercall protection mechanisms.

There are differences between protecting syscall attacks and hypercalls attacks however. For syscall attack, a tampered application is the standpoint from which an attacker can launch syscall attempt. Whereas, for hypercall attack a compromised guest OS is where an attacker can launch an attack. Compared to an application, a guest OS in the virtual world is a different and more autonomous entity. For instance, an application does not have the privilege to reboot itself without being noticed. Conversely, a guest OS has the capability to reboot (e.g., rebooting considered as one of the OS normal update mechanisms). The rebooting process when all protection measures are turned off may offer a golden opportunity for malicious code e.g. starting a new module which is inactive otherwise or executing a snippet of code to gain control over a certain part of the system...etc. In short, conventional wisdoms in protecting compromised applications may not work for a compromised guest OS.

Chapter 4

Prototype Design and Implementation

4.0.1 Hypercall Protection Considerations

According to our analysis in the previous chapter, we took into consideration the following points:

- We would apply reactive approaches to hypercall protection. These approaches include detecting and responding to hypercall attacks.
- Although misuse detection methods in general have lower false alarm rate and better performance, we are not aware of any specific real-scenario hypercall attack, therefore misuse-based detections would not be useful. Anomaly detection methods appear to be more helpful to detect the unknown attacks and hence they are more appropriate for hypercall protection.
- Modeling hypercall traces may not be as useful as modeling syscall traces. We observe that popular syscall anomaly detection approaches modeling syscall traces such as N-gram methods [22, 29] or other AI-related ones [40, 54, 63] are based on the concept of a *sequence* in which an attacker has to perform a series of syscalls to carry out a successful attack (e.g., obtain the handle, open the file, write to the file ..., etc.). Such a sequence concept may not apply in the case of hypercalls since a hypercall does not necessarily exhibit any pattern of sequence or an attacker may need only one hypercall to mount a successful attack.
- We observe that approaches utilizing existing information such as call sites and call stack to protect against syscall attacks [35, 43, 49] can be useful for hypercall protection and they are simpler to adapt. These approaches are simple yet powerful enough to satisfy our requirements for simplicity, performance and portability.

4.1 High Level Overview

We focus on protecting hypercalls at the VMM level. Our assumption is that all guests initially start with clean kernels - they are infected with malware which can launch malicious hypercalls. We aim to protect against the security incident in which a well-behaved guest OS is compromised and used as a platform to launch hypercall attacks to the VMM and other guests. Our protection measures would prevent such an incident from happening.

Our protection measures do not improve the current security conditions of guests. Guest applications are not necessarily more secure with our protection measures, but they would be less secure to run on the virtualization platform without being protected. In another words, the security measures are designed to prevent the current status from getting worse. This approach essentially provides security by detection and prevention; that is catching the *bad* guys and proactively blocking them to protect the assets.

Based on our analysis in the previous chapter, we adopt two different approaches to protect against hypercall attacks. They are authenticated hypercalls (MAC) and hypercall access table (HAT). The implementation of these approaches typically involves three steps: instrumentation of guest kernels and/or the VMM, hypercall interposition and authentication of hypercalls. We first describe the authenticated hypercalls and the hypercall access table. Then we detail the steps of implementation. Finally, for an analytical purpose, the stack trace walk technique is analyzed and explored as an alternative way to authenticate hypercall with information extracted from the call stack of the hypercall invocation.

4.2 Authenticated Hypercalls and Hypercall Access Table

4.2.1 Authenticated Hypercalls

Authenticated hypercalls are used for detecting and containing compromised guest OSes. An authenticated hypercall is a hypercall augmented with extra arguments that specify the *policy* for that call and a cryptographic message authentication code (*MAC*) that guarantees the integrity of the policy and the hypercall arguments. Xen uses this extra information to authenticate the hypercall.

4.2. Authenticated Hypercalls and Hypercall Access Table

Policies: a policy is defined as a set of verifiable properties of a hypercall request. Our current prototype enforces hypercall policies which include the hypercall number, the call site and constant parameters (e.g., integer constants). If a policy does not give a value for a parameter, then this parameter is unconstrained and any value is allowed. These policies are as expressive as those used by Rajagopalan [49] to protect syscalls and reasonably expressive enough compared with policies published by most well-known syscall monitoring systems such as Systrace which includes only the call site and the syscall number.

Policies examples: our current prototype enforces hypercall policies of the following form:

```
Permit HYPERVISOR_update_descriptor from location 0xc0101347
```

This policy says that a guest OS can invoke the *HYPERVISOR_update_descriptor* hypercall from a call site at memory address 0xc0101347 inside the guest OS kernel.

Policy Descriptors: the policy descriptor is a single 32-bit integer value that describes what parts of the hypercall are protected by the MAC.

MAC Computation: the MAC is computed over the *encoded policy*, i.e., a byte string that is a self-contained representation of the policy. The encoded policy is built by concatenating the hypercall number, the address of the call site and the policy descriptor. For example, for the policy

```
Permit HYPERVISOR_update_descriptor from location 0xc0101347
```

the encoded policy byte string would be computed as:

```
000a 00000001 c0101347 0000000
```

Here 000a is the hypercall number of *HYPERVISOR_update_descriptor*, 00000001 is the 32-bit number that says that the call site is constrained but all the parameters should be unconstrained, and c0101347 is the call site. We computed a MAC over this byte string using the AES-CBC-OMAC message authentication code, which produces a 32-bit code [32].

The MAC is then added as an additional argument to the hypercall. The result is an authenticated hypercall, with two more arguments (the encoded policy string and the MAC) than the original hypercall.

Note that with this encoded policy byte string, we can also extend the policy to protect hypercall arguments. In particular, for each original argument of the hypercall, it can encode whether the argument is unconstrained or constrained to be a constant value. In our prototype, we do not implement this feature since among those hypercalls we experiment with there are none with constant arguments. However, it would be straightforward to incorporate this protection into our model.

4.2.2 Transformation of a Guest Kernel

The transformation process is to transform a normal guest OS kernel implementing regular hypercalls with a *blessed* kernel in which regular hypercalls are replaced by authenticated hypercalls.

It is ideal to have an automatic installer program that reads the kernel binary, uses static analysis to generate policies, and then rewrites the binary with the authenticated calls. Although static analysis and binary rewriting tools such as PLTO [49] are available for binary applications, such equivalent tools which can perform similar functions on the whole Linux kernel binary are not available to the public or are currently under development [55].

Since developing such tools is beyond the scope of our research, we took a manual approach to generate the policy for a guest OS, and to produce an executable guest OS kernel image that contains authenticated hypercalls. Specifically, this approach requires 3 steps to replace the regular *HYPERVISOR_xxx* with the authenticated one:

1. Use *objdump* to identify the call site of the *HYPERVISOR_xxx*.
2. Manually search for any constant argument; generate encoded policies by hand based on the gathered information; and compute the MAC based on the encoded policies.
3. Finally, we create a dummy *HYPERVISOR_xxx_dummy* hypercall which carries the original arguments and two additional arguments, the encoded policy and the MAC. The original *HYPERVISOR_xxx* invokes

the `HYPervisor.xxx_dummy` which in turn invokes the `_hypercallx()` assembly stub. Note that this `_hypercallx()` assembly stub also has two more arguments than the one invoked by the original hypercall. We then recompile the guest kernel to generate a *blessed* guest kernel. Because hypercall handlers in the Xen VMM look at the hypercall number (which is intact) to extract the right number of arguments, so there is no need to modify them.

With this approach, if the original hypercall has 3 arguments, the authenticated one must carry 5 arguments, which are the maximum number of arguments that a `_hypercallx()` assembly stub can handle due to the fact that hypercalls pass arguments through registers (there are 6 registers technically, but one is reserved to carry the hypercall number).

4.2.3 Hypercall Address Table

There are some weaknesses with authenticated hypercalls. Firstly, the process of constructing policies and transforming the guest OS kernels is complicated, especially when all the necessary tools are not available. Secondly, cryptographic MAC computation and verification can be computationally expensive. In addition, we cannot cover all the hypercalls because of the argument constraints and the coverage issues.

We explored a slightly weaker but less extensive approach to verify hypercalls using a Hypercall Access Table (HAT). HAT is embedded into the Xen VMM and it identifies the call sites of legitimate hypercalls; hypercalls from other locations are treated as intrusions. The modifications we propose are transparent to the guest OSes that do not wish to use them.

Our goal is to distinguish hypercalls invoked illegally by attack code from those invoked legally as part of a guest OS's normal execution. We begin with the simple observation that this objective can be achieved by examining the address (call site) of the hypercall instruction: if this is not any of the known locations from which the guest OS can possibly trap into the VMM, then it must be within attack code.

This approach is not as comprehensive as the MAC approach but it can offer a certain level of security with minimal cost. Without the cost of MAC computation, clearly performance can be improved. In addition, HAT can

have full coverage over all hypercalls.

4.2.4 Constructing Hypercall Address Table

We analyzed the guest OS binary image to identify addresses (call-sites) of hypercalls in the guest kernel. These addresses are then recorded into a database called Hypercall Address Table (HAT) stored in the VMM. The information in the HAT consists of three values for each hypercall instruction found in the guest OS: (1) the address of the instruction immediately following the hypercall instruction; (2) the hypercall number associated with it and (3) the guest domain ID. Notice that there is enough information in the HAT entries that the hypercall numbers passed into the VMM by the hypercall now become redundant: the VMM could use the address pushed on the stack by the hypercall instruction to obtain the corresponding hypercall number from the HAT. However, this feature can be useful for performance reason.

When a guest is started, the Xen VMM will check to see whether its hypercall addresses are present in a HAT, if they are not, its hypercalls are executed without any of the checks described in the next section. When a hypercall instruction occurs during the execution of a guest, the kernel checks that the address pushed on the stack by the hypercall appears in the HAT entries for that guest. A hypercall from an address found in the HAT is allowed to proceed; otherwise, a possible intrusion is signalled.

4.3 Hypercall Verification

4.3.1 Verifying Authenticated Hypercalls

Enforcement of a guest OS's authenticated policies is done at run time by a small module implemented in the Xen VMM. When a hypercall occurs, the verification module intercepts it and obtain its arguments which include the hypercall number, the arguments to the original unmodified call, the policy descriptor, and the MAC. Furthermore, the verification module also determines the call site based on the return address of the caller.

Using the information captured by interception, the verification module performs the following computation to validate that the actual hypercall complies with the specified policy. It first constructs an encoding of the pol-

icy by concatenating the hypercall number, the call site, the policy descriptor, and those argument values that are specified in the policy descriptor. Then it computes a MAC over this encoding using the same key used during the transformation process, and checks that the result matches the MAC passed in as an argument. If the two MACs match, the hypercall is passed on to the right hypercall handler and it is served normally; otherwise, the verification module refuses to pass on the hypercall to the hypercall handler and logs the violated hypercall.

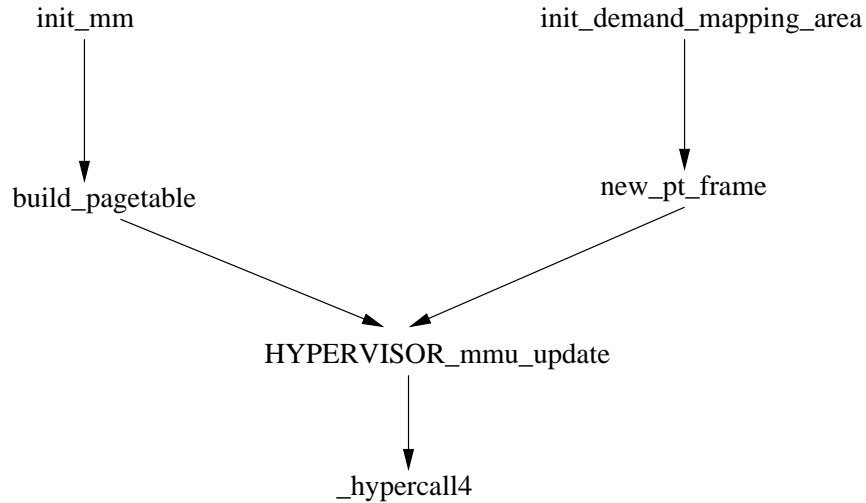
Hypercall verification is designed so that MAC matching fails if a guest OS has been compromised. Since the arguments to the authenticated hypercall are under the control of the compromised guest OS, the attacker might have tampered with any of them, including the policy descriptor and the MAC, or he might have even tried to construct a new authenticated hypercall somewhere in the guest memory. However, any change to the hypercall number, call site, policy descriptor, or values of the arguments constrained by the policy would result in a change to the encoded policy that is constructed by verification module. This in turn would change the MAC needed to pass the verification process. Our cryptographic assumption is that it is infeasible for the adversary to construct a matching MAC for its changes without access to the key, hence, any attempt by the guest OS to change the hypercall which violates the policy will fail.

4.3.2 Checking Hypercall Address Table

Similar to the MAC approach, hypercall verification for the HAT approach can start with intercepting the hypercall and calculating the return address to identify the hypercall call-site. Then the verification module will perform an HAT look-up to see if the address matches with one of the entries on the table corresponding to the guest ID and the hypercall number. If it is, the hypercall is passed on to be served by the hypercall service routine; otherwise, it is blocked and logged. The failed hypercall log can be used later for analysis purposes.

4.4 Hypercall Chain Verification

With the previous approaches, hypercall integrity and authenticity are protected to a certain degree. For instance, the attacker will fail if he tries to directly inject a malicious hypercall into a protected guest kernel or when he tries to tamper with some parameters under coverage of the protection



policy. However, the attacker is still able to launch an attack indirectly by invoking another function which in turn invokes functions that lead to hypercalls (in a manner similar to LIBC syscall attacks in OSs).

In our prototype, we aim to protect the hypercalls at lowest level in the MAC approach, we cannot protect against the case when the attacker tries to inject the malicious code from the higher levels. In the HAT approach, even though in theory we could build a table with extended entries to cover more or all call sites, but building that complete call-graph is a challenging task, especially without binary rewriting tools. In addition, in case Xen supports a large number of guests, this could result in a very big database created in the Xen VMM which is undesirable for the purpose of keeping the VMM codebase small.

4.5 Discussion

In terms of space, in the MAC approach, we cannot accommodate a hypercall with more than 5 arguments given the limitations of our current implementation approach. If we use the binary rewriting method, the size of the binary will increase by a constant number which is negligible.

In the HAT approach, the size of the table will increase when we have an increased number of guest OSes. This may be undesirable as the design

concept of Xen is to keep the VMM small. One approach is to store the database in a separate special guest and establish a trusted path to that guest.

4.5.1 Remaining Issues

We did not address the dynamic loadable module (LKM) issue in this thesis. Whether we modified the guest kernel source or binary, an attacker can inject code by LKM and a guest kernel has legitimate reasons to load a kernel module. We do not fully know that if any of possible loadable kernel modules contains hypercalls. In theory we can set up policies to prevent LKM at runtime or accommodate latest research in the field to deal separately with this issue.

Given the incompleteness of our implementation prototype, we did not go far enough to address the more complicated attacks such as mimicry or attacks that result in impossible path i.e., injected in one place and return to another. We believe that with appropriate tools and additional/complimentary techniques we could extend our models to cope with these attacks.

Chapter 5

Evaluation

5.1 Security Evaluation

5.1.1 Experimental Setup

Our experiments were run on the system of 1.7 GHz Pentium 4 processor with 512M RAM running Fedora Core 4. All Xen modifications necessary for this hypercall-based intrusion detection and prevention system were implemented in the *Xen-unstable* kernel. All guests run the modified Linux 2.6.17 kernel. The changes required to Xen were minimal, spanning only a handful of source files including the file containing hypercall assembly entry codes and the files containing the hypercall service routines; adding hypercall verification functionality. On the guest domain kernel, we modified the file containing hypercall stubs and the files containing the routines which invoke hypercalls.

5.1.2 Attack Experimental Issues

We have some difficulties to fully evaluate the efficacy of our hypercall protection measures. One common approach to evaluating a security measure would be subjecting it to currently known viruses/worms. In our case, however, this approach is not applicable. Firstly, known attack codes on Xen virtualization are not available. Even if they do exist, they may be specific to certain kinds of VMM vulnerabilities and do not represent the variety of vulnerabilities in reality. Secondly, this approach may ignore attacks that are possible in principle but which have not yet been identified. Finally, our implementation prototype is not robust and complete enough to be subjected to a real security scenario.

Instead, we focus the security evaluation on making sure that our protection measures are effective for what they are designed to be effective against. It means that our security measures should be able to detect and record injected hypercalls or hypercalls which are not in the protected guest

originally or hypercalls with an invalid number of arguments.

We used a loadable kernel module (LKM) on one of the guest OSes as a simple way to inject non-protected hypercall into the system. We assume that such an LKM could be successfully injected into a guest OS through common OS security breaches in the real world. This LKM would attempt to either try to execute a dummy hypercall which is a new hypercall and is deliberately left out when we compute MAC or HAT, or try to execute an existing hypercall with a different number of arguments. We can then verify that when the LKM tries to execute that new dummy hypercall which is not in our protection coverage, it got caught by both MAC and HAT protection. When the LKM tries to execute one existing hypercall with a different set of arguments it is caught by the MAC protection since the MAC for that call does not match.

This simple experiment is very specific and even quite contrived, but it serves the purpose of verifying the correctness of our prototype nevertheless. For a full security analysis, we need to do more experiments to cover all hypercalls.

5.2 Performance

There are two aspects to the cost of the underlying hypercall protection mechanisms: the incremental cost for an individual hypercall, and the impact on the overall performance of applications on guest OSes.

5.2.1 Micro Benchmark

We measure the performance impact of the implemented methods on individual hypercalls. To measure relatively accurate overhead requires to repeatedly execute a hypercall for a number of times. It would be ideal to have a complete coverage of all hypercalls. However, for some hypercalls it is hard to repeat their execution without changing the system state. As such, we only selected five hypercalls which are relatively more convenient to be measured than the others: `_fpu_task_switch`, `_memory_op`, `_mmuext_op`, `_update_va_mapping`, `_acm_op`.

We used a simple kernel module to make hypercalls repeatedly and we also measured overhead of these same hypercalls during the start-up and shut-down of guest domains. We used the `rttscl` system utility to measure

5.2. Performance

the system time to invoke each hypercall (directly corresponding to the hypercall) LIMIT times in a loop (LIMIT == 50). We then divided the total time by the LIMIT to get the mean execution time per invocation. Such value is collected from 10 runs, we discard the highest and lowest values, and average the remaining 8 values.

The results in Table 5.1 show that for the HAT approach, the overhead ranged from 8.07% to 58.94%. It ranged from 57.84% to 175.37% with the MAC approach. Overall the MAC approach incurs a lot more overhead which is expected. The overhead for each hypercall varies depending on the total execution time for each hypercall.

5.2.2 Macro Benchmark

We used a set of different applications to measure the macro-benchmark of the system including the Apache *ab* benchmarking tool, and *make* to compile the whole Linux 2.6.17 kernel. We also measured the time it takes to *tar*, *gzip* and *scp* on the Linux 2.6.17 kernel. We observed a small amount of increase in the overhead across the methods we used to protect hypercalls. All the overheads are smaller than the relative standard deviations.

We used the ApacheBench bench-marking suite for the Apache HTTP server as the benchmark for evaluating the performance impact of protected hypercalls on guest domains. Apache is ideal for this purpose owing to its popularity, and to the fact that it may exercise some of the memory management hypercalls. We ran the *ab* tool from ApacheBench with the parameters: `-n 10000 -C 1000 http://localhost/50KB.bin` to simulate 1000

Hypercall	Original	Xen/HAT		Xen/MAC	
	Cost (<i>ms</i>)	Cost (<i>ms</i>)	Over- head (%)	Cost Cost (<i>ms</i>)	Over- head (%)
_fpu_task_switch	7.0255	9.5083	35.34	17.6618	151.40
_memory_op	12.0005	12.969	8.07	18.9412	57.84
_mmuext_op	16.0294	19.3408	20.66	29.1182	81.65
_update_va_mapping	6.8814	9.4144	36.81	17.8745	159.75
_acm_op	3.436	5.4612	58.94	9.4616	157.37

Table 5.1: Hypercall Micro-benchmark

5.2. Performance

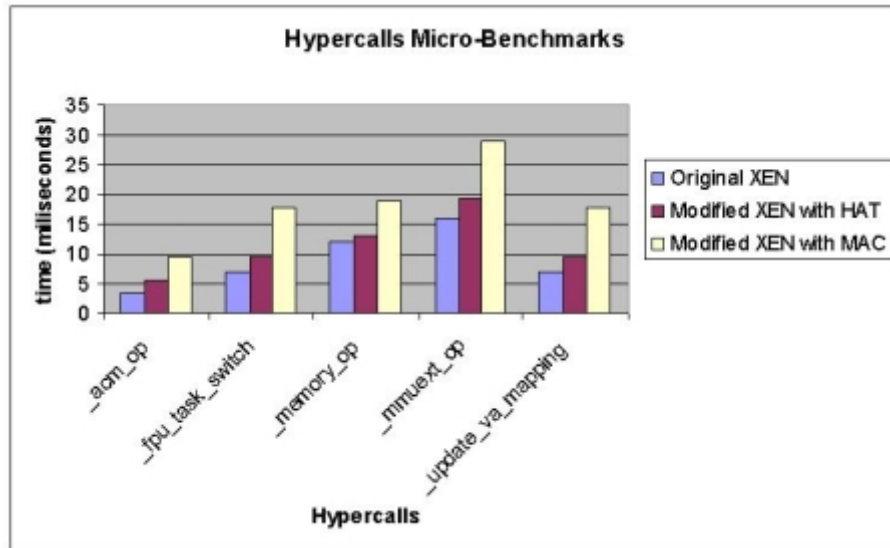


Figure 5.1: Micro Benchmarks

concurrent clients making a total of 10,000 requests for a 50KB file through the loop back network device (i.e., on the same host as the web server, to avoid network latency-induced perturbations to our results) on a guest. We collected and averaged the results for 5 runs of *ab* for each server configuration. The guest was otherwise unloaded.

Figure 5.2 plots the results of our macro-benchmark tests (also shown in Table 5.2). The Apache server suffers a small 5.13% decrease in request-handling capacity while running under HAT protection, as compared to running unprotected on original Xen. Respectively, it suffers a 7.68% decrease in request handling capacity with the MAC approach. The standard deviation indicates that these decreases are insignificant.

We also ran other applications to determine the impact of protected hypercalls on the performance of common Linux programs. Table 5.2 shows the execution time (averaged over 5 runs, and corresponding standard deviation) for the *kernel compiling*, *tar*, *gzip* and *scp* utilities, measured using the *time* command. All these tests involved the entire Linux 2.6.17 code repository: we ran *make* to compile the Linux 2.6.17 kernel, *tar* to create an archive of the kernel source code repository, and *scp* to upload the archive to a remote server (using public keys for automatic authentication). We also

5.3. Correctness

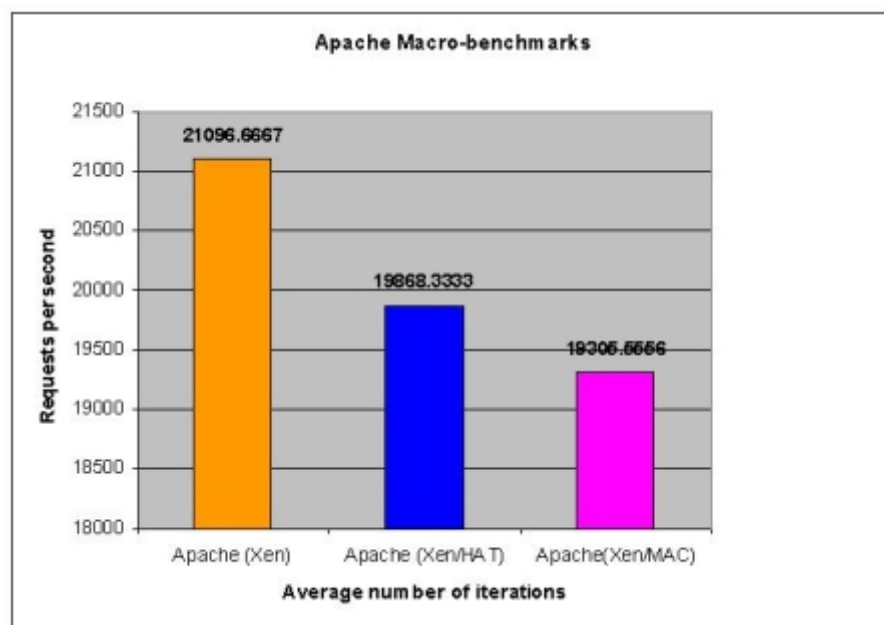


Figure 5.2: Apache Performance

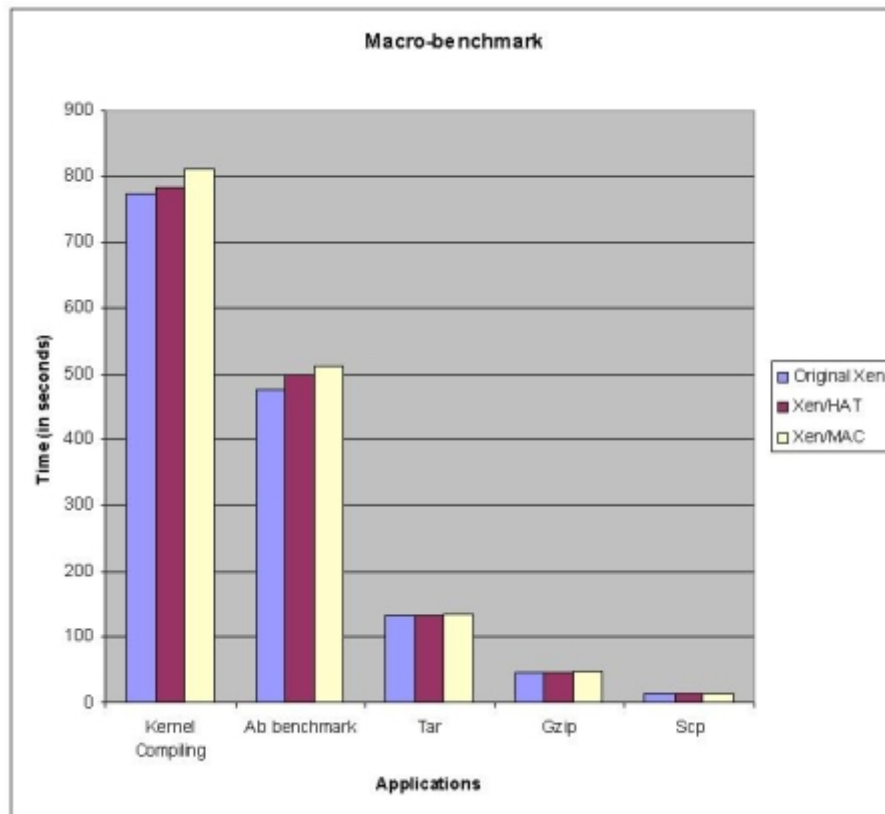
compressed this kernel archive using *gzip*. These results show that overhead added by protected hypercalls to these common programs is insignificant.

The MAC approach consistently has a slightly bigger overhead compared to the HAT approach. The reported overhead of Apache is somewhat higher than we expected. However, overall all overheads are consistently smaller in magnitude than the standard deviations.

5.3 Correctness

We have been running our Linux machine for daily use with the modified Xen with both MAC and HAT for extended periods of time without noticing abnormalities or significant performance degradation. All regular applications run normally. During our performance evaluation period, the system functions well under heavy loads. Although further stress tests may be needed, with current conditions we can come to the conclusion that our protection measures maintain system correctness.

5.3. Correctness



5.3. Correctness

Program	Original		Xen/HAT			Xen/MAC		
	Time (secs)	Std. Dev.	Time (secs)	Std. Dev.	Over- head (%)	Time (secs)	Std. Dev.	Over- head (%)
Kernel Compiling	773.351	11.67	782.224	8.61	1.15	810.394	6.73	4.79
Apache benchmark	474.467	15.91	498.815	10.78	5.13	510.883	11.26	7.68
Tar	131.184	1.57	132.282	1.36	0.84	134.217	3.1	2.31
Gzip	46.344	0.87	46.417	0.79	0.16	47.037	0.94	1.50
Scp	14.562	0.37	14.716	1.03	1.06	15.027	0.56	3.19

Table 5.2: Macro-benchmark.

Chapter 6

Conclusion and Future Work

While virtualization technology in general becomes increasingly popular and many research applications rely on it as a solution for various security problems, this thesis focuses strengthen Xen itself. We believe that making Xen more secure would bring true values to its popularity and usability. We have a high level security evaluation of Xen. We conclude that although the Xen VMM (or hypervisor in general) is much smaller than a conventional OS and has much narrower interface to low level hardware, it is still rather complex and certainly not free of vulnerabilities. We are also convinced that these vulnerabilities can be exploited through the hypercall interface and harden hypercalls would make Xen more secure in certain aspects.

Based on this premise and given the similarity in terms of functionality and characteristics of syscalls and hypercalls, we do a literature survey of syscall security including attacks and defense mechanisms, this survey gives us insights into how to protect hypercalls. Syscall monitoring is a popular technique to protect syscall and reactive syscall monitoring including detecting and repoding to attacks. Among detecting techniques, the anomaly detection approach appears to be more suitable for our case since this approach can provide capabilities to detect unknown attacks. We adopt two simple anomaly detection techniques using hypercall callsites as basic information combined with cryptographic and caching schemes to provide simple yet effective hypercall protection against hypercall injection attacks. We implement Authenticated Hypercalls (MAC) and Hypercall Access Table (HAT) in our simple proof-of-concept prototype. We would be able to do some simple evaluation and prove that protecting hypercalls can prevent certain kinds of intrusion via hypercalls. Despite the fact that hypercall protection measures can cause high overhead per each hypercall, their impact on the overall system performance is negligible.

We believe that improving security of virtualization technology is an important area to make this technology more mature. We believe that this research direction is interesting and it opens opportunities for us to learn

and to qualify some of the security mechanisms from the OS community. However, our prototype is incomplete and this thesis has been delayed over an extended period of time for some reasons. Xen itself, the research community around Xen and other security tools have been evolving so much over this period of time, we have missed some opportunities to incorporate those developments into this thesis.

There are many areas in this thesis that could be improved upon:

Use of Binary Rewriting and/or Static Analysis Tools: we chose to modify the guest OS kernel and Xen VMM directly to implement our prototype. There are some limitations with this approach. First of all, it is not always straightforward to modify the guest kernel in practice for reasons like the unavailability of kernel source code or legal constraints. In addition, in the MAC approach we cannot cover hypercalls having more than 3 arguments because we need additional space to store the policy string and MAC; if we want to protect hypercalls with more than 3 arguments, we need to change the current hypercall implementation to pass arguments via memory instead of registers which adds extra complexity. Finally, to have an extended HAT model to accommodate complete hypercall chains instead of only callsites, we need to utilize some static analysis tools because it is error-prone and very challenging to compute complete call chains manually.

At the time we carried out this project, we were not aware of any available binary rewriting tool which is free and capable of handling the Linux kernel binary. We believe that with the assistance of proper tools we can avoid the complication of kernel modification and can explore different ways to provide a better and more complete implementation prototype.

More Complete Implementation of The Prototype and Full Security Evaluation: we would like to improve the MAC implementation so that all hypercalls can be protected. As mentioned above, this would require fundamental changes to the current hypercall implementation or use of some binary rewriting tools. It would be interesting to extend protection coverage over the hypercalls at the lowest level. We could achieve this by implementing an extended HAT with each entry as a more comprehensive data structure (i.e., an inverted tree) to contain complete call chains lead to this hypercall. It would be technically challenging to generate such complete call graph, however given the small numbers of hypercall this approach

seems to be achievable with appropriate tools.

We would also want to experiment with the stack walk or hypercall chain verification approach as an alternative or complementary to the MAC/HAT approach. Technically, hypercall chain verification neither requires modification of the guest kernel nor performs any callsites computation in advance. However, it can also be used in conjunction with the extended HAT. Finally, it would be interesting to compare and contrast these different protection models in terms of performance and effectiveness.

Given the completeness of all the above protection measures, we also need to improve protection policies to detect more attacks which are not simply injected hypercalls, but more sophisticated ones such as mimicry attacks. Finally, we may also need to implement some security measures to protect our protection measures.

Bibliography

- [1] Cert security advisories. <http://www.cert.org/advisories/>.
- [2] Security focus.
- [3] Security: Ibm system z partitioning achieves highest certification. <http://www-03.ibm.com/systems/z/security/certification.html>.
- [4] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. Technical report, Santa Barbara, CA, USA, 1999.
- [5] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, 1998.
- [6] PaX Address Space Layout Randomization (ASLR). Pax address space layout randomization (aslr). <http://pax.grsecurity.net/docs/aslr.txt>.
- [7] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafepplus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, pages 45–56, 2004.
- [8] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time defense against stack smashing attacks. In *Proc. of the 2000 Usenix Annual Technical Conference*, Jun 2000.
- [9] Andrew Berman, Virgil Bourassa, and Erik Selberg. Tron: Process-specific file protection for the unix operating system. In *USENIX Winter*, pages 165–175, 1995.
- [10] S. Bhatkar, D.C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003.
- [11] Microsoft Security Bulletin. Microsoft security bulletin.

- [12] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, Dec 2002.
- [13] William W. Cohen. Fast effective rule induction. In Armand Prieditis and Stuart Russell, editors, *Proc. of the 12th International Conference on Machine Learning*, pages 115–123, Tahoe City, CA, July 9–12, 1995. Morgan Kaufmann.
- [14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuardTM: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, Aug 2003.
- [15] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of the 7th Usenix Security Symposium*, pages 63–78, Jan 1998.
- [16] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil D. Gligor. Subdomain: Parsimonious server security. In *LISA*, pages 355–368, 2000.
- [17] Brendan Cully. The virtual monkey monitor. Technical report, University of British Columbia, June 2006.
- [18] Vikram Adve Dinakar Dhurjati, Sumant Kowshik and Chris Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proc. Languages Compilers and Tools for Embedded Systems 2003*, San Diego, CA, June 2003.
- [19] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.
- [20] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.
- [21] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2003.

- [22] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proc. of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.
- [23] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proc. of the 10th Usenix Security Symposium*, pages 55–66, 2001.
- [24] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th Symposium on Operating System Principles (SOSP 2003)*, October 2003.
- [25] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.
- [26] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *In Proceedings of the 14th National Computer Security Conference*, October 1991.
- [27] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, Berkeley, CA, USA, 2002. USENIX Association.
- [28] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications — confining the wily hacker, 1996.
- [29] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [30] K. Ilgun. USTAT: A Real-time Intrusion Detection System for UNIX. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA, May 1993.
- [31] Koral Ilgun, R. A. Kemmerer, and Phillip A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Trans. Softw. Eng.*, 21(3):181–199, 1995.

- [32] T. Iwata and K. Kurosawa. Omac: One-key cbc mac, 2002.
- [33] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *In Proc. of The USENIX Security Symposium*, 2002.
- [34] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.
- [35] Gaurav S. Kc and Angelos D. Keromytis. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *Proc. of the 21st Annual Computer Security Applications Conference*, December 2005.
- [36] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM Conference on Computer and Communications Security*, pages 272–280, 2003.
- [37] Calvin Ko. *Execution Monitoring of Security-critical Programs in a Distributed System: A Specification-based Approach*. PhD thesis, Department of Computer Science, University of California at Davis, USA, 1996.
- [38] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, volume xiii, pages 134–144. IEEE, IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.
- [39] Sandeep Kumar and Eugene H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the 17th National Computer Security Conference*, pages 11–21, 1994.
- [40] Wenke Lee, Salvatore J. Stolfo, and Philip K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. AAAI Press, 1997.
- [41] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

- [42] K. Lhee and S. Chapin. Type-assisted dynamic buffer overflow detection, 2002.
- [43] C. M. Linn, M. Rajagopalan, C. Collberg S. Baker, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *Proc. of the 14th USENIX Security Symposium*, October 2005.
- [44] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramon Caceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pages 23–32, Washington, DC, USA, 2006. IEEE Computer Society.
- [45] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.
- [46] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), Dec 2001.
- [47] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. Technical report, Google Inc., 2007.
- [48] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *USENIX 2001 Technical Conference Proceedings: FreeNIX Track*, Berkeley, CA, June 2001. USENIX Association.
- [49] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. Authenticated system calls. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN 2005)*, pages 358–367, Yokohama, Japan, June 2005.
- [50] Gerardo Richarte. Four different tricks to bypass stackshield and stack-guard protection. *Core Security Technologies*, Jun 2002.
- [51] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. May 2005.
- [52] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.
- [53] E. C. Sezer P. Gauriar S. Chen, J. Xu and R. K. Iyer. Non-control-data attacks are realistic threats. In *In Proc. of The USENIX Security Symposium*, Baltimore, MD, USA, August 2005.

- [54] A. Ghosh A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *8th USENIX Security Symposium*, pages 141–151, 1999.
- [55] Benjamin Schwarz, Saumya Debray, , and Gregory Andrews. Plto: A link-time optimizer for the intel ia-32 architecture. In *Proc. of the Workshop on Binary Translation (WBT-2001)*, September 2001.
- [56] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society.
- [57] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.
- [58] United States Computer Emergency Readiness Team. Technical cyber security.
- [59] Timothy K. Tsai and Navjot Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 541, Washington, DC, USA, 2002. IEEE Computer Society.
- [60] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.
- [61] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.
- [62] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, Washington, DC, USA, 2002. ACM Press.

- [63] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, Oakland, CA, 1999.
- [64] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *RAID '00: Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 110–129, London, UK, 2000. Springer-Verlag.
- [65] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems*, pages 68–84, Karlstad, Sweden, November 2002.
- [66] Rafel Wojtczuk. Defeating solar designer non-executable stack patch. January 1998.
- [67] William D. Young and John McHugh. Coding for a believable specification to implementation mapping. In *In Proc. of The Symposium on Security and Privacy*, pages 140–148, Oakland, CA, USA, April 1987. IEEE Computer Society.
- [68] Oiwa Yutaka, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ansi-c compiler: An approach to making c programs secure: Progress report. In *ISSS*, pages 133–153, 2002.