ABSTRACT

WANG, ZHI. Securing Virtualization: Techniques and Applications. (Under the direction of Xuxian Jiang.)

Virtualization is being widely adopted in today’s computing systems. Its unique advantages in isolating and introspecting commodity OSes as virtual machines have enabled a wide spectrum of applications. However, a common, fundamental assumption of all these virtualization-based systems is the presence of a trustworthy hypervisor. Unfortunately, recent successful attacks against hypervisors, in addition to the bloated trusted computing base and highly complex internal logic of commodity (type-I and type-II) hypervisors, seriously question the validity of this assumption.

In this dissertation, we first present two systems to mitigate the threats posed by vulnerable type-I and type-II hypervisors, respectively: HyperSafe is a lightweight approach that enables control flow integrity of type-I hypervisors for their self-protection. It has two key techniques: non-bypassable memory lockdown reliably protects the hypervisor’s code integrity even in the presence of exploitable memory corruption bugs, and restricted pointer indexing enforces control flow integrity by converting control data into restricted pointer indexes; HyperLock is a systematic approach to strictly isolate vulnerable type-II hypervisors from compromising the host OSes. It also has two key techniques: hypervisor isolation runtime securely isolates a hypervisor in its own dedicated address space and restricts its instruction set for safe execution, and hypervisor shadowing efficiently creates an individual shadow hypervisor for each guest so that a compromised hypervisor can affect only the paired guest, not others. We have built a prototype for both systems based respectively on two open-source type-I hypervisors (i.e., BitVisor and Xen) and one type-II hypervisor (i.e., KVM). The security experiments and performance measurements of these prototypes demonstrated the practicality and effectiveness of our approaches.

The above two systems lay a solid foundation for secure virtualization-based systems. A wide range of virtualization-based security mechanisms can benefit from them. HookSafe, the third system presented in this dissertation, is one such system designed to mitigate serious threats from kernel rootkits. Specifically, many kernel rootkits hide their presence and activities by subverting kernel hooks (function pointers). A critical step toward eliminating kernel rootkits is to protect such hooks from being hijacked. This remains as a challenge due to the so-called protection granularity gap: kernel hook protection requires byte-level granularity but commodity hardware only provides page-level protection. HookSafe leverages the virtualization technology to address this problem. A key observation behind our approach is that a kernel hook, once initialized, may be frequently read but rarely changed. As such, HookSafe relocates those kernel hooks to dedicated memory pages and regulates accesses to them efficiently with hardware-based page-level protection. Our experiments with a prototype of HookSafe demonstrate that HookSafe can enable large-scale hook protection with a small overhead.
© Copyright 2012 by Zhi Wang

All Rights Reserved
DEDICATION

To my wife, Liang, and our wonderful daughter, Mia.
Zhi Wang is originally from Jiangsu, China. He received his Bachelor of Science and Master of Science degrees in Computer Science from the Xi’an Jiaotong University in China in 1999 and 2002, respectively. After receiving his master’s degree, he spent five years working as a software engineer in the industry. Particularly, during his three years (2004-2007) at Intel, he built the TCP/IPv4 network stack and the USB stack for Tiano, Intel’s UEFI based firmware, which is shipped in most recent PCs and Macs. Both stacks have been open-sourced by Intel as a part of the TianoCore project. Since 2007, he spent another five years in USA to pursue his Doctor of Philosophy degree in Computer Science under the direction of Dr. Xuxian Jiang. His research interests lie primarily in systems security such as operating system security, virtualization security (the focus of this dissertation), and mobile security. He will graduate with a Ph.D. degree in Computer Science from the North Carolina State University in August 2012, and join the Florida State University as an assistant professor in Fall 2012.
ACKNOWLEDGEMENTS

Pursuing a PhD degree is a daunting task that requires not only determination and persistence from the person pursuing the degree, but also a synergy from people related to it. At all stages of my graduate study, I have received various invaluable help from a number of people, whom I undoubtedly owe gratitude to.

First, I would like to thank my adviser, Prof. Xuxian Jiang, for his numerous help during my graduate study. Prof. Jiang has been instrumental, supportive, and conducive in guiding and shaping my research. His enthusiasm in research, standard for excellence, and eye for details will always be a source of motivation for me. I am also thankful to Prof. Peng Ning, Prof. Douglas Reeves, and Prof. Huiyang Zhou for their time and efforts serving in my PhD thesis committee and providing valuable advice that helped to improve this dissertation.

Second, I am grateful to all my coauthors for their helpful discussions and collaborations, to name but a few: Weidong Cui from Microsoft Research, Peng Ning from NC State, Dongyan Xu from Purdue, Xinyuan Wang from the George Mason University, and my colleagues in the Cyber Defense Lab at NC State, Ahmed Moneeb Azab, Sina Bahram, Tyler Bletsch, Mike Grace, Jinku Li, Deepa Srinivasan, Minh Q. Tran, Chiachih Wu, Wu Zhou, and Yajin Zhou.

In addition to academics, I received help in many other perspectives as well: Xuxian Jiang, Peng Ning, Douglas Reeves, and William Enck from NC State, Dongyan Xu from Purdue, Úlfar Erlingsson from Google, Dimitrios Pendarakis, Arvind Seshadri, and Josyula R Rao from IBM Research, Gary Tyson and Xin Yuan from Florida State, Ernie Salle from VMware, Weidong Cui, Michael Fortin, and Deepa Srinivasan from Microsoft, provided invaluable help during the stressful time of my job search by acting as a referee and/or giving constructive feedbacks and advice; Carol Allen, Margery Page, Tom Ryan, Douglas Reeves, David Thuente, and other staff members from the Department of Computer Science and the Office of International Services also deserve my gratitude. Because of them, my graduate study has been smooth; I am also grateful to organizations that generously funded my graduate research. The work in this dissertation was supported in part by US National Science Foundation (NSF) and the US Army Research Office (ARO) under several grants. As usual, any opinions, findings, and conclusions or recommendations expressed in this dissertation do not necessarily reflect the views of the NSF and the ARO.

Last but certainly not least, I deeply appreciate the everlasting support and encouragement from my family, particularly my wife, Liang Lu, and our daughter, Mia. Without their continuous support, none of this would have been possible, not to mention enjoyable. Finally, I can never overemphasize the importance of my parents in shaping who I am and giving me the gift of education.
TABLE OF CONTENTS

List of Tables .............................................................................. vii

List of Figures ............................................................................. viii

Chapter 1 Introduction ................................................................. 1
  1.1 Problem Overview ................................................................. 1
  1.2 Our Approach ..................................................................... 3
  1.3 Dissertation Contributions .................................................. 5
  1.4 Terminology ...................................................................... 5
  1.5 Dissertation Organization .................................................... 6

Chapter 2 Related Work ............................................................... 7
  2.1 Hypervisor Integrity ............................................................... 7
  2.2 Kernel and Application Integrity ........................................... 8
  2.3 Kernel Rootkit Defense ......................................................... 10

Chapter 3 Providing Control Flow Integrity for Type-I Hypervisors with HyperSafe . . . 12
  3.1 Introduction ...................................................................... 12
  3.2 Design .............................................................................. 15
    3.2.1 Goals and Assumptions ................................................ 15
    3.2.2 Key Technique I: Non-Bypassable Memory Lockdown .............. 16
    3.2.3 Key Technique II: Restricted Pointer Indexing ....................... 19
  3.3 Implementation .................................................................. 23
    3.3.1 Non-Bypassable Memory Lockdown ................................... 23
    3.3.2 Restricted Pointer Indexing ............................................. 24
  3.4 Evaluation ......................................................................... 28
    3.4.1 Security Analysis .......................................................... 28
    3.4.2 Synthetic Experiments .................................................... 29
    3.4.3 Performance Evaluation .................................................. 31
  3.5 Discussion .......................................................................... 33
  3.6 Summary ........................................................................... 35

Chapter 4 Isolating Type-II Hypervisors with HyperLock ................................................. 36
  4.1 Introduction ...................................................................... 36
  4.2 Design .............................................................................. 38
    4.2.1 Hypervisor Isolation Runtime ......................................... 40
    4.2.2 HyperLock Controller .................................................... 42
  4.3 Implementation .................................................................. 44
    4.3.1 Memory Access Control ................................................ 44
    4.3.2 Instruction Access Control ............................................. 45
    4.3.3 Others ....................................................................... 48
  4.4 Evaluation ......................................................................... 49
    4.4.1 Security Analysis .......................................................... 49
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.2 Performance Evaluation</td>
<td>51</td>
</tr>
<tr>
<td>4.5 Discussion</td>
<td>53</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>54</td>
</tr>
<tr>
<td><strong>Chapter 5  Countering Kernel Rootkits with HookSafe</strong></td>
<td>55</td>
</tr>
<tr>
<td>5.1 Introduction</td>
<td>55</td>
</tr>
<tr>
<td>5.2 Problem Overview</td>
<td>57</td>
</tr>
<tr>
<td>5.3 Design</td>
<td>58</td>
</tr>
<tr>
<td>5.3.1 Overview</td>
<td>58</td>
</tr>
<tr>
<td>5.3.2 Offline Hook Profiling</td>
<td>59</td>
</tr>
<tr>
<td>5.3.3 Online Hook Protection</td>
<td>60</td>
</tr>
<tr>
<td>5.3.4 Hardware Register Protection</td>
<td>62</td>
</tr>
<tr>
<td>5.4 Implementation</td>
<td>63</td>
</tr>
<tr>
<td>5.4.1 Offline Hook Profiler</td>
<td>63</td>
</tr>
<tr>
<td>5.4.2 Hook Indirection</td>
<td>64</td>
</tr>
<tr>
<td>5.4.3 Memory Protection</td>
<td>67</td>
</tr>
<tr>
<td>5.4.4 System Call Indirection Optimization</td>
<td>68</td>
</tr>
<tr>
<td>5.5 Evaluation</td>
<td>69</td>
</tr>
<tr>
<td>5.5.1 Effectiveness Against Kernel Rootkits</td>
<td>69</td>
</tr>
<tr>
<td>5.5.2 Performance</td>
<td>71</td>
</tr>
<tr>
<td>5.6 Discussion</td>
<td>73</td>
</tr>
<tr>
<td>5.7 Summary</td>
<td>74</td>
</tr>
<tr>
<td><strong>Chapter 6  Conclusion and Future Work</strong></td>
<td>75</td>
</tr>
<tr>
<td><strong>References</strong></td>
<td>77</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.1</td>
<td>Software packages for the evaluation of HyperSafe</td>
</tr>
<tr>
<td>3.2</td>
<td>LMbench results for HyperSafe (in $\mu$s – smaller is better)</td>
</tr>
<tr>
<td>4.1</td>
<td>The breakdown of HyperLock’s runtime service implementation. read MSR and</td>
</tr>
<tr>
<td></td>
<td>write MSR are implemented together. enter_guest includes 53 lines of in-</td>
</tr>
<tr>
<td></td>
<td>line assembly code and 25 lines of C code.</td>
</tr>
<tr>
<td>4.2</td>
<td>Software packages for the evaluation of HyperLock</td>
</tr>
<tr>
<td>5.1</td>
<td>Effectiveness of HookSafe in preventing 9 real world kernel rootkits: Hiding</td>
</tr>
<tr>
<td></td>
<td>fails indicates that rootkit failed to hijack the control flow because Hook-</td>
</tr>
<tr>
<td></td>
<td>Safe has redirected the hook to its shadow; Installation fails indicates</td>
</tr>
<tr>
<td></td>
<td>that the rootkit hooking behavior causes the memory protection violation,</td>
</tr>
<tr>
<td></td>
<td>hence failing the installation. Additionally, depending on how the rootkit</td>
</tr>
<tr>
<td></td>
<td>searches for the system call table, it may locate either the original system</td>
</tr>
<tr>
<td></td>
<td>call table (marked with †) or the shadow system call table (marked with ‡).</td>
</tr>
<tr>
<td>5.2</td>
<td>Software packages for the evaluation of HookSafe</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1 Three key components of the dissertation ........................................... 2
Figure 3.1 A break-down of hypervisor integrity guarantees and the corresponding key
   techniques in HyperSafe  ................................................................. 15
Figure 3.2 Traditional page table updates vs. new page table updates in HyperSafe (note
   the WP bit is ON by default) .............................................................. 18
Figure 3.3 Traditional indirect call vs. new indirect call in HyperSafe (note Ri is the return
   address of the indirect call) ............................................................... 19
Figure 3.4 Destination equivalence effect on ret instructions (a dashed line represents an
   indirect call while a solid line stands for a direct call) .......................... 22
Figure 3.5 The instrumented call to pmap_read and the corresponding ret (note the instru-
   mentation is done in x86_64) .............................................................. 25
Figure 3.6 Normalized performance of application benchmarks with the original BitVisor
   as the baseline  .................................................................................... 32
Figure 4.1 Traditional KVM execution vs. “HyperLocked” KVM execution ................... 39
Figure 4.2 Trampoline code layout: each code fragment starts with a one-byte INT3 and
   ends with a short JMP, which skips over the next INT3 of the following code
   fragment. The starting address of the trampoline code is loaded into an IDT
   entry as the interrupt handler ............................................................. 42
Figure 4.3 A VMWRITE macro-instruction that writes 0xc00195d7 into the GUEST_EIP
   VMCS field. ....................................................................................... 46
Figure 4.4 Normalized overhead of HyperLock ...................................................... 52
Figure 5.1 Distribution of 5, 881 kernel hooks in a running Ubuntu system ............... 57
Figure 5.2 The HookSafe architecture ............................................................. 59
Figure 5.3 The architecture of online hook protection. .......................................... 60
Figure 5.4 An example access profile related to ext3_dir_operations -> readdir kernel hook
   ........................................................................................................ 63
Figure 5.5 The implementation of hook indirection ............................................. 64
Figure 5.6 An example HAP instruction and the related C code ............................ 65
Figure 5.7 HookSafe foils the adore-ng rootkit. ................................................... 70
Figure 5.8 Runtime overhead of HookSafe ......................................................... 72
Chapter 1

Introduction

1.1 Problem Overview

Virtualization is being widely used in today’s computing systems including various clouds. For example, Amazon Web Services (AWS) [10], one of the largest public clouds, is based on the open-source Xen [14] hypervisor. In virtualization, commodity operating systems (OSes) are encapsulated in virtual machines, and executed under the full control of the hypervisor.

The hypervisor’s unique strength in isolating and introspecting commodity OSes as virtual machines has promoted a wave of research [37, 45, 53, 55, 64, 72, 73, 79, 83, 103] to address challenging security problems such as kernel rootkit detection and defense. One fundamental assumption shared by all these systems is the presence of a trustworthy hypervisor. A typical supporting argument is that the hypervisor has a code base that is much smaller than conventional OSes and thus can be better scrutinized to remove software bugs. However, as a matter of fact, contemporary type-I hypervisors (Section 1.4) such as Xen [14] and VMware ESXi [98] all have a large and complex code base. For instance, Xen 3.4.1 contains about 230K source lines of code (SLOC). By leveraging an existing host OS, a type-II hypervisor (Section 1.4) itself indeed has a smaller code base (e.g., a recent version of KVM [48] has about 36 KSLOC), but the huge host kernel now becomes a part of the trusted computing base (TCB).

In addition, virtualizing a platform has never been a trivial task despite the recent advances in the hardware support for virtualization. For example, one of the most vulnerable components in KVM is the instruction emulation module [63] that has to simulate all the convoluted effects and peculiarities of the x86 instruction set. A recent study of the National Vulnerability Database [63] indicates that in the last three years (until July 2011), there were 41 security vulnerabilities reported for Xen, 43 for VMware ESX, and 24 for KVM. Some of these vulnerabilities can be exploited to execute arbitrary code in the hypervisor. Moreover, successful VM escape attacks against Xen [110], KVM [29], and all VMware hypervisors [50], as well as the emerging hypervisor rootkits [17, 46], greatly exacerbate the situation. In light of these serious threats, there is a pressing need to investigate effective ways to secure the hypervisors [74].
The architecture of virtualization that gives the hypervisor dominative power in controlling and inspecting virtual machines also makes it challenging to secure. As the lowest and most-privileged layer in the system, the hypervisor cannot be secured with the traditional one-layer-below approach (e.g., securing a hypervisor with nested virtualization [15, 38]). Otherwise, we would face the same question of how to secure this new lowest software component. Yet, a failure to secure the hypervisor will have grave consequences on the whole system. To address this challenge, researchers have approached the problem from different perspectives: seL4 [49] formally verifies that a relatively small micro-kernel (about 8.7K SLOC) strictly conforms to its specification, including the absence of many software vulnerabilities (e.g., buffer overflows). However, it is still not scalable enough to verify commodity hypervisors with a much larger code base and the complex x86 CPUs and chipsets [76]. Unlike seL4, we focus on securing existing commodity hypervisors; HyperSentry [13] and HyperCheck [100] are two systems that monitor the integrity of the hypervisor from a separate environment. They can detect a violation of the hypervisor integrity only after it has happened. We instead try to protect the integrity of the system at the first place; There are also research efforts [26, 62, 90] in further reducing the TCB of a hypervisor, often with a micro-kernel based approach. As such, they can reduce the number of vulnerabilities in a hypervisor, but do not provide stronger security guarantees.

In this dissertation, we envision a systematic and integrated approach to provide a secure virtualization system that can be readily deployed in today’s hardware and software architecture. To realize this vision, we need to answer a series of critical research questions: what security mechanisms can be applied to the hypervisors to provide strong protection while still preserving performance? How can these mechanisms be integrated into the existing hardware and software architecture in a non-intrusive way, for example, without requiring changes to the hardware or the introduction of a new dominant layer below the hypervisor? What are the security guarantees provided by these mechanisms and how reliable are they under the assumption of exploitable vulnerabilities in the hypervisor? What is the performance overhead introduced by these mechanisms? After creating a trustworthy hypervisor, how can we best leverage the power of virtualization to address challenging security problems?
To address these challenges, we propose a systematic approach in this dissertation which consists of three key components as shown in Figure 1.1: HyperSafe (Chapter 3) and HyperLock (Chapter 4) are two systems to harden the security for hypervisors. They compliment each other to accommodate the architectural differences between type-I and type-II hypervisors (Section 1.4). Specifically, HyperSafe enables the self-protection for a type-I hypervisor by providing its control flow integrity, while HyperLock securely isolates a vulnerable or compromised type-II hypervisor in a dedicated address space to prevent it from affecting the host or other guests. Under the protection of these two systems, a wide range of virtualization-based security services can be reliably applied, for example, to measure or protect the guest integrity. The third system in this dissertation called HookSafe leverages the isolation and control provided by virtualization to defeat malicious kernel rootkits.

1.2 Our Approach

To protect the hypervisor integrity, we treat type-I and type-II hypervisors differently due to their architectural differences: a type-I hypervisor is self-contained and manages the hardware resources by itself. It is the sole privileged component running beneath all other software components on a physical machine. In Chapter 3, we propose HyperSafe, a system that can enable the self-protection for a type-I hypervisor by enforcing its control flow integrity [5]; A type-II hypervisor instead relies on a host OS for many of its functionalities, such as scheduling, memory management, and device management. It usually runs as a loadable kernel module inside the host OS. In Chapter 4, we propose HyperLock, a system that can securely isolate a type-II hypervisor inside its own address space to prevent it from compromising the host OS and other guests on the same physical machine.

More specifically, HyperSafe enables the self-protection for a type-I hypervisor by reliably enforcing the hypervisor control flow integrity (CFI). In other words, it guarantees that runtime execution of the hypervisor will always follow a path in the hypervisor’s control flow graph (CFG) [7]. To enforce CFI, it is necessary to protect the hypervisor code integrity to prevent any modification of its code and read-only data, and control data integrity to constrain its runtime execution. HyperSafe correspondingly has two key techniques: non-bypassable memory lockdown and restricted pointer indexing. By regulating any changes to the hypervisor page tables, the first technique locks down the memory protection (in the page table) for the hypervisor code and static data and prevents them from being modified even in the presence of exploitable memory corruption bugs. The second technique curbs the hypervisor runtime execution by converting its control data (e.g., function pointers and return addresses) into pointer indexes. Specifically, we collect the valid targets (computed from the CFG) of indirect control transfer instructions (e.g., indirect call and return instructions) into read-only target tables protected by the first technique, and use the index to select a target out of this table at runtime. As such, indirect control transfer can only go to a valid target in the CFG, and the hypervisor CFI is guaranteed.
Compared to a type-I hypervisor, a type-II hypervisor runs as an extension to the host OS. Protection schemes for a type-II hypervisor accordingly should be applied to its host OS as well due to the lack of isolation between them. Particularly, the techniques in HyperSafe cannot be directly applied to a type-II hypervisor. This is because the host OS frequently updates page tables for its applications, thus the page table access control in HyperSafe likely will introduce unacceptable performance overhead for a type-II hypervisor. In Chapter 4, we propose HyperLock, a systematic approach to strictly isolate a vulnerable type-II hypervisor from the host OS. HyperLock also has two key techniques: hypervisor isolation runtime creates a dedicated address space for the hypervisor to isolate it from the host OS, and further restricts its instruction set for safe execution; hypervisor shadowing leverages the virtual memory sharing to efficiently create multiple instances of the hypervisor, and pairs each guest with its own hypervisor instance. Under the isolation of HyperLock, a compromised hypervisor can affect only the paired guest, but not the host OS or other guests.

We have built a prototype for both HyperSafe and HyperLock based respectively on the type-I Xen [14] and BitVisor [86] hypervisors, and the type-II KVM [48] hypervisor. Our experiments show that these two systems can effectively secure a virtualization system with a small performance overhead (less than 5% for both systems).

HyperSafe and HyperLock together lay a solid foundation for secure virtualization systems. A wide spectrum of security services can be reliably deployed based on them. In Chapter 5, we propose one such system called HookSafe to defeat nefarious kernel rootkits. A kernel rootkit is a piece of kernel-level malware that subverts the OS kernel to hide its presence and maintain privileged access. Kernel rootkits are difficult to detect or remove because any defense mechanisms running in the same kernel are vulnerable to attacks by kernel rootkits. HookSafe addresses this problem by leveraging virtualization to protect the guest kernel hooks (function pointers), which are frequently subverted by kernel rootkits to hijack kernel execution. Kernel hook protection remains as a challenge because of the so-called protection-granularity gap: kernel hooks often scatter all over the kernel address space and coexist with other non-control data. Thus, kernel hook protection calls for byte-granularity memory protection to selectively write-protect these function pointers. Unfortunately, commodity hardware only supports page-level memory protection. To address this problem, we observe that kernel hooks rarely change their values, but can be read frequently. Therefore, HookSafe relocates kernel hooks to a centralized page-aligned memory space, and regulates modifications to them with the hardware-based page-level protection. By doing so, infrequent write-accesses to kernel hooks will be trapped to the hypervisor and validated before taking place, while (frequent) read-accesses are directly handled in the guest kernel without the involvement of the hypervisor. Our experiments with a prototype of HookSafe show that HookSafe is effective in defeating kernel rootkits, and can enable large scale hook protection with a small performance overhead.
1.3 Dissertation Contributions

The contributions of this dissertation are threefold: type-I hypervisor integrity, type-II hypervisor integrity, and kernel rootkit defense.

- **Type-I hypervisor integrity** We have designed, implemented, and evaluated HyperSafe, a system that uniquely enables the self-protection for type-I hypervisors by enforcing their control flow integrity. Specifically, we have proposed two techniques, non-bypassable memory lockdown and restricted pointer indexing, in HyperSafe to guarantee the hypervisor code and control data integrity on the commodity x86 platform. HyperSafe is the first system that can protect the integrity for commodity type-I hypervisors.

- **Type-II hypervisor integrity** We have designed, implemented, and evaluated HyperLock, a system that can securely isolate a vulnerable or compromised type-II hypervisor. Likewise, we have proposed two techniques in HyperLock, hypervisor isolation runtime and hypervisor shadowing. The first technique constrains the execution of a type-II hypervisor in a dedicated and isolated address space, while the second technique creates multiple instances of a hypervisor and pairs each guest with its own hypervisor instance. Under HyperLock, a compromised hypervisor will be prevented from taking over the host or other guests. HyperLock is the first system that can isolate type-II hypervisors.

- **Kernel rootkit defense** We have designed, implemented, and evaluated HookSafe, a virtualization-based system that can defeat nefarious kernel rootkits by preventing them from hijacking kernel hooks. In HookSafe, we have proposed techniques to relocate widely-scattered hooks in the guest kernel to a central location and protect them efficiently with the hardware page-level memory protection, thus addressing the protection-granularity gap challenge. HookSafe is a systematic and lightweight approach to protect the guest kernel from threats posed by kernel rootkits.

1.4 Terminology

This section establishes the terminology that is used throughout this dissertation.

- **Virtual Machine** A virtual machine is a software abstraction that resembles an actual (e.g., x86) or imaginary (e.g., Java virtual machine) machine’s hardware. A virtual machine can run all the software written for the machine that it virtualizes. For example, KVM can be used to run Microsoft Windows in a virtual machine on the Linux platform.

There are many variants of virtual machines such as the x86 virtual machine that emulates an x86 platform, and the Java or CLR virtual machines for high level programming languages. In this dissertation, we focus on the hardware-assisted virtual machine (HVM) for the x86 architecture [42]
(the discussion of the following terms is specific to the HVM). A virtual machine that has been loaded to the physical machine for execution is also called as a guest.

- **Virtualization** Virtualization is the technology to create, execute, and manage virtual machines. The software that implements virtualization is usually called virtual machine monitor or hypervisor. There are two types of hypervisor: type-I hypervisor and type-II hypervisor.

- **Type-I Hypervisor** A type-I hypervisor (also called native hypervisor or bare-metal hypervisor) runs directly on top of the machine’s hardware. It controls and manages the hardware resources by itself and is the sole privileged component in a virtualization system. Xen [14], VMware ESX [98], and Microsoft Hyper-V [3] are examples of type-I hypervisors.

- **Type-II Hypervisor** A type-II hypervisor (also called hosted hypervisor) runs as an extension (e.g., a loadable kernel module) in a commodity host OS. It relies on the host OS for many of its functionalities such as memory management, device management, and scheduling. By leveraging mature functionalities in the host OS, a type-II hypervisor is relatively easy to manage and immediately supports most, if not all, of the hardware devices supported by the host OS. However, a type-II hypervisor usually has much larger TCB than a type-I hypervisor because the host OS is a part of the TCB. KVM [48] and VMware workstation [98] are examples of type-II hypervisors.

- **Control Flow Integrity** Control flow integrity (CFI) [5] is a security property under which runtime execution of a program must follow a path in its statically decidable control flow graph (CFG) [7]. CFI was introduced by Abadi et al. [5] for user space applications. CFI requires the code integrity as a precondition to prevent the introduction of new code into the program.

- **Sandbox** A sandbox is a security mechanism to execute untrusted binary in a constrained environment to prevent it from compromising trusted parts of the system. Sandbox is used widely to isolate untrusted or malicious program extensions. For example, Native Client [113] was designed to isolate untrusted browser plugins written in the native x86 instructions.

### 1.5 Dissertation Organization

The rest of this dissertation is organized as follows. First, we present closely related work in hypervisor integrity and its applications in Chapter 2. Second, we give details about the design, implementation, and evaluation of HyperSafe in Chapter 3, HyperLock in Chapter 4, and HookSafe in Chapter 5, respectively. Finally, we conclude this dissertation and propose a few directions for the future work in Chapter 6.
Chapter 2

Related Work

In this chapter, we introduce and compare to some closely related work in the areas of hypervisor integrity, OS kernel and application integrity, and kernel rootkit defense.

2.1 Hypervisor Integrity

The first area of related work includes research efforts to verify, monitor, improve, and protect the hypervisor integrity. They are directly related to the HyperSafe and HyperLock systems of this dissertation.

**Hypervisor Integrity Verification and Monitoring** There are recent efforts in applying static analysis to identify and remove software bugs or using formal methods to prove certain security properties. For example, static analysis, model checking, and symbolic execution have long been explored in the area of security research [8, 21, 22, 30, 34]. These systems are designed to uncover bugs in the source code [22, 30]; prevent the bug from being exploited [8]; reason about the safety of one facet of the software [21]; or demonstrate the absence of certain kinds of bugs [34]. Some of these systems can be scaled to analyze commodity OS kernels. However, they typically focus on a small subset of security vulnerabilities or properties. For example, Bugrara et al. [21] validates the safety of Linux kernel’s pointer dereferences in system call arguments, where the safety is defined by the presence of sanity checks of pointers, not the proper use of the dereferenced contents.

There are parallel research efforts [9, 32, 49] that aim to formally prove the safety of system software. Among them, seL4 [49] recently made significant progress. In particular, it proves that the C code of the seL4 microkernel (about 8700 SLOC) implements the behaviors specified in the abstract specification and contains nothing more. However, it is still an open question of how well formal methods can be applied to commodity hypervisors such as Xen (note that Xen 3.4.1 has about 230K SLOC). In comparison, HyperSafe and HyperLock take different approaches to secure commodity hypervisors. As such, both approaches are complementary and our systems can be leveraged to ensure the runtime integrity of a seL4 micro-kernel.
From another perspective, HyperSentry [13] and HyperCheck [100] measure the hypervisor for integrity violations from SMM (system management mode). They can detect that a hypervisor has been compromised after it has already happened. In comparison, our systems aim to protect the hypervisor integrity at the first place.

**Hypervisor TCB Reduction** There also exist related efforts in reducing the hypervisor TCB often by adopting the micro-kernel principles. For example, Xoar [26] applies the approach to partition the control domain (of type-I hypervisors) into single-purpose components. Xen disaggregation [62] shrinks the TCB for Xen by moving the privileged domain builder to a minimal trusted component. KVM-L4 [65] extends a micro-kernel with CPU and memory virtualization to efficiently support virtual machines. NOVA [90] applies the micro-kernel approach to build a bare-metal hypervisor. NoHype [92] removes the (type-I) hypervisor layer by leveraging the virtualization extension to processors and peripheral devices. In comparison, HyperSafe enables the *self-protection* for type-I hypervisors, while HyperLock isolates type-II hypervisors by *replacing* the hypervisor’s TCB in the host kernel with the smaller (12% of the KVM code size) and simpler HyperLock controller.

Nested virtualization, in which CPU’s hardware-virtualization support is virtualized by a L0 hypervisor to run commodity hypervisors (called L1 hypervisor) on top it, may also reduce the hypervisor TCB. Nested virtualization has been implemented for both Xen [14] and KVM [15, 38]. With the nested virtualization support, we can measure or protect the integrity of a L1 hypervisor from the L0 hypervisor by applying the traditional *one-layer-below* approach. However, the L0 hypervisor is still largely left unprotected. HyperSafe and HyperLock can naturally be applied to protect the L0 hypervisors.

**Hardware Support for Static and Dynamic Root of Trust** Also closely related, the Trusted Computing Group [95] has provided foundational work, such as Trusted Platform Module (TPM) [93] and Core Root of Trust for Measurement (CRTM) [95], to enable trusted computing in commodity hardware. The recent Intel TXT technology [43] provides a reliable way called measured late launch to securely load a clean hypervisor (or OS kernel). Trusted computing technologies have been leveraged to provide secure loading (with the guaranteed load-time integrity) [60, 94], integrity measurement [44, 78], and attestation [80, 81, 85]. We also rely on them to ensure hypervisor load-time integrity and further complement these systems by effectively providing runtime integrity to the hypervisor.

### 2.2 Kernel and Application Integrity

The second area of related work aims to protect the integrity of an OS kernel or its applications.

**Virtualization-based Guest Integrity Protection** The first category of related work in this area leverages the virtualization to protect the integrity of OS kernels or running applications. Systems such as [64, 68, 72, 79, 83, 103] take advantage of the isolation and dominant control provided by a trusted hypervisor to secure the integrity of guest OS kernels. With the assumption of the existence of a trusted lower layer, the techniques in these systems cannot be directly applied to protect the hypervisor. In other
words, HyperSafe and HyperLock cannot enjoy the luxury of such a one-layer-below approach since hypervisors already run at the lowest layer.

**Device Driver Isolation** The second category of related work includes systems that isolate faults or malicious behaviors in device drivers. They are related because type-II hypervisors are often implemented as a device driver or loadable kernel module.

Nooks [91] improves OS reliability by isolating device drivers in a light-weight kernel protection domain. Nooks assumes the drivers to be faulty but not malicious. Accordingly, the Nooks sandbox by design lacks instruction access control and malicious drivers cannot be completely isolated by Nooks. A closely related system is SUD [18] that can securely confine malicious device drivers in the user space. SUD relies on IOMMU, transaction filtering in PCI express bridges, and IO permission bits in TSS to securely grant user space device drivers direct access to hardware. However, SUD cannot be applied to the hosted hypervisors such as KVM simply because that hardware virtualization extension (e.g., Intel VT) is not constrained by the IOMMU or other hardware mechanisms that SUD relies on. Microdrivers [35] reduces device driver’s TCB in the kernel by slicing the driver into a privileged performance-critical kernel part and the remaining unprivileged user part. RVM [107] executes device drivers in the user space and uses a reference monitor to validate interactions between a driver and its corresponding device. In HyperLock, we securely confine the privileged KVM code in the hypervisor isolation runtime. Gateway [87], HUKO [112], and SIM [83] are systems that use a hypervisor (e.g., KVM) to isolate kernel device drivers or security monitors. We did not take this approach because otherwise we will face the recursive question of how to isolate the hypervisor that runs at the lowest level.

**Software Fault Isolation** The third category of related work is a series of prior efforts [5, 31, 33, 59, 99, 113] in implementing SFI to confine untrusted code in a host application. HyperLock applies similar technologies to isolate type-II hypervisors.

PittSFIeld [59] and Native Client [113] both apply instruction alignment to enable the reliable disassembly of untrusted code. CFI [5] constrains runtime control flow to the statically determined control flow graph. Among them, Native Client is closely related but with a different application domain in applying user-level SFI to web plugins. As a kernel-level isolation environment, HyperLock needs to address challenges that arise from the needs of enforcing access control of privileged instructions and supporting x86 hardware architecture peculiarities (Section 4.3). XFI [31] and LXFI [58] are two closely related works. Based on CFI and data sandboxing, XFI combines inline software guards and a two-stack execution model to isolate system software. LXFI ensures API integrity and establishes module principals to partition and isolate device drivers. In comparison, HyperLock focuses on the secure isolation of hosted hypervisors and needs to address additional challenges that are unique to virtualization, such as how to prevent `VMWRITE` from being misused or how to securely support memory virtualization.
2.3 Kernel Rootkit Defense

The third area of related work includes efforts to detect, protect, and analyze kernel rootkits. They are related to our third system, HookSafe, that can defeat kernel rootkits by protecting kernel hooks.

**Kernel Rootkit Prevention** The first category of related work in the kernel rootkit defense includes systems that aim at preventing kernel rootkit infection. For example, SecVisor [79] is a tiny hypervisor that uses hardware support to enforce kernel code integrity. Patagonix [55] provides hypervisor support to detect covertly executing binaries. NICKLE [72] mandates that only verified kernel code will be fetched for execution in the kernel space. However, these systems do not protect kernel hooks from being subverted to compromise kernel control flow, which is the main goal of HookSafe.

Lares [64] is a closely related work and our work mainly differs from it in two ways. First, our goal is to enable efficient, large-scale hook protection while Lares is mainly intended to enable reliable active monitoring of a VM by securing the execution path in which a monitoring point has been planted. Second, Lares directly uses hardware-based page-level protection to trap all writes to those memory pages containing kernel hooks. As a result, any write to irrelevant dynamic data in the same physical page will cause a page fault (Section 5.2). When it is applied to protect thousands of hooks that are often co-located with dynamic kernel data, it will lead to significant performance overhead. In comparison, HookSafe recognizes the protection granularity gap and solves it by introducing a hook indirection layer that relocates protected kernel hooks to a page-aligned centralized memory space. By doing so, our approach only incurs small (6%) performance overhead.

**Kernel Rootkit Detection** The second category of related work aims at detecting the presence of kernel rootkits. For example, a number of rootkit detection tools such as System Virginity Verifier [77] validate the kernel code and examine the kernel data (including hooks) known to be the targets of current rootkits. Copilot [67] uses a trusted add-in PCI card to grab a runtime OS memory image and infers possible rootkit presence by detecting any kernel code integrity violations. The approach is extended by follow-up research to examine other types of violations such as kernel data semantic integrity [66] and state-based control flow integrity [68]. Livewire [37] pioneered the virtual machine introspection methodology to inspect the inner states of guest VM to detect malware infections. Other systems such as Strider GhostBuster [101] and VMwatcher [45] leverage the self-hiding nature of rootkits to infer rootkit presence by detecting discrepancies between the views of a system from different perspectives. Note that all these approaches were proposed to detect kernel rootkits after the system has been infected. In comparison, HookSafe targets at preventing rootkits by protecting kernel hooks and kernel text from being manipulated by them.

**Kernel Rootkit Analysis** There also exist a number of recent efforts [53, 73, 104, 114, 115] on analyzing and profiling kernel-mode malware, especially kernel rootkits. The goal of these efforts is to enrich our understanding on stealthy malware, including their hooking behavior and targeted kernel objects. Specifically, Panorama [115] performs system-wide information flow tracking to understand
how sensitive data (e.g., user keystrokes) are stolen or manipulated by malware. HookFinder [114] applies dynamic tainting techniques to identify and analyze malware’s hooking behavior. HookMap [104] monitors normal kernel execution to identify potential kernel hooks that rootkits may hijack for hiding purposes. K-Tracer [53] makes a step further to systematically discover system data manipulation behavior by rootkits. PoKeR [73] defines four key aspects of kernel rootkit behaviors and accordingly proposes a combat tracking approach to efficiently characterize and profile them. HookSafe has a different goal: instead of locating the kernel hooks hijacked by rootkits, it is designed to protect them from being hijacked. Therefore, HookSafe is complementary to the above systems and thus can be naturally integrated together with them.
Chapter 3

Providing Control Flow Integrity for Type-I Hypervisors with HyperSafe

3.1 Introduction

Recent years have witnessed the wide adoption of virtualization in today’s computing systems. The unique security advantages from virtualization, especially in isolating and introspecting commodity OSes as virtual machines (VMs), have prompted a wave of research [37, 45, 53, 64, 72, 73, 79, 83, 103, 114]. For example, Livewire [37] pioneers the concept of VM introspection and applies it for system monitoring and malware detection. SecVisor [79], NICKLE [72], VMwatcher [45], Lares [64], HookSafe [103], and SIM [83] leverage virtualization to protect the guest OS kernel integrity or enable reliable monitoring of OS kernel behavior. Most recently, a number of virtualization-based system debugging and analysis tools such as K-Tracer [53], PoKeR [73] and AfterSight [24] have been developed to examine system anomalies and study kernel-mode malware, which is considered difficult or in some situations impossible to do with conventional approaches. One fundamental assumption shared by all these research efforts is the need for a trustworthy hypervisor (or virtual machine monitor - VMM). However, as mentioned earlier in Chapter 1, existing type-I hypervisors are unavoidably vulnerable and there is a pressing need to investigate effective and efficient ways to secure the hypervisor [74].

One natural but challenging approach is to formally verify that the hypervisor is secure. For example, the L4.verified [51] project aims to guarantee the functional correctness of a micro-kernel implementation, i.e., seL4 [49], by formally proving that the C code implementation (with ~8700 SLOC) correctly and precisely follows the abstract specification and contains nothing more. This is very helpful as it can lead to strong security guarantees, especially in proving the absence of certain types of software bugs (e.g., buffer overflows and null pointer dereferences). However, to perform a formal proof, it imposes several stringent requirements on the micro-kernel design and implementation. For example, the kernel should run with interrupts mostly disabled and no address-of operator (&) and function calls through
function pointers will be allowed. Also, the memory management component is moved out of the kernel space and exempted from being formally proved. Further, besides its inherent scalability constraint, we also notice that the proven functional correctness from a manually-specified specification does not necessarily equal the actual safety properties of the system. As a result, though it is an attractive approach, significant efforts are still needed to make it suitable for commodity hypervisors as their designs are not constrained by these restrictions.

From another perspective, we can tackle this hard problem by guaranteeing runtime hypervisor integrity despite the presence of exploitable software bugs. Common wisdom holds that to secure a running application, one runs monitoring software a layer below the application. However, this is not applicable here, simply because the hypervisor already runs at the lowest level of the software stack. It may be argued that a nested hypervisor can be developed to run underneath and protect another hypervisor running above. However, a fundamental question of the same nature still remains: “how to protect the hypervisor running at the lowest-level?”

Existing hardware-based technologies including TPM [93] and measured late launch [43] are capable of effectively establishing static/dynamic root of trust by guaranteeing the loading of a hypervisor in a trustworthy manner. In other words, they can guarantee the load-time integrity of the hypervisor. However, the main challenge is how to maintain the same level of integrity continuously throughout the lifetime of the hypervisor. Due to the fact that we cannot rule out the presence of software vulnerabilities in the hypervisor, we have to address the threat that after the hypervisor is securely loaded, these vulnerabilities may be immediately exploited to sabotage its integrity.

In this chapter, we present the design, implementation, and evaluation of HyperSafe, a system that reliably establishes the continuous integrity of the lowest-level software on a system, i.e., the hypervisor. Specifically, continuous integrity in this chapter is enforced in the form of lifetime control flow integrity [5]. Our system is lightweight and can be integrated into commodity hypervisors\(^1\) without requiring specialized hardware support. And unlike the previously mentioned common wisdom, even though HyperSafe is a natural part of the hypervisor, it preserves via a self-protection mechanism the lifetime hypervisor integrity.

In particular, HyperSafe implements two key techniques: The first one is non-bypassable memory lockdown, which essentially serves as the cornerstone for the entire scheme and enables the unique hypervisor self-protection. Specifically, once a memory page is locked down, this technique guarantees that the page needs to be unlocked first – even for legitimate hypervisor code – in order to modify the page. And by design, the unlocking logic will simply disallow any attempts that will either modify existing hypervisor code or bring external (malicious) code for execution in the hypervisor space. In other words, this technique locks down those write-protected memory pages (containing hypervisor code and read-only data) as well as their attributes (in the page tables) and prevents them from being changed.

---

\(^1\)Considering the various flavors of hypervisor implementations (Section 3.2), we focus on those Type-I hypervisors. Some examples of these hypervisors are Xen [14], VMware ESX [98], and BitVisor [86].
at runtime, thus effectively achieving hypervisor code integrity. We highlight that the enforcement cannot be bypassed even in the presence of potentially exploitable memory corruption bugs such as buffer overflows.

The second key technique is restricted pointer indexing, which essentially leverages the memory lockdown technique to expand the protection coverage from hypervisor code to control data. Notice that when used in related control transfer instructions (e.g., call/jmp/ret), the control data can directly impact the control flow of hypervisor execution. Their security implications become evident, especially with the recent exposure of return-oriented programming [40, 82]. Unfortunately, we cannot directly apply the memory lockdown technique to protect all the control data, as some of them (e.g., return addresses in the stack) will be dynamically generated. To address that, we observe the potential control flow always follows the control flow graph, which can be predetermined ahead of time. With that, we can convert the control data (also called pointers in this chapter) into pointer indexes and restrict them to be conformant to the control flow graph. In other words, we can pre-compute possible control flow targets, save them in the target tables, and restrict the accesses from pointer indexes to them. Since these target tables are static, we can directly leverage the memory lockdown technique to protect them. Consequently, the protection of the hypervisor integrity is expanded from the code to the control data for control flow integrity.

To the best of our knowledge, HyperSafe is the first system that is capable of providing hypervisor control flow integrity. To validate our approach, we have implemented a prototype of HyperSafe and applied it to protect two open source Type-I hypervisors, i.e., BitVisor [86] and Xen [14]. Specifically, the first key technique is implemented by directly modifying the hypervisor source code while the second key technique is implemented as a compiler extension to the re-targetable LLVM framework [56], which is thus hypervisor-transparent. As a result, the BitVisor/HyperSafe prototype is a full implementation with both key techniques. For the Xen port, since the current LLVM release does not support compiling Xen yet, our current prototype only enables the non-bypassable memory lockdown, which still guarantees the nontrivial code integrity of Xen. Our prototyping experience indicates that HyperSafe’s code size is small and its integration with commodity hypervisors is straightforward. Evaluation with synthetic hypervisor attacks as well as a number of performance micro-benchmarks and user applications show that the integrity protection can be effectively enabled with less than 5% performance overhead.

The rest of the chapter is structured as follows. We first show the overall system design, discuss the threat model, and present the two key techniques in Section 3.2. We show implementation details in Section 3.3 and present our evaluation results in Section 3.4. We discuss the support of Type-II hypervisors as well as possible limitations and improvements in Section 3.5. After that, we conclude this chapter in Section 3.6.
3.2 Design

3.2.1 Goals and Assumptions

In order to provide lifetime hypervisor control flow integrity, we have three main design goals. First, the proposed techniques should enable the self-protection of commodity hypervisors. As the name indicates, self-protection may not introduce new lower level mechanisms. Further, the self-protection mechanism needs to be reliable in the presence of exploitable memory corruption bugs (e.g., buffer overflows or format string bugs) and effective in proactively preventing attacks from gaining execution control over the hypervisor.

Second, the proposed techniques should not require re-structuring or impacting the original hypervisor design while still providing the desired integrity guarantee. In other words, the proposed techniques need to be generic and amenable to commodity hypervisors without limiting the design choices or imposing implementation restrictions (e.g., in disabling certain programming language features). We may need to tolerate some minor modifications to commodity hypervisors, but the modifications should be minimal. Also, based on the traditional classification between Type-I bare-metal and Type-II hosted hypervisors and the fact that Type-II hypervisors require a hosted OS kernel, our focus in this chapter is the support of Type-I hypervisors.

Third, the proposed techniques can be efficiently implemented on commodity hardware, i.e., without relying on sophisticated hardware support to achieve the integrity guarantee or obtain reasonable performance for deployment. Given this requirement, the challenge is to ensure that the proposed techniques can be implemented on top of commodity hardware, have a small footprint, and remain lightweight with respect to performance impact.

Threat model and system assumption In this work, we assume an adversary model where attackers are able to exploit software vulnerabilities in an attempt to overwrite any location in memory. However, to successfully launch an attack, attackers will have to either inject and execute their own
code or leverage and misuse existing code. Note this represents a powerful adversary model as attackers can attempt to inject code, modify existing code, and exercise more sophisticated attacks such as return-oriented ones [40].

In the meantime, we do assume trustworthy hardware, especially the TPM [93]-assisted static/dynamic root of trust [94], which can be leveraged to guarantee load-time hypervisor integrity. Due to the unsafe programming language used in the implementation, we do assume the presence of vulnerabilities in the hypervisor. However, the attackers are restricted in their attempts to subvert the hypervisor integrity by only exploiting these vulnerabilities, not by out-of-band attacks (e.g., TLB cache or SMM exploitation [108, 109]) or layer-below attacks (including physical-level attacks). Malicious DMAs [110] are not considered as they can be readily blocked with hardware-based IOMMUs [43]. Notice that our attack model is similar to that used in SecVisor [79]; however, one key distinction is that HyperSafe is designed to protect the hypervisor itself while SecVisor aims to protect the guest kernel code integrity using a small trusted hypervisor.

Based on this threat model, we propose two key techniques, i.e., non-bypassable memory lockdown and restricted pointer indexing, to enable the self-protection of commodity hypervisors. Figure 3.1 shows a break-down of the required hypervisor integrity guarantees as well as the corresponding key techniques we propose to achieve them. Next, we will describe in detail these two key techniques.

### 3.2.2 Key Technique I: Non-Bypassable Memory Lockdown

As mentioned earlier, this technique serves as the cornerstone for the proposed hypervisor integrity protection. In the following, we first give a brief overview of the available memory protection mechanisms in the Intel x86 architecture on which our system is developed. In essence, the x86 architecture supports two types of memory protection: segmentation and paging. With the introduction of the new 64-bit mode of the x86 processor, segmentation has been mostly disabled in favor of paging.\(^2\) Because of that, our discussion will be focusing on paging and our system relies on paging-based memory protection.

Specifically, the paging-based memory protection divides the virtual address space into pages, and physical memory into frames of the same size. The translation from the virtual page to the physical frame is facilitated by the page tables. Each page table has a number of page table entries (512 in x86_64). Each entry contains certain bits to specify the corresponding page protection attributes, such as whether the page is writable (the \(R/W\) bit), executable (the \(NX\) bit), or requires privileged access (the \(U/S\) bit). Different from the virtual page’s privilege levels, the CPU has four privilege levels (or rings) from 0 to 3 with 0 being the highest privilege. The code running in privilege level 3 can only access user pages while the code running in privilege levels 0, 1 and 2 is considered to be supervisor code and can access both user pages and supervisor pages.

\(^2\)Some specific segments such as FS and GS may still be retained to facilitate the addressing of local data or certain OS data structures.
With these protection attributes, paging-based memory protection allows for flexible customization to each and every individual page. For example, one common usage of these attributes in commodity OS kernels (e.g., Windows, Linux, and OpenBSD) is to write-protect their code and read-only data. Another similar one is to establish the $W\oplus X$ property of the OS kernel to ensure its code integrity as demonstrated in a few recent systems [72, 79].

Similarly, we are motivated to enforce the $W\oplus X$ for hypervisor integrity protection. However, there are several notable pitfalls. First, for historical reasons [72], commodity OS kernels may allow the presence of mixed memory pages that contain both code and data. Certainly, the presence of such pages directly violates the $W \oplus X$ property and should be avoided in the hypervisor. Second, for performance and efficient resource sharing purposes, existing OS kernels typically allow the mapping of several virtual pages to the same physical frame and different virtual pages may possibly have conflicting protection attributes. Such double mapping indirectly breaks the $W \oplus X$ property and should not be allowed in the hypervisor either. Third, most importantly, the $W \oplus X$-based integrity enforcement largely relies on the integrity of page tables. For a write-protected page to be modifiable, the corresponding page table entries will need to set in a way to allow it. Unfortunately, in current hypervisors (e.g., Xen and KVM) and OS kernels (e.g., Windows and Linux), their page tables are all writable! This implies that even if a hypervisor ideally sets these memory protection attributes, the enforcement can be easily bypassed since the page tables are writable. Our experience indicates that the ability to modify even one bit in a page table entry could well be enough to subvert the entire protection.

From another perspective, if we assume the proper initialization of the hypervisor page tables (i.e., no mixed pages, no double mapping, and a correct $W \oplus X$ setup for each memory page), the attacker will be forced to first manipulate the page tables in order to bypass the $W \oplus X$ protection. This observation motivates us to also write-protect the page tables. By doing so, we can ensure that no code including legitimate code will be able to modify the write-protected hypervisor code (and related control data – Section 3.2.3). As mentioned earlier, in order to proceed with any modification, the page tables need to be changed to allow it. But the write-protection of page tables disallows such a change. Consequently, any write attempt to them (including the page tables) will be hardware-trapped into the page fault handler, which, as a part of legitimate code, is unable to modify them either. This is a strong guarantee that serves as the basis to establish and sustain hypervisor runtime integrity.

The above strong guarantee is desirable for hypervisor protection as it can effectively prevent malicious updates to page tables. Unfortunately it will also trap and block all benign updates. Again, the reason is that once paging is enabled, the hypervisor can only access its memory through virtual addresses, which will be translated and subjected to protection checks by the page tables. As a result, the creation of read-only page tables immediately leads to an unsolvable paradox: read-only page tables can

---

3 Note that although the CPU’s hardware translates virtual addresses with page tables, the hardware’s accesses are not translated as the CPU uses physical addresses directly. Therefore the CPU has no trouble at all to read and update them, e.g., to set $A$ (accessed) and $D$ (dirty) bits.
detect and deny any malicious manipulation but they also make the benign changes impossible.

To accommodate benign page table updates, we need to design a secure way to temporarily bypass the enforcement without being misused (e.g., by return-oriented attacks). This is how our technique – non-bypassable memory lockdown – comes into play. Specifically, our technique uses a hardware feature called the \textit{WP} (Write Protect) bit in the machine control register CR0 \cite{intel}, which has existed in all x86 CPUs since the Intel Pentium. The \textit{WP} bit controls how the supervisor code interacts with the write protection bits in page tables: If the \textit{WP} bit is turned off, the supervisor code has unfettered access to any virtual memory (i.e., the write-protection is ignored). Otherwise, the write protection attributes in page table entries will decide whether the supervisor code can write to the memory page or not. Note the \textit{WP} bit was originally introduced to facilitate the Copy-On-Write (COW) implementation of forking a new process. More specifically, in Linux, when a process forks, memory pages are COW-shared (or marked as read-only) between parent and child processes. Therefore, any write to a COW-shared page leads to the creation of a new copy of the page and the sharing can then be removed. As a result, OS kernel can simply set the \textit{WP} bit to trap its own writes to these pages, which greatly simplifies the COW design and implementation. (Otherwise, OS kernel must check for COW every time it writes to user space.)

With that, we can initially mark the page tables read-only and turn on the \textit{WP} bit to lock down any page table updates, regardless of their intent being benign or malicious. To allow benign ones to proceed, we can instead escort them by temporarily clearing the \textit{WP} bit right before each update and re-enabling the bit right after. Naturally, the entire escort operation needs to be atomic (e.g., with interrupts disabled). Otherwise the attackers may potentially interrupt the operation and leave the \textit{WP} protection off. Within the escort operation, HyperSafe can further validate that the new page table entries conform to the security policy, which can be specified by the hypervisor developer. In our implementation, we enforce a simple invariant that denies page table updates that attempt to change the protection attributes of hypervisor’s code and data or introduce a double mapping. Though it may appear that this security check is redundant once the hypervisor’s control flow integrity is protected, we show that this is not the case.
case in Section 3.4.1. Figure 3.2 shows the comparison between traditional page table updates and the new page table updates in HyperSafe.

It is worth mentioning that to protect the hypervisor page tables, we also need to protect related machine control registers and data structures. For example, the hypervisor’s page table base address is contained in CR3, which should not change after the initialization. The same also applies for the entries in the GDT (Global Descriptor Table) and the IDT (Interrupt Descriptor Table). In addition, the hypervisor virtualizes the guest’s memory by using either shadow page tables (SPTs) or nested page tables (NPTs). The updates to them need to be protected in a similar way to prevent the attacker from gaining the control over the hypervisor’s memory (e.g., by mapping it to a compromised guest).

To summarize, our technique effectively locks down hypervisor memory pages to strictly specify and manage memory protection attributes. Most importantly, the enforcement of these memory attributes such as $W \oplus X$ is non-bypassable (Section 3.4.1). As a result, we can reliably provide hypervisor code integrity. Next, we will present another key technique that essentially expands the protection coverage to control data and enables control flow integrity.

### 3.2.3 Key Technique II: Restricted Pointer Indexing

Control flow integrity (CFI) is a powerful security measure, which strictly dictates the software’s runtime execution paths. If the software’s runtime paths follow the statically determined control flow graph, attackers can be prevented from arbitrarily controlling the execution flow of the system. Based on how control is transferred, there are two types of control flow transfer instructions: direct and indirect. A direct control transfer is initiated by a direct function call where the destination is encoded in the machine code in the form of an absolute address or a relative offset. Accordingly, control flow integrity from direct control transfers is maintained as long as the code integrity is guaranteed.

An indirect control transfer, which is our main focus, can be caused by two sets of instructions: indirect call/jmp instructions (where the destinations may be specified in registers or memory) and the ret instructions. Each ret has an implicit destination on the top of the current stack. Correspondingly,
there are two types of control data: function pointers and return addresses. For simplicity, we also call them pointers. Due to the dependence of indirect control flow on these pointers, we need to protect their integrity to preserve indirect control flow integrity.

Unfortunately, we face three main challenges: First, control data can be widely scattered in memory and can co-exist together with other dynamic data on the same pages [103]. Naive page level protection will likely lead to huge performance overhead. Second, some control data can be dynamically generated and thus their locations cannot be determined a priori. A representative example is the return address. This implies any protection scheme that requires their locations to be static will fail. Third, some control data such as return addresses can be updated at a high frequency. This invalidates any approach that requires write-protecting frequently updated control data. Our experience with a local build of the Xen hypervisor (version 3.4.1) for the x86-64 architecture indicates that there is a call instruction on average for every 23 machine instructions in the binary and a ret instruction for every 79 instructions. At runtime, each call instruction will push a return address onto the stack, which will then be popped off by the corresponding ret instruction.

From another perspective, we notice that though the control data may be dynamically generated or frequently updated, their contents always fall in a data set that can be determined offline. As such, we can aggregate them into individual target tables and, by introducing one layer of indirection, replace each control data with a restricted index to the target table (hence the name restricted pointer indexing). More specifically, the target table contains all the legitimate destinations for an indirect control flow instruction allowed by the hypervisor program’s control flow graph (CFG). For each indirect call/jmp, its table contains the function entry points it may enter. Similarly, the target table for a ret includes all the return addresses it may return to.

Based on the target tables, HyperSafe can essentially replace all the runtime control data in the hypervisor program with their indexes in the target tables. To perform a control transfer, the indirect control flow instruction will be instrumented to convert the index back to the destination address (e.g., by looking up the index in the table). For that, we need to take the following two steps:

First, the instructions that introduce the control data into the hypervisor program must be converted to use the indexes instead. For simplicity, we call these instructions source instructions. The source instruction for a return address is the related call that pushes the return address onto the stack. As a result, the call instruction will be instrumented into two instructions: one pushes the index onto the stack and another jmps to the function entry point. For an indirect call, its source instruction is an earlier instruction that loads the function address to the register or memory. Unlike the return address case, the function pointer can possibly appear in the data section (e.g., as a member of an initialized global object or variable). As a result, we can leverage the compiler to identify and convert them.

Second, the instructions that consume the control data from the hypervisor program must be converted to translate the indexes back to their destination addresses. Similarly, we call these instructions sink instructions. Return addresses will be used by the ret instructions while function pointers will be
consumed by indirect \textit{call/jmp} instructions. During instrumentation, a \textit{ret} will be converted to a sequence of instructions to pop the index off the stack, convert it into the return address, and then return to it. An indirect \textit{call/jmp} will be converted to use the index to locate the function entry point and then continue execution there.

Based on the above instrumentation, an indirect \textit{call} acts as a sink instruction for the consumed function pointer and a source instruction for the dynamically-pushed return address. Therefore, it will be instrumented twice. There may also exist other instructions that access the control data but are not the source and sink instructions. Among them, some instructions can be left intact if the contents of the control data are not explicitly examined by them. One example is the \textit{mov} instruction that copies the index to and from registers or memory. Instructions that compare two function addresses do not need instrumentation either if we assign the pointer indexes in the order of their addresses. On the other hand, instructions that examine the contents of control data must be expanded to convert indexes into original control data. A general solution is to discover and convert all such instructions, ideally by the compiler. Fortunately, very few instructions will touch return addresses on the stack. If they do, most likely they are implemented in assembly and thus we can instrument them manually. For function pointers, most accessing instructions are \textit{mov} or \textit{cmp}. In this case, the contents of the function pointers are not examined and we can safely keep these instructions as is.

In Figure 3.3, we show the control flow for an instrumented \textit{call/ret} pair in HyperSafe when compared to the original pair. In the figure, the original \textit{call} has been instrumented to fetch the index from \textit{eax}, convert it to a function entry point by indexing into its target table, and then jump to the function. By substituting indexes for control data, HyperSafe limits the destination of a runtime control transfer to only those explicitly specified in the target table. In other words, indirect instructions can only transfer control to the targets allowed by the CFG. Moreover, because all the destination addresses are known beforehand from the hypervisor program binary, these target tables can be pre-computed offline. At runtime, they are protected by directly applying the memory lockdown technique.

Furthermore, with the help of the target tables, HyperSafe can flexibly control the precision of control flow integrity. In one extreme case, we can simply use two big tables: one is for all the \textit{ret} instructions (with all valid return addresses) and the other one is for all the indirect \textit{call} instructions (with all possible indirectly-called functions’ entry points). This scheme provides the least precision, resulting in coarse protection: namely a \textit{ret} can return to any valid return address in the hypervisor program; and an indirect \textit{call} can call any indirectly-called function. On the other extreme, each indirect \textit{call} has its own target table, and all \textit{ret} instructions inside the same function share one target table. In other words, each function has a dedicated table for all of its returns. By doing so, we can provide the finest control over what targets indirect instructions can transfer control to. Note that there is no need to use one target table per return instruction since all the \textit{ret} instructions in a function always have the same set of return addresses.
As pointed out in [5], the major factor that impairs the precision of control flow integrity is the so called destination equivalence effect. That is, two destinations are considered to be equivalent if they connect to a common source in the CFG. Further, the equivalence relation is transitive. In Figure 3.4, we show an example of the destination equivalence effect on the ret instructions. In this figure, there are one indirect call instruction and two direct call instructions. The indirect call may invoke both functions func_i and func_j while the two direct calls execute func_i and func_j, respectively. R1, R2 and R3 are the corresponding three return addresses. From the figure, the function func_i can return to R1 and R2, and the function func_j can return to R1 and R3. Because of the destination equivalence effect, R1, R2 and R3 are all equivalent in this example. More specifically, since R2 is equivalent to R1 and R1 is equivalent to R3, based on the transitivity of the equivalence relation, R2 is equivalent to R3. The destination equivalence effect also indicates that a return address has the same index in each target table that contains it. This is obvious since only one index can be assigned to a specific destination. In our example, R1, R2 and R3 forms one equivalent group, and two ret instructions in func_i and func_j can return to them. If one table per function is used to enforce the control flow integrity, we can use a table “R1, R2, error” for the ret instruction in func_i, and another table “R1, error, R3” for the ret instruction in func_j, where error denotes a special destination to trap an impossible control transfer. Therefore, our one-table-per-function-based control flow integrity enforcement policy is more precise than the one originally proposed in [5], where R1, R2 and R3 will bear the same label ID and both ret instructions can legitimately transfer control to all of them. In particular, in [5], the function func_i can legally return to R3 and func_j can legally return to R2. In comparison, our scheme can flexibly handle the destination equivalence effect and make these two paths simply impossible in HyperSafe.
3.3 Implementation

We have implemented a prototype of HyperSafe and applied it to protect two open-source Type-I hypervisors, i.e., BitVisor \([86]\) (with \(\sim 190\)K SLOC) \(^4\) and Xen \([14]\) (with \(\sim 230\)K SLOC). In particular, the first technique – non-bypassable memory lockdown – is implemented by directly extending their memory management modules. For the second technique – restricted pointer indexing, we choose to extend the open-source LLVM compiler so that we can enable it by simply re-compiling the hypervisor code with the modified compiler. Our development environment is a standard 64 bit Ubuntu 9.10 desktop. As mentioned earlier, the BitVisor port is a full implementation, while the Xen port only contains the non-bypassable memory lockdown feature, which nevertheless guarantees the nontrivial code integrity of Xen. Meanwhile, our current prototype integrates the trusted booting software, i.e., tboot \([94]\), to protect the load-time integrity. After the hypervisor is successfully loaded, HyperSafe will then ensure its runtime integrity. In the following, we focus on the BitVisor port as an example to present our implementation details.

3.3.1 Non-Bypassable Memory Lockdown

The key novelty of our system is the non-bypassable memory lockdown technique for hypervisor integrity protection, achieved purely based on commodity hardware support. Specifically, HyperSafe write-protects the hypervisor’s page tables and turns on the WP bit in \(CR0\) to initiate the memory lockdown. Our system requires only minimal modifications to the supported hypervisors, therefore satisfying the second design goal (Section 3.2). Specifically, in our BitVisor prototype, we only added or changed 521 lines of C code and 9 lines of assembly code. To avoid potential pitfalls in \(W\oplus X\) enforcement (Section 3.2), we adjust the link script to align related sections to avoid mixed pages and at runtime disallow double mappings.

In our prototype, we reserved the top \(128\)MB physical memory for BitVisor. This memory is mapped \(1 : 1\) to the virtual address \(0x40200000\). A \(32\)MB memory range, starting at the virtual address \(0x40800000\), is reserved as the shared page table pool from which all the hypervisor’s page tables are allocated. After secure booting from tboot, the hypervisor properly initializes the page table data structure, turns on the \(WP\) protection in the \(CR0\) register, and then enables the paging mode. After entering the paging mode, every virtual memory access will be automatically translated through page tables. Because of that, all the page tables have to be accessible and mapped in the hypervisor’s virtual address space. In BitVisor, since all the page tables are allocated from and mapped in the page table pool, we simply set the whole page table pool as read-only to lock the page tables. To accommodate benign updates, our system first traverses through the page table hierarchy to locate the affected page table entries, and then escorts their updates to guarantee that existing hypervisor code will not be modified and no external code will be introduced for execution.

\(^4\)In our prototype, we disabled the VPN support in BitVisor as it is not relevant.
After the page tables have been write-protected, any write attempts to modify them at runtime (e.g., either by legitimate hypervisor code or malicious code injected due to a successful exploitation) will be trapped. Inside the page fault handler, we will enforce an unlocking logic that simply preserves the W⊕X property. In the meantime, there also exist a number of legitimate reasons for the hypervisor to update its page tables without violating the W⊕X property. For example, the hypervisor may need to map part of guest memory pages or device memory for its access. This mapping is typically temporary as it will be removed immediately after the hypervisor has accessed it. For that, instead of triggering a page fault, the hypervisor first turns off the WP protection, updates related page table entries, and turns it back on. The BitVisor implementation provides several helper routines, i.e., *pmap* _wr64_ and *pmap*_ *wr32*, that are used to update page table entries. Our prototype wraps these routines by adding additional inline assembly code to turn WP protection on and off. To ensure that such escort operations are atomic, our prototype disables the interrupts when an escort operation is in progress. Further, in order to prevent the misuse of these routines, HyperSafe validates whether the change is benign. Our current prototype simply denies any change to the permission of the hypervisor’s code and data sections after initial setup. Also, it disallows the double mapping of the hypervisor’s code and data sections. This check can be implemented efficiently by verifying the page table update against various address ranges (e.g., physical or virtual address ranges of hypervisor’s code and data sections). As discussed in Section 3.4.1, this check is *not* redundant even in the presence of a control flow integrity guarantee.

In our prototype, we use the memory lockdown feature to write-protect not only the hypervisor’s code, but also its static data. Some examples of this data include the entries in the GDT, the IDT, and various target tables. As mentioned earlier, all these data structures are security-critical and should be write-protected by HyperSafe. For guest page tables, there are two main virtualization modes: shadow page table (SPT) and nested page table (NPT). In the SPT mode, hypervisor “shadows” the guest page tables by maintaining a corresponding copy, i.e., the shadow page tables. The CPU translates the guest virtual address directly to the host physical address using these shadow page tables. The hypervisor also traps updates to the guest page tables and synchronizes shadow page tables with them. In the NPT mode, there are two levels of page tables used by the hardware. The CPU first translates a guest virtual address to a guest physical address with the guest page tables, and then maps that guest physical address to a host physical address based on nested page tables maintained by the hypervisor. Note that the shadow page tables (if running in the SPT mode) or the nested page tables (if running in the NPT mode) need to be protected so that the hypervisor memory will not be accidently mapped and accessible to the guest.

### 3.3.2 Restricted Pointer Indexing

Our second technique replaces all the control data with their indexes and essentially leads to the protection of control data to enforce control flow integrity. Specifically, it first aggregates all the possible destination addresses (function entry points and return targets) in a few read-only target tables, and then
replaces the destination addresses used by the program with their indexes. By doing so, we can guarantee that an indirect control flow transfer instruction that utilizes one of these protected pointers will transfer the control only to the addresses specified in the target tables. As discussed earlier, we can also flexibly control the precision by adjusting these target tables to handle the destination equivalence effect. In our prototype, we implemented a scheme that uses one table for each indirect call/jmp instruction and one table for each function (applicable for all the ret instructions contained in the function). The table per indirect call/jmp instruction contains all the function entry points it may call. Similarly, the table per function has all the valid return addresses for this function. As such, the target tables reflect the hypervisor’s control flow graph.

Our technique is implemented as a compiler extension to the re-targetable LLVM framework. We choose it because it is a production-quality open source compiler infrastructure that has a modular design and can be flexibly extended. In particular, our extension includes a program analysis and optimization phase that can be applied to the intermediate representation (IR) of a program. In essence, it integrates the supported alias analysis to build and generate the target tables while extending the compiler backend to instrument instructions for the use of target tables. Note the target tables contain information for both direct and indirect calls. The control flow graph for the direct calls is simple to extract because their targets are already encoded in the instructions. For the control flow graph of indirect calls, we extend the existing alias analysis in LLVM. Specifically, we leverage the data structure analysis implemented in LLVM to identify possible call targets. Note that the data structure analysis is a context-sensitive, field-sensitive unification-based pointer analysis. As a result, it is considered conservative. During our implementation, we found that it is relatively effective to analyze C code. However, it is unable to handle assembly, which is one common limitation of existing alias analysis tools.

In our prototype, we realize that one specific way BitVisor saves and utilizes function pointers foils the data structure analysis. In particular, to facilitate multi-core support, BitVisor keeps a per-cpu data structure and accesses it with the help of the gs segment. (Note gs is one of the two segments that x86_64 keeps to allow for such access.) With that, each processor can set its gs segment’s base or gs_base to a location different from other processors. And the hypervisor can conveniently access the
per-cpu data with the \textit{gs : of f set} addressing mode, which will be translated by the current CPU into the \textit{gs \_base + of f set}. As assembly code is required to load the per-cpu data, the data structure analysis in LLVM is unfortunately unable to uncover all the call targets. To handle that, we manually went through the indirect calls that were not able to be analyzed and then leveraged our domain knowledge to resolve them. Our experience shows that there are 126 indirectly called functions and 360 indirect call sites in BitVisor, and the data structure analysis was able to extract indirect call targets for about three fourths of them. Therefore, we had to manually analyze the rest. We point out that most of these manual efforts only need to be done once, though ideally we still need an automated approach. For that, the recent efforts \cite{19} on alias analysis at the assembly level can be naturally integrated and extended.

Once the complete call graph is derived, it is a rather straightforward process to generate the target tables. In our prototype, we use the standard union-find algorithm to locate the equivalent targets and assign the same index to the destinations if they appear in multiple target tables. Also, our prototype assigns \textit{error} to certain target table entries if they are not reachable in the call graph. In our build of the hypervisor, it turns out that there are 681 target tables for \textit{ret} instructions and 360 target tables for indirect \textit{calls}. If we include direct calls, there are 4100 call sites in total.

After building the control flow graph and generating the target tables, we extend the compiler back-end to instrument relevant source and sink instructions. In our build environment, the compiler first compiles the C source into assembly code, and then generates the executable binary with GAS, the GNU Assembler. Our prototype extends a LLVM component called AsmPrinter to generate the instrumented assembly code. Specifically, a \textit{call} instruction is instrumented to push the index of its return address to the stack, while a \textit{ret} is instrumented to pop the index off the stack, fetch the actual return address from its target table, and continue execution there.

To better describe the scheme and discuss one possible optimization we identified and implemented in our system, we consider the instrumentation of a direct \textit{call}. In Figure 3.5, we present the results after instrumenting a direct \textit{call} to the function \textit{pmap \_read} and the corresponding \textit{ret} in that function. Basically, the instrumented \textit{call} instruction pushes the index of the return address to the stack followed by a jump to the target function. The \textit{ret} instruction is rewritten to first pop the index off the stack, then load the actual return address from its target table (\textit{RT \_pmap \_read}) and resume execution to it. In this example, \textit{RT \_pmap \_read} is defined as a static global variable to contain the return addresses of 58 calls to \textit{pmap \_read}. For alignment purposes, the table actually contains 64 elements and invalid elements are filled in with \textit{error} – the address of an error reporting routine to trap illegal control flows. In the meantime, it also allows HyperSafe to prevent possible index overflows with one \textit{and} instruction (the second instruction in the instrumented \textit{ret} as shown in the figure). To reduce the performance overhead, we use a caller-saved register, i.e., r8, to facilitate the instrumentation and avoid unnecessary register spilling.

Furthermore, in the case that a target table only contains a single entry, we apply an optimization that avoids the instrumented \textit{ret} instruction to read the target address from the target table. Specifically,
the instrumented ret will be a simple add that increments the rsp register by 8 bytes to remove the return address index from the stack, followed by another direct jmp to the corresponding target. Note this optimization has performance benefits and is made possible because the function is only called from a particular call site. When compared to the unoptimized case (Figure 3.5), this optimization saves two memory accesses (in pop and mov, respectively) and replaces the indirect jmp with a direct jmp. In our prototype, we found that 291 out of 681 target tables for return instructions contain only one entry and their optimizations greatly contribute to reducing the performance overhead of the system.

The indirect calls are handled in a similar way. Since the targets for these indirect calls are function entry points, we need to convert the uses of function entry points to their indexes. Accordingly, we convert those source instructions (e.g. mov) that load function addresses into registers to load their indexes instead. Note the locations of these source instructions are easy to identify in the compiler since one of its operands is actually the symbol of the function. After that, we replace the function addresses in data structures with their corresponding indexes. Our system instruments the corresponding sink instructions (at the corresponding indirect call sites) to examine the indexes from their operands and convert them back to the function entry points (i.e., by indexing into the target tables). Similarly, we can also apply the previous optimization if a particular indirect call has only one target in the target table. By doing so, we can replace the indirect call with a direct call and improve performance by avoiding one memory read (of the function pointer). In our prototype, we found 294 out of 360 target tables for indirect calls contain only a single entry and can thus be optimized. Further investigation shows that most of these optimized indirect calls are due to the specific way taken by BitVisor to support the two existing X86 hardware virtualization architectures, namely Intel VT and AMD SVM. In particular, BitVisor defines a common interface for the support of Intel VT and AMD SVM. The common interface contains a set of function pointers to abstract the difference between them so that the upper layer software can be shielded from low level details. With that, since only one architecture is possible at runtime, we optimized these function pointers at the compiler time according to the CPU used. Besides these source and sink instructions, we do not observe the presence of any other instructions that examine the function pointers’ content. In other words, there is not a single arithmetic operation on function pointers.

It is also interesting to mention that in the early development of our second technique, we explored another way to handle ret instructions: shadow stack. Specifically, each call pushes a copy of the return address to the shadow stack and each ret fetches the return addresses from both the shadow stack and the original stack and then compares them to detect any corruption. However, one challenge we realized is how to effectively protect the shadow stack. One feasible approach on the x86 architecture is to use segmentation: the shadow stack is kept on an isolated segment from the segments used by the normal code and will be only accessible by instrumented instructions. Unfortunately, segmentation support is largely disabled on the x86_64 platform. Another possibility is to make the shadow stack read-only with the same memory lockdown technique (i.e., by wrapping the call/ret instruction with the code to dynamically switch the WP bit on and off). We have actually implemented this approach but our
experiments show that it incurs a performance penalty of more than 300%. This is rather disappointing. The reason is that the return addresses are frequently generated and the instructions to read and write CR0 (to change the WP bit) are more expensive than other regular instructions. After these failed attempts, we eventually ended up with the current scheme that achieves the desired security properties with a small performance overhead (Section 3.4).

To summarize, our second technique effectively protects the control data and allows for a stronger hypervisor control flow integrity guarantee from the original code integrity.

3.4 Evaluation

In this section, we first analytically examine the security guarantees provided by HyperSafe. Then we present our experiments with synthetic hypervisor exploits and measure the performance overhead.

3.4.1 Security Analysis

As mentioned earlier, HyperSafe implements two key techniques: non-bypassable memory lockdown and restricted pointer indexing. The first technique essentially guarantees the hypervisor code integrity and the second technique expands the protection to enforce control flow integrity. In the following, we systematically examine possible threats to these security guarantees.

To subvert the hypervisor’s integrity, an attacker’s main goal is to modify existing hypervisor code or introduce and execute its own attack code in the hypervisor space. Since the modifiability of existing hypervisor code and executability of introduced attack code are governed by the hypervisor page tables, the attacker needs to first subvert the page tables. Due to non-bypassable memory lockdown in HyperSafe, these page tables are write-protected with the WP bit on. In the following, we examine two possible attacks to subvert the protected page tables.

Disabling the WP bit The first attack aims to turn off the WP bit. Since the attackers are not yet able to inject and execute their own attack code, they must misuse existing hypervisor code. To accommodate benign page table updates, HyperSafe does introduce additional code to temporarily turn off the WP bit. Specifically, to legitimately update a page table, HyperSafe uses an atomic function that disables interrupts and turns off the WP protection before updating the page table. Immediately after the update, the WP protection is turned back on again. With that, in order to disable the WP bit, the attacker must divert the normal execution flow (i.e., by hijacking control data) before the WP bit is turned on again. Based on the control data protection by the second key technique of HyperSafe, such a diversion attempt is effectively defeated.

Another possible method is to compromise a previously saved runtime context. For example, the hypervisor may save important control registers (such as CR0 with the WP bit and CR3 as the page table base address) to writable memory and later restore them. The attacker could potentially gain full
control of the execution if these saved states can be tampered with. In our prototype, we ensure the correctness of these states before restoring them, i.e., their values are not changed after being initialized.

**Subverting the page tables** Alternatively, the attackers could subvert the page tables. As before, due to their inability to directly execute their own code, the attackers have to misuse existing code. For that, the attackers may attempt to introduce a double mapping (Section 3.2) to bypass the protection. Specifically, they can change or provide malicious parameters to those normal routines that handle benign page table updates. Based on our current adversary model, this attack is possible (even if we can faithfully enforce control flow integrity) due to the presence of exploitable software bugs. Fortunately, our memory lockdown technique enforces a simple invariant by disallowing double mapping and any changes to the permission bits (e.g., $R/W$ and $NX$) associated with the hypervisor code and data. A more subtle attack is that the attacker might map the hypervisor’s memory to a compromised guest VM (another variant of double mapping) by manipulating shadow page table entries. HyperSafe effectively blocks this by write-protecting the shadow page tables and preventing it from happening.

Also, instead of focusing on subverting page tables, the attacker may simply misuse existing code for malicious computations — as demonstrated by recent return-oriented programming [40, 82]. We point out the recently-surfaced return-oriented attacks are the very reason behind the expanded HyperSafe protection from the code integrity to the control flow integrity. Specifically, with restricted pointer indexing, HyperSafe protects the control data and ensures their uses will always adhere to the control flow graph that is pre-computed a priori. It may be argued that a `ret` may be manipulated to return to another return address that is contained in its target table but not in the last call site. However, such an attack is seriously limited in its scope and capability due to the need to follow the pre-determined control flow graph. Also, recent efforts (e.g., WIT [8]) can be naturally integrated for improved precision and protection coverage.

To the best of our knowledge, HyperSafe is the first system that is able to provide hypervisor control flow integrity. This guarantee is achieved by creating an unbreakable deadlock for the attackers. Specifically, the deadlock is centered on the need to subvert the page table for the attackers: On the one hand, to manipulate the page table, the attackers need to execute a turn-off-WP instruction injected or misused out of the normal control flow. On the other hand, to execute the turn-off-WP instruction injected or misused out of the normal control flow, they need to hijack the execution and tamper with write-protected code or control flow data, which in turn requires manipulating the page table. By integrating recent TPM-assisted static/dynamic root of trust [43, 93] that establishes load-time integrity, HyperSafe effectively enables non-subvertable enforcement for lifetime integrity.

**3.4.2 Synthetic Experiments**

To further validate the HyperSafe’s design, we empirically evaluated its effectiveness against several powerful synthetic attacks. Specifically, we deliberately introduced a hypercall interface with various
buffer overflow vulnerabilities and ported the Wilander’s buffer overflow benchmark test-suite [106] for a number of realistic attack scenarios. By exploiting these vulnerabilities, the attacker can write to arbitrary memory locations with any value of choice. In our experiments, we have conducted four different types of attacks: the first one modifies the hypervisor code, the second one executes from the injected code, the third one modifies the page table, and the fourth one tampers with a return pointer. Note these attacks mimic the key techniques of the real world attacks against hypervisors as shown in the National Vulnerability Database [63]. Our experiments show that HyperSafe successfully prevented all of them.

More specifically, in the first experiment, we tried to overwrite one of the hypervisor’s instructions with the instruction to reload the CR0 register so that the WP bit can be turned off. The write operation immediately triggered a page fault exception with the error code 0x03. This error code indicates that the fault was caused by an illegal memory write and the register CR2 contains the address of the faulting memory write. In the second experiment, we spilled a sequence of code (that turns off the WP protection) to a global array in the heap and the exploit triggered the execution of the spilled code. This attack is successfully foiled by the HyperSafe’s NX protection as the execution attempt leads to a page fault exception with an error code 0x11. This error code reports that the page fault was caused by an NX violation. In the third experiment, we targeted the page table by attempting to make the previously mentioned array executable. We point out this attack challenges the HyperSafe’s key technique – non-bypassable memory lockdown – and will be successful in a hypervisor if not protected by HyperSafe. Fortunately, with HyperSafe, the hypervisor’s page tables are write-protected, and the page table update attempt triggered a page fault with the error code 0x03. As compared to the first experiment, the faulting address (contained in CR2) in this case now pointed to one of the page table entries. Lastly, in our fourth experiment, we attempted to alter the hypervisor’s control flow by modifying a return pointer on the stack. Interestingly, HyperSafe silently defeated and recovered from the attack. A further investigation showed that the attacked return pointer belongs to a function, which is called only from one location. Recall the optimization that avoids the unnecessary memory reads for performance (Section 3.3.2), the instrumented code essentially ignores the return pointer on the stack and uses a direct jmp instruction to return to its (single) caller. When the optimization is not applied, the modified return pointer (or more precisely pointer index) will return back to either the original caller or error. In either case, this attack is foiled.

Also, we point out that once an attack is detected, we can easily combine the knowledge of the page fault’s error code, the faulting address, and the hypervisor’s memory layout to infer the nature of captured attacks. For example, if the faulting address CR2 points to an entry in a page table and the error code is 0x03, we can tell that the attacker intended to manipulate the hypervisor’s page tables. Based on the nature of the captured attack, we can then determine the most appropriate response. In our prototype, we simply issue an alert message, dump the machine context, and recover the execution if possible by ignoring the page faults.
Table 3.1: Software packages for the evaluation of HyperSafe

<table>
<thead>
<tr>
<th>Item</th>
<th>Version</th>
<th>Configuration/command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu Desktop</td>
<td>9.10-AMD64</td>
<td>standard installation</td>
</tr>
<tr>
<td>Clang/LLVM</td>
<td>2.6 pre-release2</td>
<td>default configuration</td>
</tr>
<tr>
<td>LMbench</td>
<td>3.0-a9</td>
<td>make results see</td>
</tr>
<tr>
<td>UnixBench</td>
<td>4.10</td>
<td>./Run</td>
</tr>
<tr>
<td>Kernel Build</td>
<td>2.6.31.4 (59MB)</td>
<td>make allnoconfig &amp;&amp; make</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.0.5</td>
<td>tar -jxf &lt;kernel file&gt;</td>
</tr>
<tr>
<td>Apache Server</td>
<td>2.2.12</td>
<td>Ubuntu package</td>
</tr>
<tr>
<td>ApacheBench</td>
<td>2.3</td>
<td>ab -c3 -t 60 &lt;url&gt;</td>
</tr>
</tbody>
</table>

3.4.3 Performance Evaluation

To evaluate the performance overhead introduced by HyperSafe, we measured the runtime overhead with standard benchmark programs including LMbench [57], UnixBench [96], ApacheBench [12], and two other real world applications. Our testing platform is a Dell Optiplex 755 desktop with a 3.0GHz Intel E8400 Core2Duo processor and 3GB memory. The machine runs a default installation of the Ubuntu 9.10 desktop with the official 2.6.31.14 kernel. Table 3.1 shows the configurations of our evaluation platform.

We tested the guest OS’s performance under BitVisor in two scenarios: with and without the HyperSafe protection. In order to further evaluate the impact from the improved precision, we tested two different implementations of HyperSafe: one has the least precise implementation with two big target tables (one for return instructions and one for indirect calls), and another has the most precise implementation with one target table for each function and one target table for each indirect call instruction. For simplicity, the former prototype is represented as HyperSafe-2, and the latter as HyperSafe-m. Note that the HyperSafe-m prototype includes the optimization mentioned in Section 3.3.2.

In our evaluation with the application benchmarks, we calculated the running time with the Linux time command and reported the system time plus user time. In the ApacheBench test, we run the Apache server inside the Ubuntu 9.10 desktop and the ApacheBench client on another Dell machine with the same hardware configuration to measure the web server’s throughput. These two machines were interconnected by a Gigabit Ethernet switch. All the test results reported here were the average of 10 runs. The deviations among these 10 runs are small (< 3%).

Application Benchmarks: We first performed application-level tests to measure HyperSafe’s impacts on real world programs. For that, we decompressed the official Linux 2.6.31.4 kernel source tarball, and then compiled the kernel. Conceptually, the kernel decompression is a computation-oriented task which will involve less hypervisor intervention, while the kernel compilation involves lots of device I/Os that will be intercepted by the hypervisor. As a result, we anticipate that the impact on the
kernel compilation is more significant than the impact on the kernel decompression. The third test is the standard ApacheBench program that measures the throughput of an Apache web server running inside the Ubuntu system.

Figure 3.6 shows the normalized performance results of HyperSafe-2 and HyperSafe-m when compared to the unmodified BitVisor. Overall, HyperSafe-m introduces less than 5% performance overhead. Interestingly, in all these tests, *HyperSafe-m outperforms HyperSafe-2!* This sounds counter-intuitive as HyperSafe-m achieves better precision than HyperSafe-2 likely at the cost of higher performance overhead. A further investigation indicates that the presence of multiple target tables and the optimization that avoids unnecessary memory reads when performing indirect control transfers both lead to improved performance. More specifically, the finer destination tables can improve cache utilization due to better locality. Also, the optimization in HyperSafe-m avoids the execution of additional memory-reading instructions.

**Micro-benchmarks:** We also used the standard micro-benchmark suites to evaluate HyperSafe’s impacts on various aspects of OS operations. Here, we focus on the context switch and memory operation overheads as they are known to cause most performance impacts [86]. In Table 3.2, we show the LMbench results. Specifically, the column *ctx* shows the latency of performing a context switch; the columns *stat* and *mmap* are the latency required to execute the corresponding system call; *sh proc* is the time spent to execute the C library function *system*; and *10K file* reports the time to create a 10KB file; and the column *Bcopy* shows the time to copy 1MB data using the C library function *bcopy*. For the UnixBench results, the final scores for BitVisor, HyperSafe-2, and HyperSafe-m, are 865.9, 844.5 (2.5%), and 849.6 (1.9%), respectively.

In general, BitVisor without the HyperSafe protection performs better than the two HyperSafe-based implementations. However, for two specific tests, i.e., the *mmap* and *sh proc* tests, HyperSafe-m actually performs better than BitVisor by a small margin, likely caused by the variations in our experiments.
Table 3.2: LMbench results for HyperSafe (in $\mu$s – smaller is better)

<table>
<thead>
<tr>
<th>VMM</th>
<th>ctx</th>
<th>stat</th>
<th>mmap</th>
<th>sh proc</th>
<th>10K file</th>
<th>Bcopy</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitVisor</td>
<td>31.0</td>
<td>0.86</td>
<td>5379</td>
<td>5976</td>
<td>28.8</td>
<td>1377</td>
</tr>
<tr>
<td>HyperSafe-2</td>
<td>32.7</td>
<td>0.87</td>
<td>5411</td>
<td>6181</td>
<td>30.1</td>
<td>1451</td>
</tr>
<tr>
<td>overhead</td>
<td>5.5%</td>
<td>1.2%</td>
<td>0.6%</td>
<td>3.4%</td>
<td>4.5%</td>
<td>5.4%</td>
</tr>
<tr>
<td>HyperSafe-m</td>
<td>31.2</td>
<td>0.86</td>
<td>5249</td>
<td>5844</td>
<td>29.8</td>
<td>1416</td>
</tr>
<tr>
<td>overhead</td>
<td>0.6%</td>
<td>0%</td>
<td>-2.4%</td>
<td>-2.2%</td>
<td>3.5%</td>
<td>2.8%</td>
</tr>
</tbody>
</table>

And consistent with the previous application-level tests, HyperSafe-m always beats HyperSafe-2 for the same reason as mentioned earlier: HyperSafe-m can achieve better cache utilization and avoid unnecessary memory reads in certain control flow transfers. In other words, HyperSafe-m is not only more precise than HyperSafe-2, but also runs faster. As a result, the benefits in achieving an enhanced security guarantee and improving system performance are worth the extra development and debugging efforts to implement the more precise HyperSafe-m.

In conclusion, HyperSafe is a lightweight hypervisor protection mechanism that incurs less than 5% performance overhead, thus satisfying our third design goal (Section 3.2).

3.5 Discussion

In this chapter, we have so far discussed the protection of Type-I bare-metal hypervisors by proposing two key techniques. In the following, we examine the possibility of porting them to support other Type-II hosted hypervisors or commodity OS kernels. Note a type-II hypervisor requires a hosted OS kernel. Therefore, similar challenges will be encountered to enable their support. In the following, we discuss the challenges we may face in the process and examine possible limitations in the current prototype.

At first glance, Type-I hypervisors and commodity OSes (including the Type-II hypervisors) are both system software that directly run on top of the bare-metal hardware and can be similarly supported. However, certain choices made in commodity OS design and implementation present additional challenges to achieve the intended security guarantees. For instance, the design of modern OSes such as Linux put much emphasis on the performance. Developers use all kinds of hacks to squeeze more performance from the hardware. And unfortunately, not all of these techniques are sound in security. This is especially true in the area of memory management. Issues such as double mapping and mixed pages are quite common in commodity OS kernels (e.g., the Linux kernel always doubly-maps the lower memory). To implement the similar memory lockdown technique in HyperSafe, we need to remove all the doubly-mapped and mixed pages. In this process, the presence of doubly-mapped pages will likely cause more difficulties as it will require to overhaul the way the kernel accesses its memory. And conceivably, it is a challenging task to ensure the re-designed memory management can still achieve comparable
performance while accommodating the much more frequent (benign) page table updates. Considering the need to escort page table updates and the associated overhead, the OS might consider implementing a batch mode for page table updates so that the cost can be amortized.

Additional challenges are also present to enforce the kernel’s control flow integrity, mainly due to the asynchronous nature of context switching and interrupt handling as well as the support of (potentially closed-source) third-party drivers. In particular, when a running process is being interrupted, the machine states are saved to memory so that they can be re-used later for resumption. Note these saved states can be potentially tampered with by the attacker to hijack the control flow. For that, there is a need to carefully examine all possible situations that may lead to states being saved to memory and ensure their correctness before being restored. Fortunately, most of these efforts are required only once. However, the support of loadable kernel modules, especially closed-source third-party drivers, remains a challenge. Specifically, for the enforcement of control flow integrity, our second key technique requires a precise alias analysis. How to improve its precision when handling a large-scale system software such as OS kernels as well as assembly instructions (or binary code) is still an ongoing research topic.

Also note that in its current form, our CFI enforcement is not as restrictive as possible because impossible paths [5] are still tolerated. For example, the indirect call site R1 in Figure 3.4 is allowed to transfer control to functions func_i and func_j. However, there may exist certain execution paths where only one of them is the valid target. Similarly, a return instruction may return to call sites other than its most recent caller. In Figure 3.4, the attacker may force function func_j to return to call site R2 by manipulating the return index on the stack, even that the function should return to R1. To address impossible paths, one possible way is to make our CFI enforcement context sensitive. For instance, the shadow stack provides a viable way to enforce strict control transfer for returns. Unfortunately, our implementation experience (Section 3.3.2) shows that performance overhead of write-protected shadow stack is high. However, despite these limitations, CFI still severely limits what attackers can achieve and is able to provide protection against a wide spectrum of attacks [5, 47].

For the very same reason, in our prototype, for some specific indirect function calls, we were forced to manually compute their call targets to handle the imprecision of the existing alias analysis tool we used. Manual analysis is tedious, time-consuming and error-prone. For the support of commodity OSes, there is a need to completely eliminate the need of manual involvement. To achieve that, we need to (1) scale the current field-and-context-sensitive unification-based alias analysis method and make it applicable for commodity OS kernels; and (2) enhance it for better precision by allowing for inclusion-based or flow-sensitive alias analysis and supporting assembly code. Note some promising progresses in this direction have been made by existing research efforts [16, 19, 23, 39] and the integration of these techniques remains an interesting direction for future work.
3.6 Summary

We have presented HyperSafe, a lightweight approach to provide lifetime control flow integrity for commodity Type-I hypervisors. HyperSafe achieves its goal by two key techniques: The first technique locks down write-protected memory pages and prevents them from being manipulated at runtime, thus effectively protecting the hypervisor’s code integrity; The second key technique converts the control data into pointer indexes by introducing one layer of indirection and thus expands protection to include control flow enforcement. A proof-of-concept system has been developed to protect two open-source Type-I hypervisors: BitVisor and Xen. Experimental results with a number of (synthetic) hypervisor attacks as well as benchmarking programs show HyperSafe can reliably provide the intended security guarantee with a small performance overhead.
Chapter 4

Isolating Type-II Hypervisors with HyperLock

4.1 Introduction

In the previous chapter we have presented HyperSafe, a system that can enable self-protection for type-I hypervisors. Due to the architectural differences, HyperSafe cannot be directly applied to protect a type-II hypervisor. In this chapter, we propose HyperLock, a system that can securely isolate a vulnerable or compromised type-II hypervisor.

Compared to bare-metal or Type-I hypervisors (e.g., Xen [14]) that run directly on the hardware, hosted hypervisors typically run within a conventional operating system (OS) and rely on this “host” OS to manage most system resources. By doing so, hosted hypervisors can immediately benefit from various key features of the host OS that are mature and stable, including abundant and timely hardware support, advanced memory management, efficient process scheduling, and so forth. Moreover, a hosted hypervisor extends the host OS non-intrusively as a loadable kernel module, which is arguably much easier to install and maintain than a bare-metal hypervisor. Due to these unique benefits, hosted hypervisors are increasingly being adopted in today’s virtualization systems [71].

From another perspective, despite recent advances in hardware virtualization (such as Intel VT [42]), virtualizing a computer system is still a complex task. For example, a commodity hosted hypervisor, such as KVM, typically involves a convoluted shadow paging mechanism to virtualize the guest memory (including emulating five different modes of operation in x86: paging disabled, paging with 2, 3, or 4 levels of page tables, and hardware assisted memory virtualization, including EPT/NPT [42]). For performance reasons, many hypervisors also support an “out-of-sync” (OOS) shadow paging scheme that synchronizes the shadow page table with the guest only when absolutely necessary. In addition, hosted hypervisors still suffer from a large attack surface, because they take many untrusted guest virtual machine (VM) states as input. For example, shadow paging needs to read guest page tables for
synchronization, while instruction emulation – another complicated component of a hypervisor – involves fetching guest instructions for interpretation and execution.

Due to inherent high complexity and broad attack surface, contemporary hosted hypervisors are not immune to serious security vulnerabilities. A recent study of the National Vulnerability Database (NVD) [63] indicates that there were 24 security vulnerabilities found in KVM and 49 in VMware Workstation over the last three years. These vulnerabilities can be potentially exploited to execute arbitrary code with the highest privilege, putting the whole host system at risk. In fact, successful attacks against both KVM [29] and VMware Workstation [50] have been publicly demonstrated to escape from a guest VM and directly attack the host OS. Worse, a compromised hypervisor can also easily take over all the other guests, leading to disruption of hosted services or stealing of sensitive information. This can have a devastating effect in the scenario where a single physical machine may host multiple VMs from different organizations (e.g., in a cloud setting). In light of the above threats, there is a pressing need to secure these hosted hypervisors and protect the host system – as well as the other guest VMs – from a compromised hypervisor.

To address the need, researchers have explored a number of approaches. For example, seL4 [49] takes a formal approach to verify that a small micro-kernel (\(~8.7K\) source lines of code or SLOC) is secure including the absence of certain software vulnerabilities (e.g., buffer overruns and NULL pointer references). However, it is not scalable or still incomplete in accommodating commodity hypervisors (e.g., KVM) that have a much larger code base and support various complex x86 CPU/chipset features [76]. HyperSafe presented in Chapter 3 enables self-protection for bare-metal hypervisors to enforce their control flow integrity. Unfortunately, as discussed earlier, the proposed approach cannot be applied for hosted hypervisors due to different design choices in the commodity host OS (e.g. frequent page table updates) from bare-metal hypervisors. Other approaches [87, 112] take a layer-below method to isolate untrusted device drivers, which is also not applicable because hosted hypervisors already run at the lowest level on the system.

In this chapter, we present HyperLock, a system that is able to establish a tight security boundary to isolate hosted hypervisors. Specifically, we encapsulate the execution of a hosted hypervisor with a secure hypervisor isolation runtime, which has its own address space and a reduced instruction set for safe execution. By doing so, the host system is not accessible to the hypervisor. Instead, it must go through a well-defined interface, which is mediated and narrowed-down by HyperLock to block any unexpected side-effects. Moreover, we further propose a hypervisor shadowing technique, which can efficiently create a separate shadow hypervisor for (and pair it with) each guest so that a compromised hypervisor can affect only the paired guest, not others. By exploiting recent memory de-duplication techniques, these shadow hypervisors can be created without incurring additional resource overhead.

We have implemented a proof-of-concept HyperLock prototype for the popular KVM hypervisor (version kvm-2.6.36.1 and qemu-0.14.0). Our experience shows that HyperLock can be implemented with a small code base (\(~4K\) SLOC). We demonstrate its effectiveness and practicality by perform-
ing additional security analysis and performance measurement. To summarize, this chapter makes the following contributions:

- To address the imperative need to confine commodity hosted hypervisors, we propose a secure *hypervisor isolation runtime* with a dedicated address space and a reduced instruction set to strictly confine their execution. To the best of our knowledge, the proposed hypervisor isolation runtime is among the first to isolate hosted hypervisors and protect the host OSs from being jeopardized by them.

- To effectively prevent a compromised hosted hypervisor from taking over all guests in the same physical machine, we propose another key technique, i.e., *hypervisor shadowing*, to create a guest-specific shadow hypervisor without additional resource overhead. By doing so, we ensure that a compromised hypervisor will only affect the corresponding guest, not others.

- We have developed a HyperLock prototype and used it to protect the popular KVM hypervisor. Our prototype introduces a very small TCB (∼4K SLOC) to the current OS kernel while completely removing the original KVM code (∼33.6K SLOC) as well as the companion QEMU program (∼531K SLOC) from the TCB of the host OS. The security analysis and evaluation with standard benchmark programs shows that our prototype is not only effective, but also lightweight (< 5% performance slowdown).

The rest of this chapter is structured as follows: we first present the design of HyperLock with a focus on the KVM hypervisor in Section 4.2. After that, we discuss the implementation and evaluation of HyperLock in Sections 4.3, and 4.4, respectively. Issues and possible improvements are discussed in Section 4.5. Finally, we conclude the chapter in Section 4.6.

### 4.2 Design

Before presenting our system design, we first briefly review the basic architecture of existing hosted hypervisors and the associated threat model. For simplicity, we use KVM as the representative example throughout the chapter. As a popular, open-source hypervisor, KVM is incredibly simple to deploy and run. It can be dynamically loaded as a kernel module on Linux and once loaded, it instantly extends the host OS with virtualization support (based on hardware virtualization extensions such as Intel VT [42]). KVM uses a companion user program, i.e., a QEMU variant, to cooperatively emulate hardware devices for a guest (e.g., hard disks). In Figure 4.1(a), we show the main execution flow of a KVM-powered guest, which involves close interaction between KVM and QEMU. For example, when QEMU issues an *ioctl* command (e.g. KVM_RUN – arrow 1) to KVM, KVM proceeds by switching into the guest mode for the VM execution (arrow 2), which means the guest code can run natively on the CPU. The
guest mode continues until certain events (e.g., an I/O port access – arrow 3) happen to cause a VM exit back to KVM. Based on the VM exit reason, KVM may directly resolve it or delegate it to QEMU (arrow 4). After the VM exit is resolved, the guest can now enter the guest mode again for native VM execution (arrow 2).

In this work, we assume an adversary model where an attacker can successfully subvert the underlying hosted hypervisor from a malicious guest. To do that, the attacker can choose to exploit vulnerabilities either in the hypervisor itself (e.g., memory corruption in KVM) or in its companion user-level program (e.g., vulnerabilities in device emulation). As part of the exploitation process, the attacker may attempt to execute arbitrary malicious code in the compromised hypervisor or the user-level program [20]. In other words, by assuming the presence of exploitable software vulnerabilities in hosted hypervisors, we aim to deal with the threats from an untrusted guest so that the hosted hypervisor, even compromised, cannot take over the host and other guests.

Figure 4.1(b) illustrates the high-level architecture of HyperLock. To isolate hosted hypervisors, it has two key components, i.e., a hypervisor isolation runtime and a hyperlock controller. With these two components, unlike the traditional case of directly running KVM in the host OS (Figure 4.1(a)), HyperLock confines the KVM execution in a secure hypervisor isolation runtime with its own separate address space, where only unambiguous instructions from a reduced instruction set will be allowed to execute. Further, one isolation runtime is bound to one particular guest. That is, every guest logically has its own separate copy of the hypervisor code and data. Therefore, guests are completely isolated from each other. To confine KVM’s user-level companion program, i.e., QEMU, HyperLock limits its system call interface and available resources with system call interposition. By doing so, HyperLock can mediate the runtime interaction between QEMU and KVM by acting as a proxy to forward commands from QEMU to KVM or relay requests from KVM on the opposite direction. The controller is also designed to provide runtime services to KVM as some tasks cannot be entrusted or delegated to KVM.
4.2.1 Hypervisor Isolation Runtime

Our first component – the hypervisor isolation runtime – is designed to safely isolate or confine the privileged hosted hypervisor so that even it is compromised, our host can still be protected. Isolation is achieved through two main mechanisms: memory access control and instruction access control. In the following, we describe each mechanism in detail.

Memory Access Control

To confine the privileged KVM module, we create a separate address space based on the CPU paging mechanism. Before permitting KVM to run, HyperLock switches to the KVM-specific address space by loading the CR3 register with the corresponding page table base address. The KVM-specific page table is maintained by HyperLock and cannot be changed by KVM because we map it read-only inside the KVM address space to facilitate the guest page table update (Section 4.3.3). We stress that it is critical to make the KVM page table unmodifiable to KVM. Otherwise, a compromised KVM can take advantage of it to access or modify the host OS memory and corrupt the whole system. In our design, we further enforce $W \oplus X$ [4] in the KVM address space. That is, there is no memory page in the isolated KVM address space that is simultaneously executable and writable. Also, as there is no legitimate need for KVM to execute any of the guest code, the whole guest memory is marked as non-executable in the KVM address space. In fact, within this address space, only the KVM module contains the executable code (one exception is the trampoline code we introduced to switch the address space from KVM back to the host – Section 4.2.1). By doing so, HyperLock guarantees that no code inside the isolation runtime can alter its memory layout or change the memory protection settings.

Within the isolation runtime, KVM can directly read from or write to the guest memory as usual. For performance reasons, it is important to allow KVM access to the guest memory. Specifically, if an I/O instruction executed by the guest is trapped and emulated by KVM, it requires several guest memory accesses for I/O emulation: It has to first traverse the guest page table to convert the guest virtual address to its corresponding guest physical address (which can further be converted to a host address usable by KVM); After that, KVM can then read or write the guest memory again to actually emulate the I/O operation. Because physical memory for the guest is linearly mapped inside the isolation runtime (starting at address zero), guest physical addresses can be directly used by KVM to access guest memory without further conversion.

There also exists a trampoline code inside the isolation runtime to switch the context back to the host, which will be discussed in the second isolation mechanism (Section 4.2.1).
**Instruction Access Control**

In addition to a separate address space for the hosted KVM hypervisor, we further restrict the instructions that will be allowed to execute within it. This is possible because KVM does not contain any dynamic code, and our memory access control removes the possibility of introducing any new additional code in the isolation runtime. However, challenges arise from the need of executing privileged instructions (of hardware virtualization extension) in the KVM module. For example, KVM needs to execute `VMWRITE`, a privileged instruction that updates the VM control structure (or VMCS [42]). Though we could potentially replace these `VMWRITE` instructions with functions to enlist help from HyperLock, the performance overhand could be prohibitively high due to context switching between the isolation runtime and the host kernel.

Existing hardware does not allow granting privileges to individual instructions. Therefore, while still running the KVM code at the highest privilege, there is a need to prevent this privilege from being misused. Specifically, our instruction access control scheme guarantees that **no privileged instructions other than explicitly-allowed ones can be executed within the isolation runtime**. In our prototype, we permit only two privileged instructions, i.e., `VMREAD` and `VMWRITE`, for direct execution while re-writing other privileged instructions to rely on the trusted supporting routines in HyperLock (Section 4.3.3). Note that these two permitted privileged instructions could be executed frequently (e.g., tens of times per VM exit) and it is thus critical to execute them directly to avoid unnecessary context-switching overhead.

Moreover, to avoid the highest privilege from being abused, we need to prune the KVM instructions to remove any other “unexpected” privileged ones. Specifically, due to x86’s variable length instruction set, it is still possible to uncover “new” or unintended instructions, including privileged ones (e.g., by fetching or interpreting the same memory stream from different offsets [20]). To remove these unintended instructions, we enforce the same instruction alignment rules as in PittSFteld [59] and Native Client (NaCl) [113] to allow for unambiguous, reliable disassembly of KVM instructions. Specifically, in our prototype, the KVM code is organized and instrumented into equal-length fragments (32 bytes). As a result, no instruction can overlap the fragment boundary. Also, computed (or indirect) control transfers are instrumented so that they can only transfer to fragment boundaries. These two properties ensure that all the instructions that are executable inside the isolation runtime are known at compile time [113]. With that, we can then scan the instrumented code to verify that KVM can only contain the two explicitly-allowed privileged instructions, and not any other privileged ones.

In addition to effectively restricting the instructions allowed to execute within the isolation runtime, our scheme also provides a way to safely return back to the host kernel. This is needed as KVM is now strictly confined in its own address space and will enlist HyperLock for the tasks that cannot be delegated to itself. To achieve that, we design a trampoline that will safely load the CR3 register with the host kernel page table base address. For isolation purposes, the trampoline code also needs to switch a number of critical machine registers, including x86 segment descriptor table (GDT/LDT), interrupt
Figure 4.2: Trampoline code layout: each code fragment starts with a one-byte INT3 and ends with a short JMP, which skips over the next INT3 of the following code fragment. The starting address of the trampoline code is loaded into an IDT entry as the interrupt handler.

descriptor table (IDT), and task state segment (TSS) [42], which means that a number of critical state registers need to be accessible to KVM. Fortunately, from the trampoline’s perspective, these registers are static across the context switches. Therefore, we simply collect them in a separate memory page, mark it read-only to KVM, and make it available to the trampoline code. In other words, as long as we stay inside the isolation runtime, this critical state becomes write-protected.

Because critical hardware state is updated by the trampoline code, we take a step further by ensuring the atomicity of its execution, thus preventing the partial loading of hardware state. Specifically, we ensure that the trampoline code can be entered only from a single entry point inside the isolation runtime, and its execution cannot be interrupted. In our prototype, KVM has to issue a software interrupt (using the INT instruction) to execute the trampoline code and exit the isolation runtime. Hardware interrupts are automatically disabled by the hardware to run the interrupt handler (i.e., the trampoline code) and will not be enabled until it has returned to the host. The handler for this software interrupt is the entry point to the trampoline code. To further make sure it is the only entry point, we need to foil any attempts that jump to the middle of the trampoline code. In our design, we arrange the trampoline akin to the service runtime call in NaCl for this purpose (Figure 4.2). Specifically, we put a single byte INT3 instruction at the beginning of each code fragment in the trampoline. If executed, this INT3 instruction will immediately cause a debug exception. As mentioned earlier, HyperLock enforces instruction alignment rules for any code running inside KVM. This guarantees that indirect control flow transfers (that may be controlled by the attacker) can only jump to code fragment boundaries. By putting an INT3 instruction at these locations, HyperLock can immediately catch any attempts to subvert the trampoline code (since there is no legitimate code in the original KVM to call the trampoline). On the other hand, legitimate invocation of the trampoline code will not be interrupted by INT3 because a short jump is placed at the end of each code fragment to skip over them. As such, we can effectively ensure a single entry to the trampoline code and the atomicity of its execution.

4.2.2 HyperLock Controller

Our second component is designed to accomplish three tasks complementary to the first one. Specifically, the first task is to achieve complete guest isolation by duplicating a KVM hypervisor (running
inside an isolation runtime) for each guest. Traditionally, a compromised KVM hypervisor immediately brings down all the running guests. By duplicating the hypervisor for each guest and blocking inter-hypervisor communication, we can ensure that a compromised KVM can only take over one guest, not all of them. However, instead of simply duplicating all the hypervisor code and data, which unnecessarily increases memory footprint of our system, we propose a hypervisor shadowing technique by assigning each guest a shadow copy. The shadow copy is virtually duplicated to segregate the hypervisor instances; there is only a single physical copy. This is possible because all the shadow copies share identical (static) hypervisor code, which means we can apply classic copy-on-write or recent memory de-duplication techniques [2] to maintain a single physical copy, thus avoiding additional memory consumption overhead. Each shadow copy still runs within a hypervisor isolation runtime and can legitimately access the memory space of one and only one guest within its own address space.

The second task is to act as a proxy connecting QEMU and the isolated KVM. On one hand (arrow 2 in Figure 4.1(b)), it accepts ioctl commands from QEMU (e.g. CREATE_VM, CREATE_VCPU, and KVM_RUN) and passes them to KVM via remote procedure calls (RPCs). Our system maintains the same ioctl interface and thus supports the same companion QEMU program without any modification. On the other hand (arrow 3 in Figure 4.1(b)), HyperLock provides runtime services for tasks that either require interaction with the host OS (e.g. to allocate memory for the guest), or that cannot be safely entrusted to KVM (e.g. to update shadow page tables). Because HyperLock relies on the host OS to implement these runtime services, it is critical to understand the possible impact should KVM be compromised or these services be misused. To proactively mitigate these consequences, our prototype defines a narrow interface that exposes only five well-defined services, which are sufficient to support commodity OSs (including both Linux and Windows XP) as VMs. These five services include (1) map_gfn_to_pfn to convert a guest physical page number (gfn) to the physical page number (PFN) of its backing memory, (2) update_spt/npt to batch-update the shadow page table (spt) or the nested page table (npt), (3) read_msr to read x86 machine-specific registers (MSRs), (4) write_msr to write MSRs, and (5) enter_guest to switch the guest execution into guest mode. In our prototype, we scrutinize possible arguments to these services and block any unexpected values. Furthermore, resources allocated by HyperLock on behalf of each guest will be accounted to that guest to foil any attempts to deplete or misuse resources.

The third task is to reduce the exposed system call interface to the user-level companion program, i.e., QEMU. Specifically, we manually obtain the list of system calls that will be used in QEMU and then define a stand-alone system call table for it. This system call table is populated with only those allowed entries to prevent QEMU from being abused. In addition, we also limit the allowed parameters for each system call and deny anomalous ones. As this technique has been well studied [36, 69], we omit the details here.
4.3 Implementation

We have implemented a HyperLock prototype to isolate the KVM hypervisor (version 2.6.36.1 with ∼33.6K SLOC) and QEMU (version 0.14.0 with > 531K SLOC). Our prototype runs on Linux/x86 and has ∼ 4.1K SLOC. Specifically, our prototype contains 862 lines of C code for the hypervisor isolation runtime and 270 lines of assembly code for the trampoline that manages the context switches between the host and KVM. The five runtime services take 569 SLOC. The remaining code (∼ 2.3K SLOC) is primarily helper routines to manage the host state, confine QEMU, and support its interaction with KVM. Our current prototype is implemented and evaluated based on a Dell machine (with an Intel Core i7 920 CPU and 3GB memory) running Ubuntu 10.04 LTS and a Linux 2.6.32.31 kernel. In the rest of this section, we present details about our prototype based on the Intel VT [42] hardware virtualization extension. Note that our prototype is implemented on the 32-bit x86 architecture.

As we will explain in this chapter, new features of the 64-bit x86 architecture (e.g., the interrupt stack table) actually make the implementation less challenging than on the 32-bit architecture.

4.3.1 Memory Access Control

HyperLock confines the KVM memory access by creating a separate paging-based address space. Within this address space, there are three components: KVM itself, guest memory, and the trampoline code for host and KVM context switches. Among these three components, the memory layout for KVM and our trampoline code do not change after initialization, while the page table entries (PTEs) for guest memory have to be updated on demand (because the guest memory layout and mapping might be changed frequently when the guest is running). To set up these PTEs, KVM needs to notify HyperLock (via the map_gfn_to_pfn service), which then checks whether a page of memory can be successfully allocated for the guest. If it can, HyperLock fills in the corresponding PTE. Otherwise, it returns failure back to KVM. Notice that from the host OS’s perspective, the guest is just a normal process, i.e., the QEMU process. Therefore, the guest memory may be swapped out or in by the host OS when under certain memory pressure. To accommodate that, HyperLock needs to synchronize the KVM page table when such events happen. In our prototype, we register an MMU notifier [27] with the host kernel in order to receive notifications of these events. Upon every notification, our prototype will update the affected page table entries in the notification handler and further forward these events to KVM so that KVM can synchronize the SPT/NPT for the guest.

HyperLock’s paging-based memory access control is relatively straightforward to implement. However, there is one subtlety related to TLB (translation lookaside buffer) in the x86 paging mechanism. To illustrate, TLB is known as a fast cache of virtual to physical address mapping; if a mapping is already cached in TLB, CPU directly returns the mapping without bothering to traverse page tables again to translate it. Also, reloading the CR3 register flushes all TLB entries except those for global pages [42], which are being used by Linux to retain TLB entries for kernel memory during the task switching.
However, global pages could lead to serious security vulnerabilities in HyperLock. More specifically, because of these global pages, memory mappings for the host kernel will remain in the TLB cache even after switching to the isolation runtime for KVM, which means KVM can exploit the stale cache to access the mapped host OS kernel memory or instructions. As such, the host kernel’s memory would be exposed to the untrusted KVM code. In our prototype, we had to disable the global page support in CPU by clearing the PGE bit in the CR4 register before entering the isolation runtime for KVM execution. By doing so, we can ensure that CPU flushes all TLB entries when switching to the KVM address space, thus making host kernel pages inaccessible to KVM. Although disabling the global page support leads to more frequent TLB reloading for kernel memory, the Linux kernel’s use of large pages (2MB) for kernel memory relieves some performance overhead. In our prototype, we also considered using the VPID (virtual-processor identifier) feature [42] of Intel-VT. However, we found that the feature cannot be used to avoid disabling global page support, because it is always set to zero in the non-guest mode.

### 4.3.2 Instruction Access Control

In addition to memory access control, HyperLock also confines the available instructions inside the isolation runtime. Specifically, we first enforce instruction alignment [59, 113] on the KVM and our trampoline code by compiling them through the Native Client (NaCl) compiler, a customized gcc compiler developed by Google. With the help of instruction alignment, we can then reliably disassemble available code inside the isolation runtime with the assurance of no unintended instructions. As mentioned earlier, our prototype blocks all privileged instructions that can be executed inside the isolation runtime, except VMREAD and VMWRITE for performance reasons. To remove disallowed instructions, we further create a small script to scan the (reliably-disassembled) instructions of KVM and replace every privileged instruction (except VMREAD and VMWRITE) with a call to the corresponding runtime service.

Based on Intel VT, each guest is associated with a VMCS memory page that contains 148 fields to control the behavior of both the host and the guest. These fields can be roughly divided into four categories: host state, VM execution control, guest state, and VM exit info. Generally speaking, the first two categories need to be handled by trusted code because they can critically affect the host behavior. For example, HOST_RIP specifies the instruction CPU will return to after a VM exit; and EPT_POINTER stores the address of EPT/NPT table for the guest. In our prototype, we directly handle them outside the isolation runtime. Our development experience indicates that KVM handles these VMCS fields in a rather simple way: Most of these fields involve just loading the host state directly into its corresponding VMCS field, and will never change after the initial setup. Fields belong to the latter two categories can be safely delegated to untrusted KVM since they reflect the guest VM’s state. For example, VM_EXIT_REASON gives the reason that caused the VM exit; and GUEST_CS_SELECTOR contains the current CS segment selector for the guest. Unlike host state and VM execution control fields, these fields are
movl $0xc00195d7, %eax
movl $GUEST_EIP, %edx
vmwrite %eax, %edx

Figure 4.3: A VMWRITE macro-instruction that writes 0xc00195d7 into the GUEST_EIP VMCS field.

frequently retrieved and updated by KVM during each VM exit. For performance reasons, we would like to grant KVM direct access to guest state and VM exit information while preventing it from touching any other fields relating to host state or VM execution control. That is also the reason why our prototype makes an exception for the VMREAD and VMWRITE instructions.

To avoid these two instructions from being misused, our prototype takes the following precautions: First, we prevent KVM from directly accessing VMCS memory. Specifically, Intel VT requires that physical address of VMCS must be loaded to CPU before software can access its fields with VMREAD and VMWRITE. However, nothing prevents attackers from directly overwriting its fields if the VMCS is virtually mapped in the KVM address space. As such, HyperLock allocates VMCS for the guest outside of the isolation runtime, and loads its physical address to CPU before entering the KVM. The VMCS structure itself is not mapped inside the KVM address space, therefore attackers cannot manipulate its fields by directly changing the VMCS. Meanwhile, the CPU has no problem executing the VMREAD and VMWRITE instructions because it uses a physical address to access the VMCS. Second, both the VMREAD and VMWRITE instructions take a VMCS field index as a parameter. Our prototype ensures that only fields belonging to guest state and VM exit information can be passed to them. More specifically, we define two macro-instructions (similar to nacljmp in NaCl [113]) for VMREAD and VMWRITE as shown in Figure 4.3. Each macro-instruction first fetches the hard-coded field index from KVM’s code section (which is read-only, because it is protected by $W \oplus X$) into a register, then passes the register directly to VMREAD or VMWRITE. Further, we verify that the macro-instructions (each 17 bytes long) cannot overlap a fragment boundary (32 bytes) to block attackers from jumping into the middle of macro-instructions. There is a subtlety here: if an attacker is able to interrupt the CPU right before a VMWRITE, he might maliciously modify the register content saved by the interrupt handler. When the interrupt handler returns, registers are restored and the malicious field index gets used by VMWRITE. HyperLock avoids this problem because the trampoline handles the interrupt context, so it is not accessible to KVM. Finally, our script to scan KVM’s assembly code also makes sure that only fields pertaining to guest state and VM exit information can be passed to these two macro-instructions.

The trampoline code for host and KVM context switches is also worth mentioning. To prevent KVM from monopolizing the CPU and to ensure a timely response to hardware interrupts, we need to enable interrupt delivery while KVM is running. Specifically, the trampoline code contains a handler for each
exception or interrupt. The handler for a hardware interrupt first switches to the host OS and redirects control to the host OS’s corresponding interrupt handler (defined in the host IDT table). Execution of the KVM will resume after host interrupt handler returns. The handler for an exception, which is caused by error conditions in the KVM, instead switches to the host OS and then immediately terminates the VM after dumping the KVM’s state for auditing and debugging purposes. Under normal conditions, KVM should never cause exceptions, in particular, page faults: updates to the guest memory mapping by the host OS (e.g., paging out a block of memory) are synchronized to KVM through an MMU notifier [27]. When the need arises for KVM to access guest memory, it proactively calls the map.gfn.to.pfn service to read in the page, thus avoiding page faults.

At first glance, enabling interrupt delivery while KVM runs may only require setting up the IDT (Interrupt Descriptor Table). However, one quirk of the x86 architecture makes this more complicated than it should be: when an interrupt happens, the CPU will save the current state (such as EIP, ESP, and EFLAGS) to the stack so that it can resume the execution of the interrupted task. Because the sandbox is running at the highest privilege (ring 0), this state is saved to the current stack, which could then be manipulated by the attacker to launch a denial of service attack on the 32-bit x86 platform. For example, a double fault will be triggered if the attacker manages to set the stack pointer ESP to an invalid (unmapped or write-protected) memory address and then write to the stack. The first write to the stack will cause a page fault. To handle the page fault, CPU tries to push more content to the invalid stack, which will lead to a second page fault. This time, CPU throws a double fault instead of more page faults. However, the stack pointer remains invalid for the double fault handler. Eventually, the CPU can only be recovered by power-cycling the machine. An astute reader may point out that we can switch to an interrupt task (thus a known-good stack) through task gate for an interrupt handler, and it has been used by Linux to handle double faults. Unfortunately, task gate cannot be securely deployed inside the isolation runtime where untrusted code runs. Specifically, to switch to an interrupt task, CPU uses a data structure called the TSS (task state segment) descriptor that specifies where to load CPU state information from, including CR3, EIP and ESP. Therefore, it is critical to write-protect the TSS descriptor. However, this structure cannot be write-protected in this case because CPU needs to change the descriptor’s B (busy) bit from zero to one before switching to the interrupt task. Write-protecting the TSS descriptor will lead to another undesirable situation where the CPU can only be recovered by a hardware reset.

To accommodate that in 32-bit x86 architecture, our system always keeps a valid ESP to foil such attacks. Specifically, we allocate three continuous memory pages (12KB total) at a fixed location in the KVM address space: the middle page is used for the stack itself (the same size as the stack in recent Linux kernels), while the top and bottom pages are used as overflow space. At the compiling time, we instrument the instructions that change ESP to maintain the stack location invariant by replacing the page number part (top 20 bits) of ESP to that of the pre-allocated stack (the middle page). This can be implemented with an AND instruction (6 bytes) and an OR instruction (6 bytes). Both of them take a
Table 4.1: The breakdown of HyperLock’s runtime service implementation. read_msr and write_msr are implemented together. enter_guest includes 53 lines of inline assembly code and 25 lines of C code.

<table>
<thead>
<tr>
<th>Runtime Service</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>map_gfn_to_pfn</td>
<td>84</td>
</tr>
<tr>
<td>update_spt/npt</td>
<td>251</td>
</tr>
<tr>
<td>read_msr, write_msr</td>
<td>156</td>
</tr>
<tr>
<td>enter_guest</td>
<td>78</td>
</tr>
</tbody>
</table>

4-byte constant and ESP as operands. As such, no scratch registers are required for this instrumentation. Moreover, to prevent the check from being circumvented, the check and its related instruction are kept in the same fragment (A similar inline software guard to prevent stack overflow was also explored in XFI [31].) HyperLock support for 64-bit x86 architecture will not suffer from the same issue because the CPU can be programmed to always switch to a known-good stack for interrupt handling with the help of a new feature called IST (interrupt stack table) [42].

4.3.3 Others

To properly isolate the KVM hypervisor, HyperLock also exposes a narrow interface to five well-defined runtime services to KVM. All these five services were implemented in a small number of lines of source code (Table 4.1). Among them, update_spt/npt is the most involved as it is directly related to the memory virtualization in KVM. For a concrete example, we use hardware assisted memory virtualization (NPT) to describe how update_spt/npt is implemented.

With NPT support, the CPU uses two page tables to translate a guest virtual address to the corresponding physical address: a guest page table (GPT) to convert a guest virtual address to a guest physical address, and a nested page table (NPT) to further convert a guest physical address to the (actual) physical address. The guest kernel has full control over GPT, while KVM is responsible for maintaining the NPT. Since NPT maps physical memory into the guest, only the trusted NPT should be loaded to CPU for guest address translation. In HyperLock, KVM still maintains its own NPT for the guest. However, this NPT is not used for address translation. Instead, HyperLock creates a mirror of the NPT outside the hypervisor isolation runtime to translate guest addresses. The NPT table and its mirror are synchronized via update_npt calls. In update_npt, HyperLock ensures that only physical pages belonging to this guest will be mapped to the guest. Noticing that the guest memory is mapped in the hypervisor isolation runtime with the (HyperLock-maintained) KVM page table, this guarantee can be efficiently achieved using the KVM page table. More specifically, KVM provides a guest physical page number and its memory protection attributes as parameters to update_npt. In this function, HyperLock traverses the KVM page table to find the physical page for this guest physical page and combine it with the
memory protection attributes to update the corresponding page table entry in the NPT mirror. Moreover, the KVM page table is made available to KVM (by mapping it read-only in the KVM address space) so that KVM can use it to maintain its own NPT in the same way.

Overall, to isolate KVM with the proposed hypervisor isolation runtime, our prototype re-organizes KVM in a slightly different way. However, our modification to KVM is minor and focuses on three areas. First, the original ioctl based communication interface between KVM and QEMU is replaced by our RPC calls through HyperLock, which results in changing six ioctl functions in KVM (e.g., kvm-dev.ioctl, kvm_vm_ioctl, etc.). Second, we replace certain dangerous KVM calls with RPCs to runtime services, which results in changing eight functions in KVM. As an example, the original KVM’s function (set_spte) that writes directly to shadow page table entries is replaced by the update_spt service. Third, we also need to reduce one file, i.e., vmx.c, to avoid changing host state and VM execution control fields of VMCS in KVM. Instead, functions that access these two categories are moved to HyperLock while the rest stays the same. Our experience shows these changes (1) are mainly one-time effort as they essentially abstract the underlying interaction with hardware, which remains stable over the time, and (2) do not involve the bulk KVM code, which could undergo significant changes in future releases.

4.4 Evaluation

In this section, we present our evaluation results by first analyzing the security guarantees provided by HyperLock. After that, we report the performance overhead with several standard benchmarks.

4.4.1 Security Analysis

Based on our threat model (Section 4.2), an attacker starts from a compromised guest and aims to escape from HyperLock’s isolation and further take over the host OS or control other guests (by exploiting vulnerabilities in KVM or QEMU). In HyperLock, we create a separate system call table for the QEMU process to constrain system calls available to it and validate their parameters. The security guarantee provided by such a system call interposition mechanism has been well studied [36, 69, 105]. In the following, we focus our analysis on the threats from a (compromised) KVM hypervisor when it aims to break out of HyperLock confinement.

Breaking Memory Access Control The first set of attacks aims to subvert memory protection in the isolation runtime to inject malicious code or modify important data structures, especially (read-only) control data in the trampoline. Since $W \oplus X$ is enforced in the isolation runtime, any attempt to directly overwrite their memory will immediately trigger a page fault and further cause the guest to be terminated by HyperLock. Having failed direct memory manipulation, the attacker may try to disable $W \oplus X$ protection by altering the KVM page table. Since the KVM page table is not directly changeable in the isolation runtime, the attacker has to leverage the trusted HyperLock code to manipulate the KVM...
page table. Fortunately, HyperLock will never change memory attributes for the (static) trampoline and KVM after initial setup, and the whole guest memory is marked as non-executable. Another possibility is for the attacker to trick HyperLock to map the host or HyperLock memory (e.g. host page table) into the isolation runtime or a malicious guest as (writable) guest memory. Sanity checks in the map_gfn_to_pfn and update_spt/npt service would prevent this from happening.

**Subverting Instruction Access Control**  Since the hypervisor isolation runtime has the highest privilege, it is critical to prevent attackers from executing arbitrary privileged instructions. With the protection of $W \oplus X$ and instruction alignment, the attackers cannot inject code or uncover “new” instructions based on legitimate ones. Instead, they would have to target existing legitimate privileged instructions in the isolation runtime, for example, to maliciously modify host state or VM execution control fields in the VMCS by exploiting the field index parameter of the `VMWRITE` instruction (Figure 4.3). Notice that hard coding the field index (`GUEST_EIP` in Figure 4.3) and instruction alignment alone can _not_ prevent misuse of the `VMWRITE` instruction. This is because that the second (fetching the field index into a register) and third instructions (executing `VMWRITE`) can be separated by an instruction fragment boundary. In other words, `VMWRITE` is the first instruction in an instruction fragment. With the capability to jump to any instruction fragment boundary under the instruction alignment rule, the attacker can directly jump to the `VMWRITE` instruction after loading the `edx` register with a malicious field index. This attack is prevented in HyperLock by ensuring that the three-instruction sequence (17 bytes) in Figure 4.3 cannot overlap any fragment boundary, thus ensuring `VMWRITE` and `VMREAD` always receive the fixed known-good field index parameters.

Another source of legitimate privileged instruction is the trampoline code (to load `CR3` etc). Similar to the `VMWRITE` instruction, it is necessary for HyperLock to prevent the attacker from jumping to the middle of the trampoline code. Unfortunately, the trampoline code (about 4K bytes) cannot fit in a single instruction fragment. In HyperLock, we set up a one-byte `INT3` instruction at each fragment boundary of the trampoline code to capture direct jumps to the middle of the trampoline. Moreover, the execution of the trampoline cannot be disrupted by KVM because interrupts are disabled by the hardware and will not be enabled until the trampoline has safely returned to the host.

The attacker may also try to perform a denial of service attack by corrupting the interrupt stack (Section 4.3.2). Such attempts will be foiled by the runtime check before the `ESP`-changing instructions. Similar to the `VMWRITE` instruction, the runtime check and its following instruction that modifies `ESP` cannot overlap the fragment boundary. Therefore, the runtime check cannot be bypassed by the attacker.

**Misusing HyperLock Services**  Another set of targets for the attacker is the five services provided to KVM by HyperLock. Since they can directly access the host OS, we have sanity checks in place to prevent these services from being misused. For example, we validate that only guest memory can be mapped by the `update_spt/npt` services, and `map_gfn_to_pfn` can never allocate more memory than that specified by the user when starting the VM. Moreover, as shown in Table 4.1, these services have a small code base (569 SLOC) and therefore can be reviewed or verified to remove vulnerabilities.
Table 4.2: Software packages for the evaluation of HyperLock

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bonnie++</td>
<td>1.03e</td>
<td>bonnie++ -f</td>
</tr>
<tr>
<td>Kernel (61MB) build</td>
<td>2.6.32.39</td>
<td>make defconfig;make</td>
</tr>
<tr>
<td>SPEC CPU 2006</td>
<td>1.0.1</td>
<td>reportable int</td>
</tr>
<tr>
<td>Ubuntu desktop</td>
<td>10.04.2 LTS</td>
<td>Linux-2.6.32.31</td>
</tr>
<tr>
<td>Ubuntu server</td>
<td>10.04.2 LTS</td>
<td>Linux-2.6.32.28</td>
</tr>
</tbody>
</table>

Case Studies To better understand the protection provided by HyperLock, we examine several real-world vulnerabilities from NVD [63] and show how HyperLock could mitigate these threats. The first vulnerability we examined is CVE-2010-3881, a kernel-level bug in KVM in which data structures are copied to user space with padding and reserved fields uninitialized. This bug could potentially lead to leaking of sensitive content on the kernel stack. Under HyperLock, the host OS is not directly accessible to KVM and each guest is paired with its own KVM instance. Therefore, only data related to the guest itself could be leaked to it. The second vulnerability we examined is CVE-2010-0435, in which a malicious guest can crash the host by causing a NULL pointer dereference in KVM’s x86 emulator. Under HyperLock, this vulnerability could be similarly exploited by the guest and trigger an exception. However, instead of crashing the host, HyperLock would terminate only the KVM instance paired with this guest (Section 4.3.2). The last vulnerability we examined is CVE-2011-4127, a bug related to device emulation in QEMU in which a guest can gain access to the data of other guests that reside on the same physical device due to insufficient checks of the SCSI ioctl commands. Under HyperLock, this vulnerability could be mitigated by system call introspection on QEMU.

4.4.2 Performance Evaluation
To evaluate the performance overhead caused by HyperLock, we test the guest performance with several standard benchmarks, including SPEC CPU 2006 [89], Bonnie++ (a file system performance benchmark) [25], and two application benchmarks (Linux kernel decompression and compilation). Our test platform is based on the Dell XPS studio desktop with a 2.67GHz Intel Core™ i7 CPU and 3GB memory. The host runs a default installation of Ubuntu 10.04 LTS desktop with the 2.6.32.31 kernel. The guest is based on the standard Ubuntu 10.04 LTS server edition. Table 4.2 lists the software packages and configurations used in our experiments. Among these benchmarks, SPEC CPU 2006, kernel decompression are CPU intensive tasks, while the other two tests (kernel compilation and Bonnie++) are more intensive in I/O accesses. For the kernel decompression and compilation test, we run the tests in the guest with the `time` command, and reported the sum of system and user time. As a result, these two experiments are based on the virtual time and due to the difficulty of keeping exact time in the guest OS [97], they could be less accurate. However, we did not observe clock drift during our experiments.

In our evaluation, we repeated the experiments with three different KVM configurations: the vanilla KVM, the gcc-compiled KVM under HyperLock, and the NaCl-compiled KVM under HyperLock.
Comparing the performance of gcc- and NaCl-compiled KVM allows us to separate the effects of memory access control (switching between address spaces) and instruction access control (instruction alignment). As shown in Figure 4.4, the overall performance overhead introduced by HyperLock is less than 5%. Moreover, the gcc-compiled system has better performance than the NaCl-compiled system in most tests except mcf and Bonnie++ write. Generally speaking, instruction alignment will increase the binary size and reduce the performance because of additional code (mostly NOPs) inserted to align instructions. For example, the NaCl-compiled KVM in HyperLock contains 99,695 instructions, 265% more than the gcc-compiled version’s 37,553 instructions. Also, 91.2% (56,713 out of 62,142) of the added instructions are NOPs. Meanwhile, due to complex interaction of instruction alignment, instruction cache and TLB, the NaCl-compiled system may actually perform better than the gcc-compiled system [113].

To better understand the performance of HyperLock, we measured the latency of context switching between the host OS and hypervisor isolation runtime in HyperLock. Specifically, we created a null RPC in the isolation runtime that did nothing but returned directly back to the host, then called this function 1,000,000 times from the host OS and calculated the average latency of round trip to the isolation runtime. Our results show that each round trip to the isolation runtime costs about 953 ns, or 45% of that to the guest mode (2,115 ns). The performance overhead of HyperLock is directly related to the frequency of context switches to the isolation runtime, which is further determined by the frequency of VM exits as illustrated by Figure 4.1 (at runtime, arrows 3, 4, 5, and 2 form the most active execution path). Advances in both hardware virtualization support (e.g., EPT [42]) and hypervisor software (e.g., para-virtualized devices [75]) have significantly reduced the number of necessary VM exits. For example, the average number of VM exits for the kernel compilation benchmark is 4,913 per second. Also, the latency of address space switch in 64-bit x86 architecture could be significantly reduced by using a new CPU feature called process-context identifier (PCID) [42]. When PCID is enabled, the CPU
tags each TLB entry with the current process-context id, thus rendering it unnecessary to flush all the TLB entries during an address space switch. However, we must still disable global page support (Section 4.3.1) because TLB entries for global pages are shared by all the address spaces even though PCID is enabled.

4.5 Discussion

In this section, we re-visit our system design and explore possible alternatives for either enhancement or justification. First, HyperLock confines KVM with its own paging-based memory space, not segmentation. Though segmentation could potentially provide another viable choice (especially in 32-bit x86 architecture), in HyperLock, we are in favor of paging for two reasons. (1) The 64-bit x86 architecture does not fully support segmentation [42]. Using paging can make HyperLock compatible with both the 32-bit and 64-bit x86 architectures. (2) Paging provides more flexible control over the layout and protection (e.g., readable, writable, or executable) of our isolation runtime. For example, our system maps the guest physical memory starting at address zero, which allows a guest physical address to be directly used by KVM to access guest memory without further address translation. While segmentation may limit memory access to a continuous range, unnecessary components (e.g., system libraries) could be loaded in the middle of this range and cannot be excluded from the segment.

Second, for performance reasons, HyperLock allows KVM to retain and execute two privileged instructions, i.e., VMREAD and VMWRITE. This design choice significantly affects the design of hypervisor isolation runtime for KVM confinement. Particularly, because KVM is still privileged, it is critical to prevent it from executing any unwanted privileged instructions, either intended or unintended [20]. Also, certain x86 architecture peculiarities complicate the design for safe context switches between the host and KVM, which will be invoked by the unsafe KVM. If it runs non-privileged, the design would be much simpler and straightforward. On the other hand, we may choose to run KVM at ring-3 by further replacing these two privileged instructions with runtime services. The use of runtime services is necessary because Intel VT mandates accessing the VMCS with the VMREAD and VMWRITE instructions as the format of the VMCS is not architecturally defined [42]. Thus, it is not feasible to simply map the VMCS in the KVM address space and use memory move instructions to access it. The overhead of frequent VMCS access through runtime services could potentially be reduced though pre-fetching VMCS reads and batch-processing VMCS writes. As such, this design and HyperLock offer different design trade-offs. We leave further consideration of the implications of this choice to future work.

Third, HyperLock enforces instruction alignment [59, 113] to prevent unintended instructions from being generated out of legitimate ones. Alternatively, we can enforce control flow integrity (CFI) [5, 102] on KVM inside the isolation runtime to provide a stronger security guarantee, especially in eliminating recent return-oriented programming (ROP)-based code-reuse attacks [20]. However, one challenge behind CFI enforcement is the lack of an accurate and complete points-to analysis tool that could be
readily applied to KVM. From another perspective, the lack of CFI does not weaken our security guar-
antee because by design the isolation runtime is not trusted and has been strictly confined within its own
address space. In HyperLock, for ease of implementation, we choose to enforce the instruction align-
ment and combine it with other instruction/memory access control mechanisms to meet our design goal
of isolating hosted hypervisors.

Finally, our current prototype defines a narrow interface that exposes five runtime services to support
guests with virtual devices. A few more runtime services could be added to incorporate new functional-
ity. For example, our current prototype does not support multi-core VMs, which could be accommodated
by adding a new service to handle Inter-Processor Interrupts (IPIs). For the current lack of hardware
pass-through support, we can develop a new service to request and release a PCI device on demand and
accordingly manage IOMMU [42] to enforce hardware isolation (e.g., to block DMA-based attacks).
Further, a service to release guest memory back to the host OS might also be needed to support a balloon
driver for cooperative memory management. However, as mentioned earlier, the number of added ser-
vices should be kept to a minimum and scrutinized to avoid being abused (as they will be considered as
part of the HyperLock TCB). Furthermore, our prototype implements update_spt/npt by mirroring
the corresponding KVM data structure. This makes the prototype relatively simpler to implement by
trading off extra memory. This additional memory could be reclaimed in future enhancements to the
prototype by sharing the SPT/NPT between HyperLock (readable and writable) and KVM (read-only).

4.6 Summary

We have presented the design, implementation and evaluation of HyperLock, a system that establishes
secure isolation of hosted hypervisors. Specifically, we confine the hypervisor execution in the isolation
runtime with a separated address space and a constrained instruction set. Moreover, we create a logically
separated hypervisor for each guest, thus ensuring a compromised hypervisor can only affect its own
guest. We have implemented a prototype of HyperLock for the popular open source KVM hypervisor.
The prototype is only 12% of KVM’s code size, and further completely removes QEMU from the TCB
(of the host and other guests). Security analysis and performance benchmarks show that HyperLock can
efficiently provide the intended isolation.
Chapter 5

Countering Kernel Rootkits with HookSafe

5.1 Introduction

In the previous two chapters, we have introduced HyperSafe and HyperLock to provide a solid foundation for secure virtualization systems. A wide range of virtualization-based security mechanisms can be reliably deployed on top of them. HookSafe, the third system proposed in this dissertation, builds upon these two systems to mitigate serious threats posed by kernel rootkits.

Kernel rootkits are considered one of the most stealthy computer malware and pose significant security threats [84]. By directly subverting operating system (OS) kernels, such rootkits can not only hide their presence but also tamper with OS functionalities to launch various attacks such as opening system backdoors, stealing private information, escalating privileges of malicious processes, and disabling defense mechanisms.

Given the serious security threats, there has been a long line of research on rootkit defense. Specifically, prior research efforts can be roughly classified into three categories. In the first category, systems such as Panorama [115], HookFinder [114], K-Tracer [53], and PoKeR [73] focus on analyzing rootkit behaviors. Systems in the second category are primarily designed for detecting rootkits based on certain symptoms exhibited by rootkit infection. Examples are Copilot [67], SBCFI [68], and VMwatcher [45]. In the third category, systems such as SecVisor [79], Patagonix [55], and NICKLE [72] have been developed to preserve kernel code integrity by preventing malicious rootkit code from executing. Unfortunately, they can be bypassed by return-oriented rootkits [40], which will first subvert kernel control flow (i.e., by hijacking function pointers or return addresses on the stack) and then launch the attack by only utilizing legitimate kernel code snippets.

In light of the above threat, it becomes evident that, in addition to the preservation of kernel code integrity, it is also equally important to safeguard relevant kernel control data so that we can preserve
the kernel control flow integrity and thus block rootkit infection in the first place. In this chapter, we consider kernel data as control data if it is loaded to processor program counter at some point in kernel execution. There are two main types of kernel control data: return addresses and function pointers. In prior research, there exist extensive studies [1, 88, 28] on how to effectively protect return addresses some of which [1, 28] have been deployed in real-world applications. In this work, our primary focus is to protect those function pointers. Note that function pointers are typically hijacked or “hooked” by rootkits. For ease of presentation, we use the term function pointers and kernel hooks interchangeably.

To safeguard a kernel hook, an intuitive approach [64] is to leverage hardware-based page-level protection so that any write-access to the memory page with the kernel hook can be monitored and verified. This approach will work well if (1) there exist only a very limited number of kernel hooks for protection and (2) these hooks are not co-located together with frequently modified memory data. Unfortunately, in a commodity OS kernel such as Linux and Windows, it is not uncommon that there exist thousands of kernel hooks and these kernel hooks can be widely scattered across the kernel space. Further, many of them might be dynamically allocated from kernel heap and are co-located together with other writable kernel data in the same physical memory frames. If this intuitive approach is deployed, it has to trap all writes to memory pages containing kernel hooks, even those not targeting at kernel hooks. Consequently, it will introduce significant performance overhead, particularly from frequent unnecessary page faults that are caused by write-accesses to irrelevant data. In fact, our investigation with a recent Linux system indicates that about 1% of kernel memory writes may cause such unnecessary page faults.

To address the above challenges, in this chapter, we present HookSafe, a hypervisor-based lightweight system that is able to efficiently protect thousands of kernel hooks in a guest OS from being hijacked. Our approach recognizes a fundamental challenge, namely the protection granularity gap, that hardware provides page-level protection but kernel hook protection requires byte-level granularity. To tackle this challenge, we observe that these kernel hooks, once initialized, rarely change their values. This observation inspires us to relocate kernel hooks to a dedicated page-aligned memory space and then introduce a thin hook indirection layer to regulate accesses to them with hardware-based page-level protection. By doing so, we avoid the unnecessary page faults caused by trapping writes to irrelevant data.

We have implemented a prototype of HookSafe based on the latest Xen hypervisor [14] (version 3.3.0) and used it to protect more than 5,900 kernel hooks in a Ubuntu 8.04 Linux system. Our experiments with nine real-world rootkits show that HookSafe can effectively defeat their attempts to hijack kernel hooks that are being protected. We also show that HookSafe achieves such a large-scale protection with only 6% slowdown in performance benchmarks [12, 96]. To the best of our knowledge, HookSafe is the first system that is proposed to enable large-scale hook protection with low performance overhead.

The rest of the chapter is structured as follows. We first discuss the problem space HookSafe aims to address in Section 5.2. Then we present our system design and implementation in Section 5.3 and Section 5.4. We show our evaluation results with real-world rootkits and performance benchmarks in
Section 5.5. After discussing limitations of our HookSafe prototype in Section 5.6. Finally, we conclude this chapter in Section 5.7.

5.2 Problem Overview

Kernel rootkits can be roughly classified into two categories: Kernel Object Hooking (KOH) and Dynamic Kernel Object Manipulation (DKOM). KOH rootkits hijack kernel control flow while DKOM rootkits do not hijack the control flow but instead subvert the kernel by directly modifying dynamic data objects. In this work, we focus on KOH rootkits since majority of kernel rootkits in the wild are of this type. In fact, a recent thorough analysis [68] on 25 Linux rootkits indicates that 24 (96%) of them make control flow modifications.

A KOH rootkit can gain the control of kernel execution by hijacking either code hooks or data hooks [104, 114]. Since hijacking a kernel code hook requires modifying the kernel text section which is usually static and can be marked as read-only, it is straightforward to protect them [55, 72, 79]. Kernel data hooks instead are typically function pointers and usually reside in two main kernel memory regions. One is the preallocated memory areas including the data sections and the bss sections in the kernel and loadable kernel modules. The other are the dynamically allocated areas such as the kernel heap. By design, HookSafe aims to prevent kernel rootkits from tampering with kernel hooks in both memory regions with low performance overhead.

As mentioned earlier, to efficiently protect kernel hooks, we face a critical challenge of the protection granularity gap, where the hardware provides page-level protection but kernel hooks are at the byte-level granularity. Since kernel hooks are scattered across the kernel space and often co-located with other dynamic kernel data, we cannot simply use hardware-based page-level protection.
To better understand it, we have analyzed a typical Ubuntu 8.04 server by using a whole-system emulator called QEMU [70]. Our analysis with 5,881 Linux kernel hooks (we describe how we obtain these hooks in Section 5.5) indicates that they are scattered across 41 physical pages and some of them are located in dynamic kernel heap. In Figure 5.1 we show the histogram of the number of kernel hooks in a single memory page. We can see that 14 pages contain less than 50 hooks. In the worst case, one hook is allocated in a page (4,096 bytes) along with other 4,092 bytes of dynamic data. As a result, writes to these physical pages would trigger frequent unnecessary page faults if one marked them as write-protected. To quantify these unnecessary page faults, we recorded all kernel memory write operations in the Ubuntu server. Based on the collected log, within a randomly-selected period of 100 seconds, we found that there are in total 700,970,160 kernel memory writes. Among these writes, there is no single write to the set of 5,881 kernel hooks for protection, while the number of writes to the 41 memory pages that contain protected hooks is 6,479,417. In other words, about 1% of kernel memory writes would cause unnecessary page faults and thus introduce expensive switches between a VM and a hypervisor. When designing HookSafe, a key technical task is to avoid these unnecessary page faults while still effectively securing the protected kernel hooks.

In this chapter, we assume that a trusted bootstrap mechanism such as tboot [94] is in place to establish the static root of trust of the entire system. With that, a trustworthy hypervisor can be securely loaded which, in turn, can protect the integrity of the guest kernel at boot time. We also assume the runtime integrity of hypervisor is maintained (e.g., with the protection of systems such as HyperSafe (Chapter 3) or HyperLock (Chapter 4))

5.3 Design

5.3.1 Overview

HookSafe is a hypervisor-based lightweight system that aims to achieve large-scale protection of kernel hooks in a guest OS so that they will not be tampered with by kernel rootkits. To efficiently resolve the protection granularity gap, in HookSafe we relocate kernel hooks from their original (widely-scattered) locations to a page-aligned centralized location and then use a thin hook indirection layer to regulate accesses to them with hardware-based page-level protection. In other words, we create a shadow copy of the kernel hooks in a centralized location. Any attempt to modify the shadow copy will be trapped and verified by the underlying hypervisor while the regular read access will be simply redirected to the shadow copy. By using hook indirection, we avoid the performance overhead caused by trapping legitimate writes to dynamic kernel data around protected hooks.

In HookSafe, all read or write accesses to protected kernel hooks are routed through the hook indirection layer. For performance reasons, we handle read and write accesses differently. Specifically, for

---

1 We consider the attacks to hypervisor including recent SMM ones [52] fall outside the scope of this thesis.
a normal write access that updates a kernel hook, because only the hypervisor can write to the memory pages of protected kernel hooks, we will transfer the control from the guest kernel to the hypervisor to commit the update and then switch back to the guest kernel. However, for a read access, we use a piece of indirection code residing in the guest OS kernel memory to read the corresponding shadow hook. By doing so, we avoid the overhead of switching from the guest to the hypervisor and vice versa in read accesses. Note that read accesses to protected hooks could be very frequent, and thus we can benefit significantly by keeping read indirection inside the guest OS kernel.

Figure 5.2 shows the overall architecture of HookSafe. Given a set of kernel hooks (for protection) as input, HookSafe achieves its functionality in two key steps:

• First, an offline hook profiler component profiles the guest kernel execution and outputs a hook access profile for each protected hook. A hook access profile includes those kernel instructions that read from or write to a hook and the set of values assigned to it. In the next step, a hook’s access profile will be used to enable transparent hook indirection. For simplicity, we refer to those instructions that access a hook as Hook Access Points (HAPs).

• Second, taking hook access profiles as input, an online hook protector creates a shadow copy of all protected hooks and instruments HAP instructions such that their accesses will be transparently redirected to the shadow copy. The shadow hooks are aggregated together in a central location and protected from any unauthorized modifications.

In the rest of this section, we will describe these two steps in detail. We will focus on key ideas in HookSafe’s design and defer implementation details to Section 5.4.

5.3.2 Offline Hook Profiling

Our first step is to derive, for the given kernel hooks (as input), the corresponding hook access profiles. To do so, there are two main approaches: The first approach is to perform static analysis on the OS kernel source code and utilize known program analysis techniques such as points-to analysis [11] to automatically collect hook access profiles. The second approach is to leverage dynamic analysis without the need of requiring the OS kernel source code. In particular, dynamic analysis runs the target system on top of an emulator (e.g., QEMU [70]) and monitors every memory access to derive the hook access instructions. In comparison, dynamic analysis allows for recording precise runtime information such as
the values a hook has taken, but potentially has less coverage while static analysis is more complete but less precise.

We note that both approaches have been widely explored before and the results can be directly applicable in HookSafe. In our current prototype, we have chosen to implement the offline hook profiler based on dynamic analysis.

5.3.3 Online Hook Protection

After collecting hook access profiles, our next step is to efficiently protect kernel hooks in a guest OS from being manipulated by kernel rootkits. Figure 5.3 shows the architecture of our online hook protection. The essential idea here is to leverage a thin hook indirection layer to regulate accesses to kernel hooks. Specifically, after a guest OS boots up, we first create a shadow copy of identified hooks, and then instrument all HAPs in kernel code so that read or write accesses will be redirected to the hook indirection layer. In addition, there exists a memory protection component in the hypervisor that protects the indirection code, the shadow hooks, and the kernel code in the guest OS. Next, we will describe how HookSafe initializes online hook protection and how it handles read/write accesses.

Initialization

HookSafe initializes the online hook protection in two steps. It first uses an in-guest short-lived kernel module to create the shadow copy of kernel hooks and load the code for the indirection layer. Then it leverages the online patching provided by the hypervisor to instrument the HAPs in the guest kernel. As mentioned earlier, we assume the system bootstrap process and the in-guest kernel module loading are protected by trusted booting [94]. Next we describe these two steps in detail.

As a part of system bootstrap process, the in-guest kernel module will be loaded to allocate memory pages from the non-paged pool. The non-paged memory allocation is needed to prevent them from being
swapped out. After that, the kernel module will copy protected kernel hooks at their original locations to the newly allocated memory pages. Then it will load the code of the indirection layer to these memory pages. Before the guest kernel module unloads itself, it will make a hypercall to notify the hypervisor the starting address and size of the memory pages so that they can be protected.

When the hypervisor receives the hypercall regarding the guest memory pages for shadow hooks and indirection code, it conducts the second step to patch HAPs in the kernel code. The basic idea is to detour the execution of an HAP instruction to the hook indirection layer so that the access to the original kernel hook will be redirected to the corresponding shadow copy.

**Run-Time Read/Write Indirection**

After the initialization, all accesses to protected hooks at the HAPs will be redirected to the hook indirection layer. It then handles hook accesses differently depending on whether it is a read or write access. For read accesses, the indirection layer simply reads from the shadow hooks then returns to the HAP site. For write accesses, the indirection layer will issue a hypercall and transfer the control to the hypervisor. Then the memory protection component in the hypervisor will validate the write request and update the shadow hook if the request is valid. To validate a write request, HookSafe requires the new hook value to be seen in the offline profiling phase. We note that other policies can also be naturally supported in HookSafe, including those proposed in SBCFI [68] to check if the new kernel hook points to a valid code region, a valid function, a valid function with correct type, or the related points-to set calculated from static analysis.

We point out that the hypervisor-based memory protection component protects not only the centralized shadow hooks but also the code for hook indirection and other legitimate kernel code. To do so, HookSafe utilizes the shadow page table realized in the hypervisor for a running guest VM and sets proper protections for memory pages of the protected contents (more in Section 5.4.3).

**Run-Time Tracking of Dynamically Allocated Hooks**

The design of HookSafe is complicated by the support of dynamically allocated hooks. In particular, since those hooks are allocated and deallocated at runtime, we need to effectively keep track of these events. To this end, we notice that a dynamically allocated hook is typically embedded in a dynamic kernel object. In other words, a dynamic hook will be created when the hosting kernel object is being allocated and instantiated from the kernel heap. Similarly, a dynamic hook will be removed when the hosting kernel object is being de-allocated and the related memory space is being returned back to kernel heap. Accordingly, we can instrument the memory allocation/deallocation functions and utilize the runtime context information to infer whether a particular kernel object of interest is being allocated or de-allocated. If one such kernel object that contains a kernel hook is being allocated, a hypercall will be issued to HookSafe to create a shadow copy of the hook (not the entire hosting kernel object!). Similarly,
another hypercall is triggered to remove the shadow copy when the hosting kernel object is released. By introducing two extra hypercalls, HookSafe is able to track the creation and removal of dynamically allocated kernel hooks. After that, HookSafe’s hook indirection layer works the same regardless of the nature of kernel hooks.

Another related challenge is the recognition of those dynamically allocated kernel hooks that may already exist before loading HookSafe’s in-guest module for hook protection. To address that, a natural approach is to initiate the run-time tracking immediately after the guest OS begins execution. However, it implies HookSafe needs to instrument the memory management functions at the very first moment when the guest OS executes. Also without the help from an in-guest module, HookSafe needs to maintain its own buffer to record those dynamically-allocated kernel hooks. To avoid that, in our prototype, we instead take another approach. Specifically, our approach exploits the fact that any dynamically allocated kernel object must be accessible in some way from certain global variable(s) or CPU register(s). If one imagines kernel objects as a graph where the edges are pointers, then all objects will be transitively reachable from at least one global variable. If an object is not reachable in this way, then the kernel itself will not be able to access it and the object cannot be used. A similar observation has also been made in previous work on both garbage collection and state-based control flow integrity [68]. As a result, with the knowledge of these global variables, it is a straightforward process to identify the current run-time addresses of target kernel objects, which contain kernel hooks. After that, we can apply the normal process of creating shadow copies of these pre-existing kernel hooks and patching the guest kernel to redirect access to their shadow copies.

5.3.4 Hardware Register Protection

In addition to regular memory-based kernel hooks, hardware registers such as GDTR, IDTR, DR0-DR7 debug registers, and SYSENTER MSR registers can also be potentially exploited by rootkits to hijack kernel control flow. These hooks are special in that they are invoked directly by the hardware. Therefore it is vital for HookSafe to regulate accesses to these registers as well. To do that, we use hardware-based virtualization support to intercept and validate any write attempts to these registers. Related to the hardware register protection is how we secure the GDT and IDT descriptor tables. These two tables contain critical system data structures and their contents must be protected. We protect these two tables using the hardware-based page-level protection.

Another related issue is how we prevent DMA from being abused to subvert the HookSafe’s memory protection. In particular, to limit the physical memory accessible to a DMA engine of an I/O device, we utilize the hardware-based IOMMU support in Xen to map the address space of DMA engine to a safe place. In other words, the way the IOMMU is set up precludes the possibility of overwriting HookSafe as well as HookSafe-protected memory regions in a guest.
5.4 Implementation

We have implemented a prototype of HookSafe. The online hook protection component was developed based on the Xen hypervisor [14] (version 3.3.0), and the offline hook profiling component is based on QEMU [70], an open source whole-system emulator. The current prototype is mainly implemented and evaluated in a system running Ubuntu Linux 8.04. Because of that, for most of this section, we choose it as the default guest OS protected by HookSafe. In the following, we will first present the implementation of the offline hook profiler. Then we will describe how we implement the hook indirection and the memory protection.

5.4.1 Offline Hook Profiler

Our offline hook profiler is essentially a whole-system emulator with additional functionality in instrumenting and monitoring the execution of every memory access instruction. In particular, given a list of kernel hook locations for protection, we run the target system on top of the profiler. At the end of a run, the profiler records, for each kernel hook, a list of HAP instructions that read from or write to these kernel hooks and the set of values the hook may take at runtime.

QEMU implements a key virtualization technique called binary translation that rewrites the guest’s binary instructions. Our prototype extends the binary translator in QEMU with additional instrumentation code to record executions of instructions that read or write memories. If an instruction accesses any kernel hook in the given list, we mark it as an HAP instruction and log the value that is written to or read from the hook. For a dynamically allocated kernel hook, the profiler also tracks the creation of the hosting kernel object and locates the runtime hook location. At the end of profiling, the collected HAP instructions and recorded hook values will be compiled as the corresponding hook access profile.

Figure 5.4 shows an example profile for the kernel hook `ext3_dir_operations -> readdir` kernel hook located in the `ext3.ko` loadable kernel module (LKM) and has been hijacked by existing kernel rootkits.
Figure 5.5: The implementation of hook indirection

(Section 5.5) for hiding purposes. The profiled results indicate that this specific hook is accessed by two instructions located at 0xc015069a and 0xc01506dd (both in the vfs readdir function). During the entire profiling, this particular hook always points to the ext3 readdir function and there is no instruction observed that will update the hook value.

5.4.2 Hook Indirection

The key novelty in HookSafe is to instrument every HAP instruction such that the access to the (original) kernel hook is transparently redirected to the corresponding shadow hook. In Figure 5.5, we show how the hook indirection is realized in our prototype. In essence, the hypervisor replaces the HAP instruction at runtime with a jmp instruction to detour the original execution flow to specially-crafted trampoline code. The trampoline code collects runtime context information that will be used by the hook redirector to determine the exact kernel hook being accessed. After the hook redirector processes the actual read or write to a shadow hook, the trampoline will execute the HAP-specific overwritten instructions, if any, before returning back to the original program. To make our prototype memory-efficient, each HAP instruction has its own copy of trampoline code but the hook redirector code is shared by all the HAPs. Note that both trampoline and hook redirector code reside in the guest kernel memory and are protected by the hypervisor’s memory protection component. In Figure 5.5, the arrowed lines (with instructions and numbers on them) show the control flow transfers among HAPs, trampolines and the hook redirector.

HAP Patching

Our implementation uses a five-byte jmp instruction (one byte opcode 0xe9 plus the 32-bit offset as operand) to detour the control flow from an HAP instructions to its trampoline code in the hook indirection layer. Since x86 architecture uses instructions with variable lengths, we must align the overwritten
bytes to the instruction boundary. When an HAP instruction occupies more than five bytes, we will fill the rest space with NOP instructions. When an HAP instruction has less than five bytes, we overwrite the subsequent instructions to make the space for the five-byte jmp instruction, and execute these overwritten instructions at the end of hook indirection. By doing so, our detouring code preserves the boundary of subsequent instructions.

Using the same example in Figure 5.4, we show the first HAP instruction (located in 0xc015069a) and its surrounding instructions in Figure 5.6(a). The corresponding C source code is shown in Figure 5.6(b), which belongs to the vfs_readdir function defined in linux/fs/readdir.c of the Linux kernel. In the disassembled code, the first three instructions test whether file->f_op is NULL. The fourth instruction is the HAP instruction that checks whether the hook file->f_op->readdir is NULL. As shown in Figure 5.6(a), this particular HAP instruction only occupies four bytes. To successfully detour its execution with the five-byte jmp instruction, we need to overwrite the subsequent instruction located at 0xc015069e with two bytes as well.

In the patching process, there are two caveats that we have experienced in our prototype, particularly when overwriting additional instructions that follow the patched HAP instruction. First, if two HAPs are close to each other, the first detouring jmp instruction may overwrite the second HAP instruction. Second, even worse, some instruction overwritten by the jmp instruction may be a jump target. In the example shown in figure 5.6(a), there may exist an instruction that directly jumps to the second je instruction, which unfortunately has been overwritten by the detouring jmp instruction. In other words, that instruction would essentially jump to the middle of the detouring jmp, which typically causes an invalid opcode exception.

These two scenarios occur because other instructions than the original HAP instruction are overwritten by the detouring jmp instruction. We solve this problem by conducting function-level binary rewriting similar to Detours [41]. Instead of locally modifying the original function, we create a new copy of it in which we replace HAP instructions with jmp instructions and shift the subsequent instructions accordingly. In addition, we replace the first instruction in the old function with a jmp instruction so that any function call to it will be redirected to the new function. In this way, we avoid rewriting the entire kernel image. Note that we assume there is no control transfer from one function to the middle of another function, which was empirically confirmed in our evaluation.
Read/Write Indirection

The hook indirection layer in HookSafe has two components: trampoline and redirector. The trampoline code prepares the hook-related context information (e.g., the HAP address and machine registers’ contents). The redirector uses the context information to find which hook is being read or written and then identify the corresponding shadow hook.

For the support of variable-length instructions, the HAP patching may overwrite additional instructions that follow an HAP instruction. To reclaim them, our prototype customizes the trampoline code for each detoured HAP instruction as follows. First, the additional overwritten instruction(s) are appended to the end of the trampoline code. By doing so, they will be executed when the control returns back from the hook redirector. Second, at the end of the trampoline code, we further append an additional `jmp` instruction so that the control is transferred back to the original program.

Upon the call from the trampoline code, the redirector first determines which hook is being accessed based on the current CPU context and the semantics of the detoured HAP instruction. Using the first HAP instruction in Figure 5.6(a) as an example the hook’s address is the register `eax` plus `0x18`. The redirector retrieves the content of register `eax` from the CPU context saved by our trampoline code and determines the original kernel hook that is being accessed. After that, the redirector identifies the corresponding shadow hook and performs the desired access indirection. If it is a read access, the redirector will read the shadow hook and update the saved CPU states to reflect the effect of HAP instruction. Continuing the previous HAP instruction example (Figure 5.6(a)), the redirector will update the `eflags` register in the saved CPU states to indicate whether `file->f_op->readdir` is `NULL`. After returning from the redirector, the trampoline will restore saved CPU states. By doing so, the read access of the original kernel hook is effectively redirected to its shadow copy. If it is a write access, the redirector will make a hypercall to the hypervisor so that the memory protection component can verify the new hook value and update the shadow hook if it is legitimate.

In our prototype, instead of completely ignoring original kernel hooks, we also utilize them to detect rootkits’ hooking behavior. More specifically, for each redirected hook read access, the hook indirection layer in addition performs a consistency check between the original kernel hook and its shadow copy. Any difference would indicate that the original hook has been compromised. Similarly, for each redirected hook write access, if the write operation is legitimate, we need to update both the shadow hook and the original hook to keep them synchronized.

Run-Time LKM and Hook Tracking

To support runtime tracking of dynamically allocated kernel hooks, we first observe that the lifetimes of these hooks are consistent with their hosting kernel objects. In addition, in Linux, these hosting kernel objects are typically allocated/deallocated through the SLAB interface. More specifically, the SLAB allocator manages caches of objects (that allow for fast and efficient allocations) and each different
type of object has its own SLAB cache identified by a human-readable name. Since there are two main
functions to allocate and deallocate a kernel objects, i.e., \texttt{kmem\_cache\_alloc} and \texttt{kmem\_cache\_free},
our current prototype instruments these two functions using a very similar technique with hook indirection.
Specifically, before these two functions return, the instrumented code checks whether the SLAB
manages a particular kernel object of interest (i.e., whether it contains a kernel hook that needs to be
shadowed). If so a hypercall will be issued to notify HookSafe so that it can track the hook creation and
termination. Note that the instrumented code runs in the guest kernel space and the world switch only
occurs when the kernel object being allocated/deallocated through the SLAB interface contains a kernel
hook of interest.

Related to dynamic kernel hook tracking is how we support the kernel hooks inside the Loadable
Kernel Modules (LKMs). For a given LKM, because its runtime memory will not be determined until
at runtime, it poses additional challenges for HookSafe to precisely determine the locations of those
hooks contained in the LKM. Fortunately, for those kernel hooks (and HAP instructions) inside a LKM,
the relative offsets to the LKM base address where the module is loaded are fixed. Therefore a LKM
hook’s runtime location can be simply calculated as the addition of the LKM’s current base location and
the relative offset. Based on this observation, a hook address is extended to a tuple of \((\text{module\_hash},
\text{hook\_offset})\), where the hash is used to uniquely identify a module.

To precisely locate the LKM base address at runtime, we utilize the well-known technique called
virtual machine introspection [37, 72, 79]. Specifically, during our profiling and online HookSafe pro-
tection, we leverage the virtual machine introspection to intercept module loading/unloading events in
the guest kernel. After a module is loaded and fixed up by the guest kernel, we then derive the runtime
location of a LKM hook by simply adding the base address of the module and the relative offset of the
hook. An example of such LKM hook is shown in Figure 5.4 where the \texttt{ext3\_dir\_operations \rightarrow readdir}
is a kernel hook inside the \texttt{ext3.ko} module.

5.4.3 Memory Protection

To protect the guest kernel code and the in-guest memory used by HookSafe, we leverage the shadow
page table (SPT) management subsystem in the Xen hypervisor. In the SPT memory management mode,
the hypervisor maintains an SPT for each guest, which regulates the translation directly from a guest
virtual address to the host physical address. Any update to the guest page table (GPT) in the guest
kernel is trapped and propagated to the SPT by the hypervisor. In other words, the hardware performs
the address translation solely with the shadow page table.

To protect guest memories, we check if the guest virtual address is in the range of our protected
memories before propagating the changes in GPT to SPT. If so, we make sure the physical pages are
marked read-only.
Table 5.1: Effectiveness of HookSafe in preventing 9 real world kernel rootkits: *Hiding fails* indicates that rootkit failed to hijack the control flow because HookSafe has redirected the hook to its shadow; *Installation fails* indicates that the rootkit hooking behavior causes the memory protection violation, hence failing the installation. Additionally, depending on how the rootkit searches for the system call table, it may locate either the original system call table (marked with †) or the shadow system call table (marked with ‡).

<table>
<thead>
<tr>
<th>Rootkit</th>
<th>Attack Vector</th>
<th>Hooking Behavior</th>
<th>HookSafe Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>adore-ng 0.56</td>
<td>LKM</td>
<td>proc_root_inode_operations -&gt;lookup</td>
<td>Hiding fails</td>
</tr>
<tr>
<td></td>
<td></td>
<td>proc_root_operations -&gt;readdir</td>
<td>Hook indirection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ext3_dir_operations -&gt;readdir</td>
<td>Hiding fails</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ext3_file_operations -&gt;write</td>
<td>Hook indirection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unix_dgram_ops -&gt;recvmsg</td>
<td>Hiding fails</td>
</tr>
<tr>
<td>eNYeLKM</td>
<td>LKM</td>
<td>kernel code modification</td>
<td>Installation fails</td>
</tr>
<tr>
<td>sk2rc2</td>
<td>/dev/kmem</td>
<td>sys_call_table[...NR_oldolduname]†</td>
<td>Installation fails</td>
</tr>
<tr>
<td>superkit</td>
<td>/dev/kmem</td>
<td>sys_call_table[...NR_oldolduname]‡</td>
<td>Installation fails</td>
</tr>
<tr>
<td>Phalanx b6</td>
<td>/dev/mem</td>
<td>sys_call_table[...NR_setdomainname]‡</td>
<td>Installation fails</td>
</tr>
<tr>
<td>mood-nt 2.3</td>
<td>/dev/kmem</td>
<td>sys_call_table[...NR_olduname]‡</td>
<td>Installation fails</td>
</tr>
<tr>
<td>override</td>
<td>LKM</td>
<td>sys_call_table[...NR_geteuid]†</td>
<td>Hiding fails</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys_call_table[...NR_geteuid]‡</td>
<td>Hook indirection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys_call_table[...NR_getdents64]†</td>
<td>Hiding fails</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys_call_table[...NR_chdir]‡</td>
<td>Hook Indirection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sys_call_table[...NR_read]‡</td>
<td>Hiding fails</td>
</tr>
<tr>
<td>Sebek 3.2.0b</td>
<td>LKM</td>
<td>sys_call_table[...NR_read]‡</td>
<td>Installation fails</td>
</tr>
<tr>
<td>hideme.vfs</td>
<td>LKM</td>
<td>kernel code modification</td>
<td>Installation fails</td>
</tr>
</tbody>
</table>

5.4.4 System Call Indirection Optimization

When building our prototype, we realized that the system call table provides a unique opportunity for optimization. Note that the Linux’s system call table shares a memory page with other dynamic data [72], which means that we cannot simply mark the memory page read-only but rely on HookSafe’s protection. In the meantime, a system call table usually contains a large number of hooks. For instance, in the kernel we tested, the system call table has 330 function pointers and each of them may be hooked by a rootkit. Due to the fact that accesses to the system call table is fairly frequent, it is also critical to optimize its indirection to reduce the overhead.

We observe that the system call table in the linux kernel is accessed by only two *read* HAP instructions, and both of them has the base address of the system call table hardcoded in the instructions. This provides a unique opportunity for optimization. In our prototype, instead of creating a shadow for each individual system call hook, we simply create a shadow system call table and replace the base addresses in the two HAP instructions with the new shadow table address. By doing so, we essentially eliminate the system call redirection overhead caused by hook indirection. Our experience indicates that this is a special case and we did not find any other hooks that can be optimized this way.

68
5.5 Evaluation

In this section, we present our evaluation results. In particular, we have conducted two sets of experiments. The first set of experiments (Section 5.5.1) is to evaluate HookSafe’s effectiveness in preventing real-world rootkits from tampering with kernel hooks. We tested HookSafe with nine real-world rootkits. It successfully prevented all of them from modifying protected hooks and hiding themselves. The second set of experiments (Section 5.5.2) is to measure performance overhead introduced by HookSafe. We evaluated HookSafe on benchmark programs (e.g., UnixBench [96] and ApacheBench[12]) and real-world applications. Our experimental results show that the performance overhead introduced by HookSafe is around 6%.

In our experiments, HookSafe takes as input two sets of kernel hooks. The first set includes 5,881 kernel hooks in preallocated memory areas of main Linux kernel and dynamically loaded kernel modules. Specifically, we derive this set by scanning the data/bss sections of the kernel and LKMIs in a guest VM running Ubuntu Server 8.04 (with a default installation). In our experiment, we examine those sections every four bytes (32-bit aligned) and consider it as a kernel hook if it points to the starting address of a function in the kernel or kernel modules. At the end, we found 5,881 kernel hooks in the guest VM. The second set is from 39 kernel objects (with function pointers) that will be dynamically allocated from kernel heap. We obtain this set by manually going through a subset of the entire Linux source code and locate those kernel objects of interest. Note that during runtime, it is not uncommon that tens or hundreds of copies of the same type of kernel objects will be allocated. Not surprisingly, a large part of them are related to timer, callback/notifier functions, and device drivers. We point out that this set can be further improved from existing systems such as [53, 73, 114, 115] as they are capable of profiling rootkit execution and reporting those compromised kernel objects with hooks. Given the above two sets of kernel hooks, with our offline profiler, we identified 968 HAP instructions for these hooks: 785 of them are read HAPs while the remaining 183 are write HAPs. Next we describe our experiments in detail.

5.5.1 Effectiveness Against Kernel Rootkits

We have evaluated HookSafe with nine real-world Linux 2.6 rootkits shown in Table 5.1. These rootkits cover main attack vectors and hooking behaviors of existing kernel rootkits, therefore providing a good representation of the state-of-the-art kernel rootkit technology. HookSafe successfully prevented these rootkits from modifying the protected kernel hooks: these rootkits either failed to hide their presences or failed to inject code into the kernel. In the following, we describe in detail our experiments with two representative rootkits.

Adore-ng Rootkit Experiment The adore-ng rootkit infects the kernel as a loadable kernel module. If successfully installed, it will hijack a number of kernel hooks and gain necessary control over kernel execution so that it can hide certain rootkit-related files, processes, and network connections.
Meanwhile, it also has a user-level control program named \textit{ava} that can send detailed instructions (e.g., hiding a particular file or process) to the rootkit module.

For comparison, we performed our experiments in two scenarios: First, we loaded \textit{adore-ng} in a guest OS that is \textit{not protected} by HookSafe and showed that it can successfully hide a running process as instructed by the \textit{ava} program (see Figure 5.7(a)). Second, we repeated the experiment in the same guest OS that is now \textit{protected} by HookSafe. This time the rootkit failed to hide the process (see Figure 5.7(b)).

By analyzing the experiments, we found that the rootkit was able to locate and modify certain kernel hooks at their original locations. However, since now control flows related to these hooks are determined by the shadows hooks, this rootkit failed to hijack the control flow and thus was unable to hide the running process.

As mentioned earlier, the hook indirection layer performs an additional check by comparing the original kernel hook with its shadow copy whenever the hook is accessed. As a result, we are able to successfully identify these kernel hooks that are being manipulated by \textit{adore-ng}. Our experiments show that the \textit{adore-ng} rootkit hijacks a number of kernel hooks, including \texttt{proc\_root\_inode\_operations ->lookup}, \texttt{proc\_root\_operations ->readdir}, \texttt{ext3\_dir\_operations ->readdir}, \texttt{ext3\_file\_operations ->write}, and \texttt{unix\_dgram\_m\_ops ->recvmsg}. As pointed out earlier, the \texttt{ext3\_file\_operations} kernel object is a part of the \texttt{ext3.ko} and this module will be loaded somewhere in the kernel heap at run time. A detailed analysis with the rootkit source code reveals that the first two are hijacked for hiding processes, the next two are for hiding files and directories, and the last one is for filtering messages to the \texttt{syslogd} daemon. This demonstrates that, with HookSafe’s protection, this rootkit and others of its kind will not be able to hijack these hooks to hide their malicious activities.

\textbf{Mood-nt Rootkit Experiment} The \textit{mood-nt} rootkit attacks the Linux kernel by directly modifying the kernel memory through the \texttt{/dev/kmem} interface. This is a rather advanced kernel-level attack that
leverages an attack strategy reflecting the so-called return-to-libc attacks [20, 40, 82]. Specifically, it first overwrites a function pointer, i.e., a system call table entry called \texttt{NR\_olduname}, via the \texttt{/dev/kmem} interface and makes it pointing to a kernel function called \texttt{vmalloc}. Then it “executes” this function by invoking a syscall (with syscall number \texttt{NR\_olduname}) and this legitimate kernel function will allocate a chunk of contiguous kernel memory for the rootkit. After that, it populates the memory with its own code (via the same \texttt{/dev/kmem} interface), which essentially installs itself in the kernel. Next it overwrites the same function pointer again but this time it will point to its own code, and invokes it by making another syscall (with the same syscall number). Finally, it launches its payload and starts manipulating the OS kernel.

The \texttt{mood-nt} rootkit demonstrates the trend of leveraging the return-to-libc attack strategy to evade systems [55, 72, 79] that aim to preserve kernel code integrity. Instead of hijacking a syscall table entry, it can also choose to control any other kernel hook and potentially make use of it multiple times and each time it can point to an arbitrary kernel function routine or code snippet. By doing so, the rootkit can manage to sequentially execute these legitimate kernel functions in the order chosen by the rootkit without injecting its own code.

With HookSafe’s protection, this rootkit was stopped at the first step when it tried to overwrite the function pointer \texttt{NR\_olduname} in the shadow syscall table. Interestingly, the rootkit somehow is able to locate and attempt to modify the corresponding shadow copy of the function pointer. It turns out it identifies the shadow copy by following the new base address of the syscall table in the syscall handler, which is provided in our optimization (Section 5.4.4). As a result, by stopping \texttt{mood-nt} at the first place, HookSafe essentially prevents function pointers from being subverted to manipulate kernel control flow.


d\texttt{5.5.2 Performance}

to evaluate the performance overhead introduced by HookSafe, we measured its runtime overhead on 10 tasks including those in the UnixBench [96] and Linux kernel decompression and compilation. We also measured its throughput degradation on a web server using the ApacheBench [12].

Our testing platform is a Dell Optiplex 740 with an AMD64 X2 5200+ CPU and 2GB memory. HookSafe ran with the Xen hypervisor of version 3.3.0. The guest OS is a default installation of Ubuntu

---

Table 5.2: Software packages for the evaluation of HookSafe

<table>
<thead>
<tr>
<th>Item</th>
<th>Version</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu Server</td>
<td>8.04</td>
<td>standard installation (kernel 2.6.18-8)</td>
</tr>
<tr>
<td>Kernel Build</td>
<td>2.6.18-8</td>
<td>make defconfig &amp; make</td>
</tr>
<tr>
<td>Gunzip</td>
<td>1.3.12</td>
<td><code>tar -xzf &lt;file&gt;</code></td>
</tr>
<tr>
<td>Apache</td>
<td>2.2.8</td>
<td>default configuration</td>
</tr>
<tr>
<td>ApacheBench</td>
<td>2.0.40-dev</td>
<td><code>-c3 -t 60 &lt;url/file&gt;</code></td>
</tr>
<tr>
<td>UnixBench</td>
<td>4.1.0</td>
<td>default configuration</td>
</tr>
</tbody>
</table>
server 8.04 with a custom compile of the standard 2.6.18-8 kernel. Table 5.2 lists the configurations of the software packages used in our evaluations. In the Apache test, we ran an Apache web server to serve a web page of 8K bytes. We ran the ApacheBench program on another machine in the same gigabit Ethernet to measure the web server throughput. For each benchmark, we ran 10 experiments with and without HookSafe and calculated the average.

In Figure 5.8 we show the runtime overhead on 7 tasks in the UnixBench, kernel decompression and kernel build, as well as the throughput drop in ApacheBench. We can see that the maximum overhead added by HookSafe is 6.5% when the standard Linux kernel source package linux-2.6.18-8.tar.gz (51 megabytes) is decompressed with the gunzip program. In the ApacheBench test, the throughput degradation caused by HookSafe is 6.1%.

The ApacheBench experiment is particularly interesting. In a short time period of 1 minute, the Apache server accepted more than 10,000 TCP connections from the ApacheBench. For each TCP connection, two dynamic kernel objects containing function pointers were created: `struct sock` (a kernel object for an active network socket) and `struct ip_conntrack` (a kernel object used in the Linux packet filtering framework for IP connection tracking purposes). Note that although the `sock` object will be destroyed when the connection is closed, the `ip_conntrack` object lives longer until the connection completely expires. During the ApacheBench test, at its peak, there were 24 active sockets and 10,096 `ip_conntrack` objects alive, which means during this particular test, there are around 16,000 kernel hooks (including more than 10,120 dynamically allocated hooks) being protected by HookSafe. To quantify the performance overhead, we measured the ApacheBench’s throughput by only enabling run-time hook tracking. The results show that the Apache server is able to transmit 491.81 MB in one minute period, an overhead of 0.7% when compared to the throughput of 495.39 MB without HookSafe’s protection.

In addition, we performed a micro-measurement on the overhead due to the hypercall made when creating a socket object. In this measurement, we recorded the hardware timestamp counter register
(with the instruction `rdtsc`) [6] right before and after the socket creation call, i.e., the `sk_alloc` function. Originally, it took 2266 CPU ticks for `sk_alloc` to complete while, with the overhead of an additional hypercall, it required 5839 CPU ticks to complete. As a result, the hypercall incurs 157% overhead for a socket creation. Note that this overhead occurs only when the socket is being created, which counts for a small part of the computation for the entire lifetime of a socket connection. In other words, the performance degradation due to these hypercalls is amortized over the lifetime of kernel objects. The longer the lifetime, the smaller the performance overhead.

In conclusion, HookSafe is lightweight and able to achieve large-scale hook protection with around 6% performance slowdown.

### 5.6 Discussion

A fundamental limitation faced by our current HookSafe prototype for Linux guest OS is that hook access profiles are constructed based on dynamic analysis and thus may be incomplete. The lack of completeness is an inherent issue of dynamic analysis shared by other approaches [53, 73, 104, 114]. In our case, it could result in missing HAPs and legitimate hook values and has the following impact on the protected system. First, since HookSafe maintains the consistency between original hooks and shadow hooks, unpatched read HAP instructions would work normally in a clean system by reading from original hook locations. However, if a guest OS is compromised, there may exist a (small) time window in which a rootkit may hijack the control flow. Specifically, after a rootkit modifies original hooks and before HookSafe detects the inconsistency, the control flow will be hijacked if an unpatched HAP instruction reads from the original hook location to decide a control transfer. Second, for unpatched write HAP instructions, their writes to original hook locations will not affect patched read HAPs. Before HookSafe detects the inconsistency, the protected system will be in an unstable state. Third, for legitimate hook values missing in hook access profiles, HookSafe will raise a false alarm when an HAP instruction attempts to update a hook with such a value. However this is not a serious problem during our experiments. We detected only 5 (or 0.085%) kernel hooks with an incomplete hook value set.

To mitigate the incompleteness problem, there are two possible approaches. The first one is to improve the coverage of dynamic analysis. Recent efforts such as multiple path exploration [61] have shown promise that one can leverage runtime information to systematically guide the exploration process to achieve better coverage. The second one is to combine a complementary approach – static analysis (Section 5.3.2), which is more complete but less precise. Note that an imprecise hook access profile has the implication of causing false alarms. However, the integration of dynamic analysis and static analysis remains an interesting research problem.

Another limitation is that HookSafe assumes the prior knowledge of the set of kernel hooks that should be protected from rootkit attacks. In other words, HookSafe itself is not designed to automatically discover those kernel hooks. From another perspective, this problem is being addressed to some extent
by a number of existing systems, such as HookFinder[114] and HookMap [104], to systematically derive the set of kernel hooks that are or will be of interest to rootkits. We expect that our work will be combined with these efforts in the future to enable integrated hook discovery and rootkit defense.

5.7 Summary

We have presented the design, implementation, and evaluation of HookSafe, a hypervisor-based lightweight system that can protect thousands of kernel hooks from being hijacked by kernel rootkits. To enable large-scale hook protection with low overhead, HookSafe overcomes a critical challenge of the protection granularity gap by introducing a thin hook indirection layer. With hook indirection, HookSafe relocates protected hooks to a continuous memory space and regulates accesses to them by leveraging hardware-based page-level protection. Our experimental results with nine real-world rootkits show that HookSafe is effective in defeating their hook-hijacking attempts. Our performance benchmarks show that HookSafe only adds about 6% performance overhead.
Chapter 6

Conclusion and Future Work

In this dissertation, we have presented a systematic and integrated approach to establish secure virtualization systems. By hardening the hypervisors, our approach provides a solid TCB for these systems. Specifically, HyperSafe enables the self-protection for a type-I hypervisor by enforcing its control flow integrity, and HyperLock strictly isolates a vulnerable type-II hypervisor in a dedicated address space to prevent it from compromising the host or other guests. With the help of these two systems, a wide spectrum of virtualization-based security services can be reliably deployed, for example, to introspect the guest kernel status [45], detect kernel rootkit [104], and protect the guest kernel integrity [54, 103]. In this dissertation, we demonstrate the effectiveness of this approach with a third system called HookSafe that can protect the guest kernel from malicious kernel rootkits.

The approach in this dissertation has laid a solid foundation towards trustworthy virtualization systems. It creates various opportunities for future work. In the following, we propose three directions for future work:

• Hypervisor Enhancement HyperSafe enforces the control flow integrity for type-I hypervisor. By protecting the code and control data integrity, HyperSafe can defeat a large number of attacks that rely on malicious code execution, thus severely limiting the attacker’s capabilities and provide fairly strong security guarantees. However, the enforcement of CFI only requires runtime execution to follow a path in the CFG, regardless whether this path is correct in the context. For example, an attacker can manipulate the program to return to a function, say \( f \), that is not the active caller of the current function, say \( g \), as long as function \( f \) calls function \( g \) somewhere. In the future, we will investigate how to leverage the context (e.g., the call stack) to enforce a stricter control flow integrity. In addition, an out-of-scope threat to type-I hypervisors is the attacks that rely only on the manipulation of non-control data. This calls for the broader security property of data integrity. How to efficiently enforce data integrity is largely an unsolved problem, especially in the context of hypervisors.
In HyperLock, we isolate a type-II hypervisor in its own dedicated address space. For performance reasons, we still run the hypervisor at ring-0, the most privileged level in the x86 architecture, but constrain its execution by limiting the availability of privileged instructions to the hypervisor. Because of this, we cannot run the hypervisor in an unprivileged user space. In the future, we will investigate efficient approaches to move the hypervisor to the user space, for example, through aggressive caching of the VMREAD and VMWRITE instructions.

- **Mobile Virtualization** Smartphones have become ubiquitous nowadays. A typical smartphone contains a wealth of sensitive personal information, a wide variety of sensors, and many third-party applications. As such, smartphones are increasingly becoming the focus of malicious attacks, such as privacy leak, root exploits, or mobile botnets. Meanwhile, smartphones have become powerful enough to efficiently support virtualization. This makes it possible to deploy many virtualization-based security mechanisms in smartphones to protect user security and privacy. In the future, we will seek efficient ways to apply some of the techniques in this dissertation to build secure mobile hypervisors and protect the user data.

- **Cloud Computing** Cloud computing is a disruptive technology that revolutionizes the way we do computing today. Security remains as one of the major concerns that hinders the adoption of cloud computing. In this dissertation, we have been focused on securing the underlying infrastructure of cloud computing. Beyond that, cloud computing has wider security implications, for example, how to reduce the amount of trust that we must have in the cloud provider, or how to ensure regulation and law compliance (e.g., location constraints) when outsourcing data to the cloud. In the future, we will investigate reliable and secure virtualization-based systems in the context of cloud computing to address these unique challenges.
REFERENCES


