1-2014

Virtualization-based System Hardening against Untrusted Kernels

Yueqiang CHENG
Singapore Management University, yqcheng.2008@phdis.smu.edu.sg

Virtualization-Based System Hardening Against Untrusted Kernels

YUEQIANG CHENG

SINGAPORE MANAGEMENT UNIVERSITY
2013

Virtualization-Based System Hardening Against Untrusted Kernels

by

Yueqiang Cheng

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Robert DENG Huijie (Supervisor / Chair)
Professor of Information Systems
Singapore Management University

Xuhua DING (Co-supervisor)
Associate Professor of Information Systems
Singapore Management University

Debin GAO
Assistant Professor of Information Systems
Singapore Management University

Yongdong WU
Senior Scientist of Infocomm Security Department
Institute for Infocomm Research

Singapore Management University
2013

Virtualization-Based System Hardening Against Untrusted Kernels

Yueqiang Cheng

Abstract

Applications are integral to our daily lives to help us processing sensitive I/O data, such as individual passwords and camera streams, and private application data, such as financial information and medical reports. However, applications and sensitive data all surfer from the attacks from kernel rootkits in the traditional architecture, where the commodity OS that is supposed to be the secure foothold of the system is routinely compromised due to the large code base and the broad attack surface. Fortunately, the virtualization technology has significantly reshaped the landscape of the modern computer system, and provides a variety of new opportunities for us to protect application and sensitive data.

In this dissertation, we first design and implement a lightweight and reliable hypervisor *Guardian* as the system secure foothold, which leverages virtualization technology and a secure boot and shutdown mechanism to protect itself in its whole life cycle. Guardian is the first bare-metal hypervisor with integrity and availability guarantees. Moreover, we extend Guardian to be a framework of secure foothold, which consists of summarized common security primitives for facilitating our proposed systems and other security services. Based on the reliable secure foothold (Guardian), we propose *AppShield*, which protects critical applications through putting them into isolated execution environments (IEEs). In an IEE, AppShield is able to reliably and efficiently protect data secrecy and integrity of a critical application, as well as the execution integrity, against kernel rootkit attacks. Moreover, it is able to defend against newly identified threats, which are evidence that protecting applications against the malicious OS is more difficult than previously realized.

The inputs and outputs of protected application are not protected by AppShield such that they could be tampered by kernel rootkits. To fix this gap, we propose

a trusted path (TP) scheme, named as *Driverguard*, to protect I/O flows between hardware input/output devices and protected applications. DriverGuard is the first generic approach that protects all kinds of I/O flows with a combination of cryptographic and virtualization techniques. The combination of IEE and TP could protect almost all applications and sensitive data. But for certain user data, we could do it better. In this dissertation, we purpose a dedicated system *KGuard* to protect user passwords in the increasingly popular online services without needing any IEE and trusted path. In particular, KGuard does not trust any software components in the guest kernel and user space (without IEE requirement), and also not leverage any special hardware to assist the protection.

We implement the prototypes of all the above systems, and evaluate their performance overheads. The experiment results show that the performance costs on CPU computation and device I/O are insignificant.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am grateful to my advisors, Robert H. Deng and Xuhua Ding, for guiding me in my research, helping me to develop strong research stills, and encouraging me to become a better person overall. I am also very grateful to the other members of my thesis committee, Debin Gao and Yongdong Wu, for their advice and guidance throughout the dissertation process. Their invaluable comments, guidance and encouragement dramatically helped me clarify my thesis, refine my approach and broaden my vision of security research. Seeing their work and accomplishments inspired me to become a more rigorous researcher. I greatly appreciate the guidance and assistance of my mentors Virgil Gligor and Adrian Perrig at Carnegie Mellon University. I would also like to thank all my co-authors and collaborators: Zongwei Zhou and Miao Yu, Qijia Wang, Zhi Zhang, Li Deng, for their indispensable collaboration help.

Finally, I dedicate this dissertation to my wife, Caiyu Lu, for her continuous love and support, and my son, Haoran Cheng, for the ultimate joy and happiness he brings to me.

Chapter 1

Introduction

1.1 Problem Overview

Commodity operating systems are ubiquitous in home, commercial, government, and military settings. The OSes are supposed to be the secure footholds of systems since they manage the entire resources, so compromising an OS compromises every program and all security services upon it. In fact, unfortunately, the OSes are routinely compromised due to the large code base, the broad attack surface and the buggy loaded modules (drivers). Specifically, modern OSes usually have large code base. According to the statistics from kernel developers [103], the Linux kernel version 3.5 contains 39096 files with $15.6M$ SLOC, and its next version (i.e., version 3.6) adds 637 files with around $300,000$ SLOC in less than 3 months (i.e., 71 days). For such large code base, it is impossible for current formal verification techniques to formally remove all bugs [39]. The broad attack surface is also a threat for OS security. The statistical results show that the Linux kernel version 3.6 has up to 337 system calls besides many implicit interfaces (e.g., $/proc$ and $/sys$ virtual file systems) for applications to communicate with the OS. The dynamically loaded modules and drivers are usually buggy (not fully tested) and they inevitably increase the complexity of the OS and the number of attack interfaces. All these factors cause the OS is not a good candidate of secure foothold of a system.

Once a part of the monolithic commodity OS is compromised, the whole OS is compromised and consequently all applications are compromised. Applications are used in our daily life to process sensitive information, from sensitive I/O data, such as individual passwords, biometric finger prints and camera streams, to private application data, such as online banking information, medical reports and email content. When the OS (secure foothold) is compromised, all data within application address space are freely accessed by the adversary, and all security protections that are built upon the OS to intentionally protect the application data are consequently broken. Thus, a new application protection scheme that should be able to protect the whole applications address space against the potentially compromised OS is desired.

Another main threat from the compromised OS is the secrecy of the I/O data, which are transmitted between the hardware input/output devices and user applications via the compromised OS. The I/O data usually contains sensitive information, such as the data rendered from applications, e.g., sound data and printer data, or generated for applications, e.g., keyboard passwords and fingerprints. In such I/O data, all network traffics and disk files could be well protected through encryption, such as the SSL technique can protect network I/O data. However, for the *raw* I/O data, such as keyboard inputs, camera streams and fingerprints cannot be protected through encryption techniques, because they are structured data. The encryption may break constrains or change the meaning of certain fields, and consequently cause unpredictable errors. For example, a camera stream consists of header and data sections that are encoded by the camera device. If we simply encrypt the whole stream, the header information will be lost or wrongly translated, which will lead to video errors and even crash the camera driver or the corresponding user application. In addition, the commodity hardware devices are not encryption-capable. Thus, the generated data or the received data should be in plain text, such as the keyboard scan code and the document content receiving by the printer. Once kernel rootkits know the locations of the buffers caching the plain text, they can reveal and

tamper with the data by directly accessing them.

To address the above problems, we need to answer a series of critical research questions: which component/layer of the system could be the secure foothold? What security mechanisms can be applied to it to make it reliable while still compatible with existing OS and applications? How to protect the whole application against untrusted OS while still keeping it usable, e.g., allowing it to exchange data with outside and use the memory as a fashion in the normal setting? How to efficiently protect the secrecy of the I/O data transmitting between hardware input/output devices and applications via the potentially compromised OS? Is the encryption-capable device necessary? Can we do it better if we only focus on the protection on one specific I/O data? Furthermore, Based on the secure foothold, what secure primitives it can provide, and what security services can be built, besides the application protection and the I/O data protection?

1.2 Research Objectives

In this dissertation, we aim to answer the above questions. Moreover, we aim to design and implement several secure virtualization systems to demonstrate and evaluate our solutions on today's hardware and software architecture.

1.2.1 Reliable Secure Foothold

The hypervisor is a promising secure foothold candidate. As a small piece of software between the OS and the hardware, the hypervisor has the unique advantages, in terms of the balance between security and versatility. However, to be a reliable secure foothold, the hypervisor should meet two critical requirements. Firstly, it should be secure against attacks from rootkits which can subvert the operating system. Secondly, it should be always *available* throughout the life cycle even when the OS is corrupted. By virtue of the virtualization, a hypervisor is widely deemed as software which can resist attacks from an untrusted guest OS. However, *no* hypervisor can simultaneously satisfy all the above requirements, especially for the

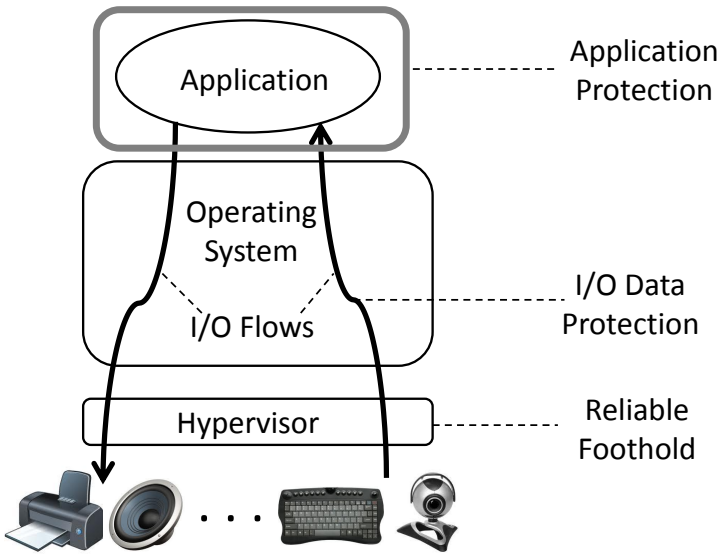


Figure 1.1: Research Objectives. In this dissertation, we focus on the application protection and I/O data protection based on a reliable secure foothold.

availability requirement. It is challenging to achieve the reliability property, because 1) the kernel rootkits can directly modify or even delete the TCB (including the hypervisor) images from the disk to make them unavailable. If the hypervisor chooses to do runtime checking through intercepting disk I/O, it will lead to numerous context switches, reducing the performance of the disk I/O as well as the CPU utilization; and 2) the rebooting and shutdown mechanism of commodity systems are complex. The hypervisor should efficiently and completely handle all of them to ensure the availability before the system really rebooting/shutdown.

In this dissertation, we aim to build a reliable and lightweight secure foothold with integrity and availability guarantees. Moreover, we aim to extend our secure foothold to be a framework, which consists of common security primitives that are potentially needed by our proposed systems and many other security systems. In this way, people can quickly adopt it to support new security services with no or minimum customizations.

1.2.2 Application Protection

The application protection is challenging due to the complex memory management and the intensive data exchange between the protected applications with

outside (e.g., the OS). The approaches like Flicker [57], TrustVisor [56] and Fides [89] simplify the problem setting by assuming that the protected code is self-contained with pre-defined inputs and outputs (e.g., inputs are the initial parameters and outputs are the final returns); and that the protected execution does not involve dynamic memory allocation or deallocation. Some other systems [57, 56, 89, 13, 17, 106, 92, 16, 90] aim to protect the whole application in the real settings. However, they have various drawbacks, such as high performance overhead, large Trusted Computing Base (TCB), hardware modifications, or with a restriction imposed on the protected code. Moreover, several newly identified threats in Chapter 4 are evidence that protecting applications from malicious OS is more challenging than previously realized. For example, the malicious OS may swap two address translation mappings to break the data integrity without directly accessing the data pages (i.e., mapping reorder attack). It may also illicitly return an allocated memory region with its virtual addresses occupied by the application stack (i.e., Iago attack [15]). By doing so, the application may happen to modify the control data (e.g., return address) in the stack, and thereby compromise the execution of the victim application.

Thus, in this dissertation, we aim to propose a system to protect an application with small TCB and low performance loss, as well as no modifications on system hardware. Moreover, the new scheme should be able to defend against newly identified attacks.

1.2.3 I/O Data Protection

Application protection can only protect the data within application address space. For the I/O data out of the protected application (transmitting in the kernel space) are not protected. Thus, a I/O data protected scheme is desired. In fact, protecting I/O data is to build trusted paths, which are secure channels that assure the secrecy and/or authenticity of data transfers between a hardware devices and an isolated execution environment. Without a trusted path, an adversary could illicitly

obtain sensitive user inputs and the outputs of an application. Traditional trusted paths are built upon commodity OS, which is not a reliable secure foothold. Thus, some researchers propose new trusted path schemes to defend against untrusted OS. However, they all have several drawbacks, such as needing encryption-capable devices [59, 97], and incompatible with legacy applications [109].

Thus, in this dissertation, we aim to propose a generic, secure (against untrusted OS) and practical (without needing special hardware and compatible with legacy applications) trusted path scheme for protecting I/O flows on the commodity platforms. It is challenging since it needs to handle *all* kinds of device I/O, and has *no* extra assistance from special hardware.

Password Protection

The combination of IEE and TP could protect almost all applications and sensitive data. But for certain user data, we could do it better, even in terms of security. In this dissertation, we aim to propose a dedicated system to protect user passwords in the increasingly popular online services. To make the new system secure and practical, it should satisfy several requirements. Firstly, the new system should not need any trusted information from isolated/trusted components in the guest space. Secondly, it should not need special hardware to build trusted path. Thirdly, the new system should be compatible with existing OS and all applications, especially for legacy web browsers. Fourthly, the new system should be user-friendly, meaning the interaction interface for web authentication should keep the same. Last but not the least, the performance overhead and the latency in the whole process of the web authentication should be insignificant. In this dissertation, we aim to let the new proposed system support all these five requirements.

1.3 Threat Model

We assume the end-users themselves are security conscious, meaning that they are aware of the potential dangers/attacks that arise from adversaries. The security-

conscious users are willing to enable some protection mechanisms and are wary of authentication of the availability of the protection, e.g., the user will verify if he/she gets a secret feedback (a secret message or a flashing signal) when the corresponding protection mechanism is enabled.

We assume that a remote adversary completely controls the guest operating systems, meaning that the remote adversary is able to launch arbitrary code or applications to access any system resources, such as I/O ports, Memory-Mapped I/O (MMIO) and main memory. Specifically, the adversary may 1) get the data stored in the device memory or I/O ports by manipulating the configuration space of the target or other devices, 2) intercept the I/O flows that goes through the kernel space via device drivers, and 3) read user sensitive data from the user space by directly accessing a particular memory region or following the links in some data structures. The purposes of the adversary are to break the system foothold (the hypervisor), and/or get the secrecy of the I/O data.

We assume that the adversary can not compromise the hardware devices whose behaviors always exactly follow their specifications. We also assume the system firmware is trusted. In fact, the modern BIOS has a built-in hardware lock mechanism [91] to set itself as read-only so that the OS cannot tamper with it. Furthermore, the modern BIOS only accepts signed updates [45, 97]. Due to the complexity of the x86 platform (e.g., optional ROM), this assumption may not always true. Nonetheless, it is still possible to validate the system firmware by the proposed attestation approach [52] or by a trusted system integrator.

The hypervisor is trusted. The load-time integrity of the hypervisor is achieved through secure boot mechanism [96], and the runtime security is guaranteed by the virtualization technology. Specifically, the hypervisor isolates its memory region by configuring Extended/Nested Page Table (EPT/NPT) and IOMMU to stop software and DMA access driven by hardware. The hypervisor is not as secure as the TPM chip since several attacks have been discovered to compromise some versions of hypervisors [94, 27, 49, 69]. However, the security of the hypervisor can be verified

by those schemes [99, 98, 5, 69].

1.4 Metrics

We identify several key metrics to evaluate the security, efficiency and practicality of a protection mechanism.

- Size of Code Base (e.g., SLOC). The size of the TCB should be as small as possible. The smaller TCB will export less attack surface which may be used by adversaries to compromise the system, and its security may be proven through the rigorous formal verification mechanism [39].

- Performance Overhead. The security protection system should only introduce reasonable performance overhead for the protected target (e.g., an application) and other components in that system. The poor performance may dramatically reduce the value of the protection system.

- Compatibility. The compatibility issue should be considered during the design of a protection system. It is better to be transparent to either the legacy applications or the operating systems if it is not simultaneously compatible with both of them.

- Practicality. The whole protection mechanism should be user-friendly for normal end-users who do not possess or lack security domain knowledge. Furthermore, the mechanism should be able to be widely deployed without requiring special hardware (e.g., encryption-capable keyboard).

1.5 Terminology

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

- **Secure Foothold.** A secure foothold is a piece of code with highest privilege in a system. The secure foothold is trusted and always behaves as expected. It can bolster various security systems by growing the trust chains.

- **Isolated Execution Environment (IEE).** The execution environment is defined by code S executing on a specific platform. The isolated execution environment protects the execution of S from any other code.

- **Trusted Path (TP).** A Trusted Path (TP) is a secure channel that assures the secrecy and/or authenticity of data transfers between a hardware devices and an isolated execution environment. Without a trusted path, an adversary could illicitly obtain sensitive user inputs and the outputs of an application.

- **Trusted Computing Base (TCB).** The trusted computing base of a computer system is the set of all hardware, firmware, and software components that are critical to its security. Once they are compromised by the adversary, the security properties will be broken.

- **Virtualization.** Virtualization is a technology that creates, manages, introspects and destroys guest domains (virtual machines). The software that implements virtualization is usually called *virtual machine monitor (VMM)* or *hypervisor*.

1.6 Dissertation Overview

In this dissertation, we leverage virtualization technology to harden systems [19, 18, 20, 21]. Specifically, we build a lightweight and reliable hypervisor (Guardian) as a system secure foothold, which is able to boost computer security and provide reliable assistance for the end user to cope with various threats. Guardian as the system secure foothold has two prominent features. The first is a secure bootup and shutdown mechanism, which enhances the existing hardware-based secure bootup by offering integrity and availability protection of the TCB images and critical information. Another feature is a user-hypervisor interface which allows the end-user to issue commands to and receive responses from Guardian at runtime. The interface is secure in the sense that the channel between the end-user and the hypervisor is authentic and the exchanged information is not exposed to the guest.

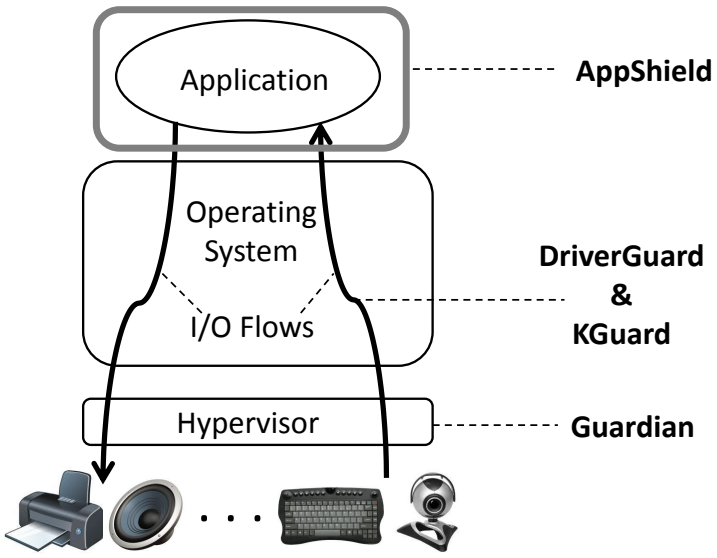


Figure 1.2: AppShield protects an application through an isolated execution environments built upon Guardian (the reliable secure foothold). The I/O flows between the protected application and hardware I/O devices are protected by DriverGuard through trusted paths. For user passwords, they are specifically protected by K-Guard.

Based on Guardian, we propose AppShield, which protects critical applications by putting them into isolated execution environments (IEE). AppShield is able to reliably and efficiently protect data secrecy and integrity of a critical application, as well as its execution integrity, against kernel rootkit attacks. Specifically, AppShield leverages the hardware-assisted virtualization techniques [46] to isolate the application's address space such that all accesses from the untrusted OS are blocked except those explicitly authorized by the application through system calls. The protected application is allowed to utilize the main memory in the same fashion as in a normal (unprotected) setting, and to access the memory with native speed, i.e. without encryption/decryption or being intercepted. AppShield localizes the performance overhead to the protected application and keeps the performances of those unprotected applications unaffected.

The inputs and outputs of protected application are not protected by AppShield so that they could be tampered by the untrusted OS. To fix this gap, we propose DriverGuard to support trusted paths between the protected application and hardware input/output devices. DriverGuard is a holistic and compact I/O protection

system making use of a combination of cryptographic and virtualization techniques. In particular, the trusted path built by DriverGuard focuses on those devices that render raw data, e.g., sound cards and printers, or generate raw data for applications, e.g., seismic sensors and fingerprint scanners. Disk and network I/O are less concerned, because such data could be simply protected through encryption.

The combination of IEE and TP could protect almost all applications and sensitive data. But for certain user data, we could do it better, even in terms of security. In this dissertation, we purpose a dedicated system *KGuard* to protect user passwords in the increasingly popular online services without needing any IEE and trusted path. In particular, KGuard does not trust any software component in the guest kernel and user space (without IEE requirement), and also not leverage any special hardware to assist the protection. Note that KGuard is able to protect user passwords against all kernel- and application-level key-loggers as well as all kinds of network eavesdroppers.

Based on the above proposed systems, and many other existing virtualization-based security systems, we summarize the common security primitives into our secure foothold (Guardian), upgrading it to be a framework/template of secure foothold so that it could be easily customized by end users according to their demands to harden their systems. To demonstrate the value of the framework, we create four security utilities, i.e., hypervisor-based firewall, device monitoring, software runtime attestation and user present attestation for password authentication. The experiment results show that we only need adding a few lines of code to achieve all these security services, and the performance overheads are insignificant.

1.6.1 Dissertation Organization

The remainder of this dissertation includes the following chapters. Chapter 2 summarizes the related work, and Chapter 3 presents Guardian as a lightweight and reliable system foothold. Based on Guardian, Chapter 4 describes AppShield that creates isolated execution environments to protect critical applications. Chapter 5

presents a trusted path scheme that is a generic solution to protect all kinds of I/O data, and Chapter 6 presents KGuard to protect user passwords in the web authentication services. Chapter 7 summarizes a framework of secure foothold that could be directly used or customized by end users to harden their systems. Finally, Chapter 8 concludes the dissertation and discusses a few directions for the future work.

Chapter 2

Related Work

We first describe isolated execution environment (IEE) approaches, and then we present the trusted path schemes which aims to protect I/O data. The IEE scheme are complementary with trusted path mechanism to protect the whole lifecycle of application and I/O data. Finally we describe the hypervisor security and the root of trust to illustrate the reasonability of choosing the hypervisor as system secure foothold.

2.1 Application Protection

There are several approaches proposed to protect application through creating isolated execution environments, and all of them attempted to remove the OS out of TCB.

2.1.1 Self-contained Code Protection

Flicker [57] system built on the TPM-based Dynamic Root Of Trust (DROT) technology can build an isolation environment to protect a piece of code and data. Due to the limitation of the TPM, the latency of the Flicker system is significantly high. To minimize the latency, TrustVisor [56] scheme are proposed. By leveraging virtualization technology, TrustVisor virtualizes the physical TPM into Virtual TPM (VTPMs) and migrate them into hypervisor space. Note that both of them simplify

the problem setting by assuming that the protected code is self-contained with predefined inputs and outputs (e.g., inputs are the initial parameters and outputs are the final returns); and that the protected execution does not involve dynamic memory allocation or deallocation. Protecting complex environment such as the whole application or device drivers may lead both schemes to failure.

2.1.2 Whole Application Protection

Secure-Processor-Based Protection. AEGIS [90] and XOM OS [53] are secure-processor based approaches that provide compartments to isolate one application from others. Both of them incur poor computability since they require substantial modifications on the OSes and applications. AEGIS [90] also provide an alternative implementation, which requires to build security into the OS.

Bastion [12] and SecureME [22] aim to deal with untrusted OS and untrusted hardware attacks simultaneously with the assistance of a secure processor. Bastion focuses on the protection of a security module, while SecureME attempts to provide privacy and integrity for data and code of the application. SecureME requires modifications on both OSes and applications.

In addition, a Processor-Measured Application Protection Service P-MAPS [72] is announced by Intel, which is built upon Intel TXT [24] and Intel VT [46] hardware capabilities. P-MAPS provides runtime isolation to protect standard applications with small TCB. P-MAPS is quite similar to our scheme at a high level. However, the details of P-MAPS are unavailable for public to conduct an in-depth comparison.

Microkernel-Based Protection. EROS[81], Perseus[68], Microsoft's NGSCB [29] and Nizza [37] are microkernel(or small kernel) based solutions. They attempt to run commodity OS and untrusted applications in the low-assurance partitions, and run the applications with higher security requirements in the high-assurance partitions, which are isolated and protected by the microkernel itself. However, all of them incur compatibility issue since they may require splitting or

even redesigning on the applications.

Virtualization-Based Protection. The approaches like TERRA [34] and Proxos [92] are hypervisor-based trust partitioning systems. They protect applications by isolating them into trusted domains with application-specific OSes. These systems incurs large TCB since they include all secure domains inside. In addition, they are still vulnerable once the application-specific OSes are compromised.

OverShadow [17], CHAOS [16] and SP^3 [106] aim to protect the whole application execution against malicious application and OSes. However, all of them need complex encryption and decryption operations on the application data. Obviously, these additional costly cryptographic operations may reduce the performance and increase the latency of the whole system, especially for the protected application. In addition, none of them claims that they protect applications from the MS attack. Thus, the data and code integrity may still be broken by potentially compromised OS. InkTag [41] is a new proposed approach, which also protects the whole application and verifies the OS behaviors through paraverfication technique. The paraverfication technique needs to modify the source code of the kernel, which is not always available. Thus, it may lead to the failure of the protection on the close-source OSes, e.g., Windows. In addition, it is unclear if InkTag can defend against the new attacks identified in our dissertation 4.

BIOS-Based Protection. Lockdown [97] system relies on a BIOS-assisted lightweight hypervisor and an ACPI-based mechanism to provide two switchable worlds - green world for trusted applications and red world for untrusted applications. Lockdown uses a trusted path built upon LEDs to provide a verifiable protection. The main drawback of the Lockdown system is the switch latency is too high. The switch requires 31 seconds on Windows and $13 - 28$ seconds on Linux. SecureSwitch [91] system that is quite similar to Lockdown also leverages a BIOS-assisted mechanism for secure instantiation and management of trusted execution environments. The switch latency is relatively smaller. The switch requires 6 sec-

onds by using the ACPI standard. Both approaches needs to shut down one world to run another one, meaning they can not simultaneously execute two worlds.

2.2 I/O Data Protection

We attempt to categorize these relevant trusted path schemes according to the differences of their secure foothold.

2.2.1 Virtualization-based Protection

The trusted path [109] proposed by Zhou et al. aims to assure the *secrecy* and *authenticity* of I/O data transferred between a periphery device and an application. To build an exclusive trusted path between the device and the expected application, the hypervisor fixes the device configuration space which is achieved using Virtual Machine Control Structure/Block, Nested/Extended Page tables and IOMMU to prevent unauthorized software and DMA access, and interrupt delivery path which is achieved using Interrupt Remapping features and LAPIC x2APIC mode to make sure that the interrupt is delivered to expected handler. The expected applications are extended to support user-level drivers, which directly issue command to devices through the built trusted path. Obviously, it suffers compatibility issues since applications are numerous and any new application will need to be modified to satisfy the trusted path requirements. In comparison, DriverGuard requires modifications on driver code only. Moreover, DriverGuard does not need to defend against interrupt spoofing attack since it focuses on the *secrecy* of the I/O data between devices and applications, and only authorized PCB is able to access the decrypted I/O data. Even if the unauthorized codes are involved by unintended interrupt, they still can not access the protected I/O data.

The virtualization-based trusted path schemes are closely related to our work. BitVisor [84] is a dedicated hypervisor to I/O management. It uses a parapass-through mechanism whereby access operations on the monitored devices are intercepted and the operations on the other devices pass through without any checking.

The interception allows the hypervisor to protect itself and to perform security functions on the device I/O. However this approach does not protect the I/O data in the kernel space. BitVisor does not claim that they protect the MMIO mapping attacks.

Hypervisors with privileged root domains (e.g., the *Dom0* in Xen) are able to assign different device drivers to separate virtual machines (e.g., *driver domains* in Xen) and securely associate them with application virtual machines (e.g., *guest domains* in Xen) [23, 8, 6, 75, 104]. These hypervisors isolate the device resources (i.e., I/O ports and the memory address-space, including MMIO regions) belonging to a device driver domain from other domains. Note that all these schemes do not consider the MMIO mapping attacks. Moreover, their TCBs enclose the whole operating system, which dramatically increases their trusted code bases.

2.2.2 Hardware-based Protection

The Zone Trusted Information Channel (ZTIC) [50] is a dedicated hardware, which provides a trusted path for users to confirm online transactions. The ZTIC-based trusted path completely bypasses the legacy channel in the users' computers. Bumpy [59] system proposes to protect user keyboard inputs by building a trust environment. It requires an encryption-capable keyboard and therefore is not applicable to generic devices.

The special-device-based schemes usually combine with cryptographic technology to protect the secrets in user and kernel space, such as Bumpy, which usually requires many mediations on the system. Such medications not only consiquently affect the compatibility of the scheme, but also often significantly reduce the usability, and even make the scheme impractical sometimes.

The UTP system [30] proposes an isolated kernel module to temporally manage user-centric I/O devices (e.g., keyboard and display) and enables a remote server to verify that a transaction summary is confirmed by a local keyboard input. BIND [83] binds data and code and uses cryptographic techniques to guarantee the integrity of data. However BIND is limited to derived data and can not help on the

confidentiality of the I/O data. Both of them require a hardware supported secure execution environment (e.g., secure kernel based on the AMD's Secure Execution Mode chip), which often occurs high latency and significant performance overhead.

2.2.3 OS-based Protection

Langweg et al. [51] propose a COTS-based scheme to solve the confidentiality, integrity and authenticity of input and output data[1]. Authors focus on the windows platform, where the Windows message mechanism is able to be exploited by a malicious program to access the input and output messages (data) which are originally intended for other applications. By leveraging the advantages of the the DirextX, authors build a secure user interface to directly fetch input data and access the display hardware in exclusive mode. Trusted paths for browsers [107] focus on providing a trusted GUI to user, protecting user inputs to the intended browser. These two schemes only address security issues at the driver-applications interface, whereas the battlefield of DriverGuard is the entire I/O path. BitE [58] is an approach for preventing user-space malicious applications from accessing sensitive user input via a dedicated trusted path between input devices and the target application, and providing visual verification feedback to the user to prove that the input is really caught by the expected application.

All of them suffer from a large TCB since they are built atop large operating systems, and some of them even contains the Window Manager [58].

2.3 Hypervisor Security

Comparing with legacy monolithic Operating Systems, the hypervisor is more secure since its size is relatively smaller and the exported attack surfaces for guest domains are considerably less. Although there have been several attacks discovered to compromise some versions of hypervisors [94, 27, 49, 69], the security of the hypervisor can be enhanced through some existing mechanisms. The TPM-based

[1]The confidentiality of the input data is not done.

authenticated boot can verify the integrity of the hypervisor when being launched, and the hardware-assisted virtualization technology, i.e., Intel VT-x and AMD V, is able to significantly reduce the code size of the hypervisor, thereby the attack surface is reduced. Furthermore, there are some sophisticated framework systems [99, 98, 5, 69] proposed to enhance the security of the hypervisor. HyperGuard [69], HyperCheck [98] and HyperSentry [5] are three System Management Mode (SMM)-based frameworks to measure and verify the integrity of hypervisors. The code for the SMM mode are protected by hardware chipset. HyperSafe [99] is a lightweight approach that protects existing bare-metal hypervisors with a unique self-protection capability to provide lifetime control flow integrity. In order to eliminate the programming bugs in the hypervisor, the rigorous formal verification mechanism [39] is able to be used to prove the correctness of the hypervisor.

Many hypervisor-based security systems have been designed and reported in the literature. For instance, a hypervisor can be applied for I/O related protection [84, 19], for kernel integrity protection [78, 71, 100, 65, 4, 60, 105], and for user space protection [34, 17, 106, 56]. By studying these systems, we identify interception and manipulation as the hypervisor's major security primitives which are adopted in Guardian as well. Our work has remarkable differences with the aforementioned systems. Guardian caters to the enduser's security needs, instead of the security of platform components such as the kernel or a user process. This demands Guardian to be highly efficient, user friendly and compatible with the operating system and applications, such that it is suitable for practical use. Most existing systems just briefly mention the TPM based secure boot for loading the hypervisor. In contrast, we devise a novel method to establish the hypervisor as the root of trust and show how to build a variety of security services on top of it.

2.4 Root of Trust

2.4.1 Software-based Root of Trust

Software-based ROTs have been proposed and used in [77, 3, 79]. The trust establishment is based on a challenge-response protocol. A speed-optimized function (code block) is established as the ROT on a platform if, within an acceptable time delay, it can compute a correct checksum of memory regions according to a given challenge. It is based on the assumption that it incurs a noticeably longer delay for any other implementation of this function. It also has a restriction on both the adversary's capability, for instance no collusion with a third party, as mentioned in [28]) and the capabilities of the target platforms. In addition, to stop the proxy attack, it may even require to unplug the network and disable the wireless to physically cut down the connection with outside. These limitations and requirements lead to inconvenience or even to impracticability. Thus, software ROTs are unqualified to be a security foothold for normal users' computers.

2.4.2 Hardware-based Root of Trust

The hardware-based ROT can be categorized into static ROTs and dynamic ROTs. A static ROT is a built-in platform component. When the platform boots up, a trust chain can be established from the ROT up to the operating system. The TPM chip [96] is a typical example of static hardware ROT. As a chip on the motherboard, it is secure against all software attacks. Secure (or authenticated) boot up, remote attestation and sealed storage are the main security services provided by the TPM framework. The main disadvantages of TPM are its low speed, inflexibility and passiveness. Therefore, to support various security services, it usually requires assistance from certain secure software routine (e.g., hypervisor). IBM's secure coprocessor [2] is a strong hardware root of trust with such a high price tag that it is not feasible for the mass market. SMART [28] is a hardware-software co-designed scheme, where a piece of code works on a modified *low-end* microcontroller u-

nits (MCU) to function as a dynamic ROT. The SwitchBlade architecture [11] can prevent persistent rootkits from infecting security-critical files (e.g., kernel image) with an ROT residing on the disk controller. These ROTs may be integrated with Guardian though carefully design and implementation.

AMD Secure Virtual Machine (SVM) [1] and Intel Trusted Execution Technology (TXT) [24] are dynamic ROTs. These new processor features allow a piece of code to be securely executed in an isolated environment enforced by the hardware. Despite of their easiness of use, they incur high latency as showed in the Flicker system [57]. Fortunately, the high latency may be tolerable for the end-users, since it only required once when the system as well as Guardian boots up. The boot mechanism of Guardian is compatible with dynamic ROT techniques.

Chapter 3

Reliable Secure Foothold

In this chapter, we harness the fast-growing hardware-assisted virtualization techniques to build a tiny but reliable hypervisor as the security foothold for personal computers. The hypervisor we propose is named as *Guardian*. Guardian has two prominent new features which are the enabling techniques for the hypervisor to become a security foothold. The first is a new secure bootup and shutdown mechanism, which enhances the existing hardware-based security boot up by offering integrity and availability protection of the TCB image and critical information. The other feature is a secure user-hypervisor interface which allows the end-user to issue commands to and receive responses from Guardian at runtime. The interface is secure in the sense that the channel between the human end-user and the hypervisor is authentic and the exchanged information is not exposed to the guest. We also propose two practical security utilities based on Guardian. The first is a device monitor utility, whereby the user can instruct Guardian to monitor the state of peripheral devices, e.g., a camera. The second is a hyper-firewall whereby Guardian inspects inbound/outbound network traffic and drops illegal packets. We have implemented Guardian on a desktop with a Linux guest. Guardian consists of around $25K$ SLOC, and the utilities consist of around $2.1K$ SLOC. Our experiments show that Guardian inflicts an insignificant workload to the whole system.

The growing hardware support for virtualization will continue to empower the

hypervisor with more effective and stronger security control over commodity platforms with smaller code size and better performance. We envisage that using a hypervisor as a generic security foothold is a promising direction to greatly boost up the security for commodity platforms. Our work presented in this chapter is an important step towards this ultimate goal. We summarize our contributions as follows:

1. We design and implement Guardian which is the first system to provide both *integrity* and *availability* guarantees. Note that all existing hypervisors do *not* achieve the availability guarantee.

2. We design and build a device monitor and a hyper-firewall as two security utilities on top of Guardian.

In the next section, we present our research objectives and threat model. Then we present the design of Guardian in Section 3.2. In Section 3.3, we describe the implementation and the evaluation. Finally, we conclude the chapter in Section 3.4.

3.1 Problem Definition

We aim to provide a tiny and reliable hypervisor as a security foothold for personal computers. Namely, we undertake to furnish the end-user with a reliable security basis when the conventional one (typically the operating system) fails. Though the security foothold, the human user can configure security policies and manage resources in the platform. It not only boosts up the system security, but also facilitates the end-user to determine the trustworthiness of her system. Note that we do *not* attempt to detect and remove malicious software from the platform, nor is to protect the operating system or a user application.

3.1.1 Threat Model

Since our goal is to assist the end-user, we assume that they are security- conscious users, who are happy and intended to use our system to protect their systems. We do

not consider any human adversary who may have physical access to the system. For instance, the adversary can issue malicious DMA accesses by inserting extra physical devices (e.g., a firewire device). A malicious human user can always remove the hypervisor from the platform.

The adversary in our threat model is malware residing in the operating system which can subvert the operating system and launch arbitrary attacks. However, we assume that they can not compromise the hypervisor. Note that the hypervisor makes use of hardware-assisted virtualization techniques to defend against malicious software accesses and illicit DMA accesses. This assumption can be more reasonably held if the hypervisor has a tiny code size and simple logic so that only a small attack interface is exposed to the adversary. Existing techniques [99, 98, 5, 69] can also be applied to enhance hypervisor security.

We assume that the adversary can not compromise the hardware devices whose behavior always exactly follow their specifications. We also assume the system firmware is trusted. In fact, the modern BIOS has a built-in hardware lock mechanism [45, 91] to set itself as read-only so that the OS cannot tamper with it. Furthermore, the modern BIOS only accepts signed updates [93, 97]. Due to the complexity of the x86 platform (e.g., optional ROM), this assumption may not always true. Nonetheless, it is still possible to validate the system firmware by the proposed attestation approach [52] or by a trusted system integrator.

3.2 Design of Guardian

In this section, we introduce the techniques for establishing Guardian as a security foothold, and describe the functionalities of the two secure user interfaces.

3.2.1 Establishing Guardian as a Security Foothold

To establish Guardian as a security foothold, it is necessary but not sufficient to ensure a secure boot. The secure boot alone can only validate the integrity of the system's TCB image during booting up, while a reliable security foothold needs

both integrity and availability guarantee, so that the system still boots up into a trusted state even if the TCB image on the hard drive are modified by attackers. We do not elaborate the details of secure boot (e.g., TPM-based secure boot [96]) to avoid verbosity as it has been widely used in the literature. Our focus is to explain how to ensure that the intact TCB image is always available for the boot up. The TCB of our system consists of the BIOS, the bootloader-core and the Guardian image. Recall that the BIOS is protected by the hardware and is trusted in our threat model. Therefore, we intend to protect the bootloader core and the Guardian image against runtime attacks.

A straightforward approach is for Guardian to intercept and validate every disk I/O, such that any access to the security critical image residing on the disk is blocked. Obviously, this solution is costly due to the high overhead and complexity of a disk I/O interception multiplied by the huge number of disk operations.

We devise a novel scheme without interposing on disk operations. The basic idea (visualized in Figure 3.1) is that once Guardian is launched, it immediately relocates its image and the bootloader core from the disk into a protected memory region *prior to* launching the guest. Then, Guardian intercepts all power off events, and writes the protected image back to the disk before cleaning up the memory. In the following, we describe the details of secure boot up and secure shutdown, which in tandem with runtime protection bolster the availability of Guardian throughout its whole life cycle.

Secure Bootup

Figure 3.2 illustrates the disk layout for Guardian, where a special partition, referred to as the *hypervisor-partition*, is created during installation to avoid being trespassed by normal file systems. To allow for a secure boot without increasing the TCB size and complexity, we make slight changes on the bootloader (e.g., Grub 2). The BIOS passes the control to the bootloader core in the boot track. The bootloader core includes the Master Boot Record (MBR), the diskboot image and the basic-

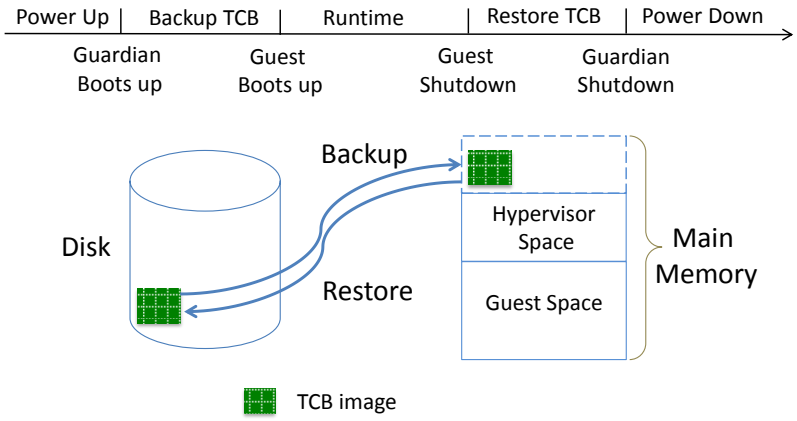


Figure 3.1: Protection of the TCB (from power up to power down). The TCB consists of the Guardian image and the bootloader core. The protected memory for the TCB image is reserved by Guardian and inaccessible for the guest OS.

function image, which provides all basic functions and usually has to load other modules and configuration files such as *grub.cfg* to launch an operating system due to the limited size of the boot track (32KB in maximum).

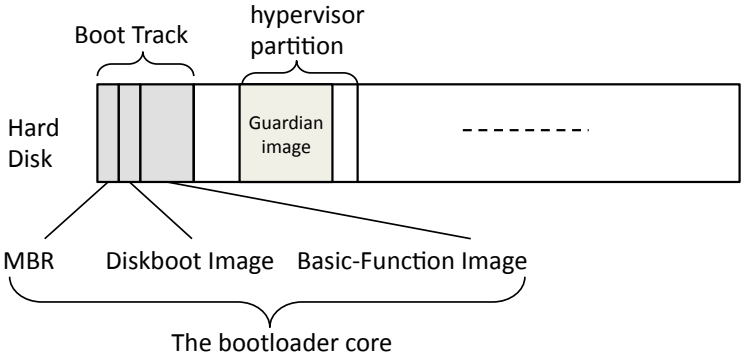


Figure 3.2: An illustration of the disk layout.

Our modification is on the basic-function image only, such that it always launches Guardian *before* loading other components including the OS. In specific, once the core is loaded to the CPU by the BIOS (illustrated by Step 1 in Figure 3.3), it checks a bit flag in main memory (referred to as VMM_flag) which indicates Guardian's presence. If VMM_flag is not set, i.e., the core immediately passes the control to Guardian whose image is placed at a *fixed* disk address upon installation (Step 2 in Figure 3.3). The address of Guardian is hard-coded into the core, such that it loads Guardian directly using disk I/O without involving any file system.

After occupying the CPU, Guardian loads the TCB image into a reserved memory region. It then configures the hypervisor page table, the EPT and IOMMU to ensure that the reserved region is not in the hypervisor or the guest's space and not accessible by DMA devices either. Separating the reserved region from the hypervisor space ensures no accidental accesses to the region. (As shown later, Guardian must map the region into its space by re-configuring the page table in order to access it.)

Finally, Guardian sets VMM_flag indicating its presence, and passes the control back to the bootloader core (Step 3). After asserting the flag is set, the core loads other modules and configuration files (Step 4) and proceeds to boot up the guest in the normal way (Step 5).

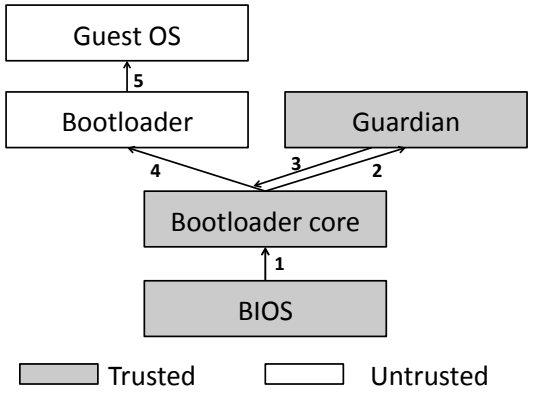


Figure 3.3: The sequence of secure bootup.

Device Configuration Space Protection. A rootkit may manipulate the device configuration space (e.g., the space-overlapping attack [109]) to thwart Guardian to intercept certain I/O events or access to I/O data. In order to defeat the configuration space manipulations and conflicts/overlapping between different devices, Guardian is poised to intercept and validate any update to the device configuration registers after its boot up. Note that these registers are located in the northbridge chipset [31]. The interception are realized via configuring Virtual-Machine Control Structure (VMCS) for I/O ports and the EPT for MMIO regions.

Secure Shutdown

The guest may modify the Guardian image on the disk. Therefore, when the system is powered off, the TCB saved in the reserved memory must be written back to their original locations in the disk for the next round of execution. There exist two types of shutdown events. One type is the sleep events, where the system enters a sleep state through the Advanced Configuration and Power-management Interface (ACPI) [40]; the other is the reboot event, where the system restarts from the BIOS. Guardian intercepts both types of shutdown events and responds accordingly.

ACPI Sleep. The ACPI sleep event is managed by the Operating System Power Management (OSPM) subsystem on the modern ACPI-compatible system. Receiving commands from software (e.g., system call) or external interrupts (e.g., the System Control Interrupt triggered by pressing the power/sleep button or closing the laptop lid), the OSPM subsystem sets the PM1a_CNT register to force the system entering the corresponding sleep state. Note that Guardian prohibits the ACPI sleep event to be triggered by the optional sleep control and PM1b_CNT registers. Specifically, there is a 32-bit pointer in the Fixed ACPI Table (FADT) pointing to the PM1b_CNT block. Guardian clears this pointer and intercepts accesses to the PM1b_CNT register. The same method is used on the control sleep register.

Guardian intercepts the guest's sleep command issued to the PM1a_CNT register. Note that the actual interception method depends on whether the register is accessed by PIO or MMIO. The former involves VMCS configuration whereas the latter requires the EPT.

Among the six Sleep states ($S0$ to $S5$) defined in the ACPI specification (in Figure 3.4), the light-sleep ($S0$ to $S3$) states are not of concern, because the main memory remains powered and Guardian remains alive. Therefore, Guardian performs no action. For the soft-off state ($S5$) where the system will be powered off, Guardian restores the TCB image back to the respective disk locations by using direct disk I/O operations. Note that Guardian needs to re-activate the disk which has

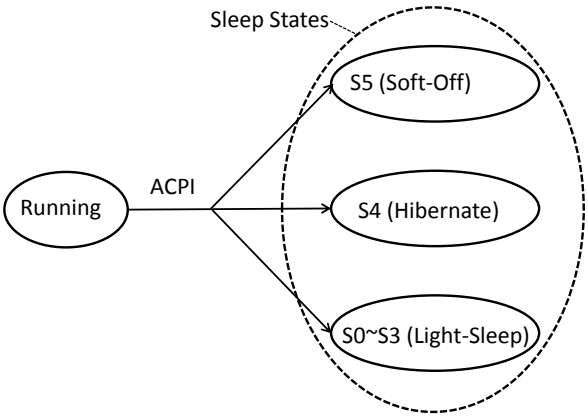


Figure 3.4: ACPI sleep states.

been closed (but remains powered) before the ACIP sleep command is issued. In the end, Guardian clears VMM_flag and resumes the intercepted ACPI command which turns the platform off.

It is slightly more complicated to deal with the hibernation state $S4$ due to the need for platform context saving. Guardian needs to save its context into the hypervisor partition, in addition to the restoration work done for $S5$. For the guest context, Guardian disables and prohibits the ACPI S4BIOS Transition [1], which bypasses Guardian as the BIOS directly saves *all* memory content into the hard disk including Guardian's context. Therefore, only the OS-assisted hibernation method is supported and the OS must write its own context into the disk before hibernation.

Note that after the PM1a_CNT register is set, the platform passes the point of no return, because the ACPI hardware will force the platform to enter $S4$ or $S5$ state and no software will be loaded to the CPU. In other words, Guardian is the last piece of code executed before shutdown, which guarantees the security of the TCB and critical data resting on the disk.

System Reboot. There are three possible ways to reboot a system. One is ACPI reset, which is activated by the ACPI reset register. Note that the system will immediately reboot once the reset register is set. The ACPI reset register can be accessed by port I/O or memory-mapped I/O, which can be intercepted by Guardian through

[1] It clears the F bit in the Firmware ACPI Control Structure (FACS) and intercepts accesses to the SMI_CMD command register, which is S4BIOS service activation.

configuring the VMCS or EPT, respectively. The second way is essentially triggered by the CPU INIT signal. Guardian intercepts the event through configuring the VMCS.

In the third way, an attacker can switch the CPU to the real mode and jump to the BIOS entry to reboot the system. The tricky part is that it can bypass the INIT and ACPI reset mechanisms, meaning that the previous two interception methods will fail to intercept this one. To intercept it, a straightforward solution is to intercept the CPU switch from protected mode to real mode. However, the cost will significantly rise up when legitimate CPU-mode switches take place frequently, e.g., in Windows. Our solution is to prevent jumping to the BIOS reboot-routine from the guest by configuring the EPT. Any attempts from the guest OS to reboot the system will be intercepted by Guardian whose response is to repeat Step 3-5 in secure bootup without rebooting the whole platform.

Recovery

Guardian provides an alternative secure boot mechanism, where the system is able to boot up from a trusted-storage, such as a live CD or a read-only USB token. The bootup sequence is the same as the one described in Section 3.2.1. For convenience, the end-user can configure the system always boot up from a trusted storage, such that the system still can boot up into a trusted state.

The secure shutdown procedure may not be triggered due to some unexpected and irresistible events, e.g., power failure or system crash. Given that such unexpected system failure events may lead to the untrustworthiness of the TCB image, we need the TPM-based secure boot [96] to guarantee that only the trusted image can be booted. In such cases, the system can not boot up, and the security-conscious end-users need the recovery mechanism to restore Guardian image. Specifically, the bootloader in the trusted storage is extended to restore TCB image into the hard drive. Note that the bootloader originally has the capabilities to read/write the hard drive, the trusted storage and the main memory. Therefore, we can easily combine

these functions to do the recovery.

Guardian Update

The end user can upgrade or patch Guardian in a secure environment. Note that the whole system is clean and secure before the guest boots up, because there is no untrusted code executing in the system at that time. Thus, the end user can upgrade/-patch Guardian at that phase (i.e., the secure boot phase). Specifically, during the secure boot phase, the end user activates the BUSUI interface of Guardian (details in Section 3.2.2), where an option is for upgrading or patching Guardian. Following the guidelines, the end user asks Guardian to load a new or patched Guardian image into the system and replace the original copy. After the upgrading/patching step, the system reboots to let the updated Guardian re-control the system.

3.2.2 Secure User-Hypervisor Interface

The secure interface is a duplex channel between the end-user and Guardian without involving the guest OS. Guardian shields the channel against any access from the guest. With the interface, the end-user can configure Guardian during its boot-up, and issue commands during runtime. For the sake of usability and simplicity, we do not rely on any external device such as a USB token. The user inputs are through the keyboard while the outputs are via the display in VGA mode.

Guardian provides two secure UIs. One is the Boot Up Secure User Interface (BUSUI), which is used in the secure boot phase before the guest starts to run. Since the platform then is in a trustworthy state, the implementation of BUSUI is straightforward. Guardian utilizes the BIOS services (i.e. INT 0x16 and 0x10) for input and output. The end-user activates it by holding a special key for a few seconds. In our current design, a user can deposit a text message to Guardian as a shared secret and can also input policies.

The other interface is the Run Time Secure User Interface (RTSUI), which is used after the guest boots up. The RTSUI can be dynamically launched by the end-user. RTSUI extends the secure user interface in KGuard [18]. Namely, Guardian

securely receive inputs of a human user through a keyboard while it securely produces outputs through the display. Both the input and output paths are inaccessible to the guest OS. Since the interface in KGuard is only for password input, we extend it to a command-line interface such that the user can conveniently input commands and read responses.

3.3 Implementation

We have built a prototype of Guardian on a Dell OptiPlex 990 MT desktop with an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz processor[2] and 4GB main memory. Guardian consists of around 25K SLOC for its core functions, which is much smaller than Xen (263K SLOC for Xen-4.1.2) and Linux (8,143k SLOC for Linux-2.6.33.20). A comprehensive comparison between Guardian and other hypervisors is listed in Figure 3.5. Specifically, TrustVisor itself is around 17K SLOC. NOVA [88] consists of the microhypervisor (9K SLOC), and several trusted components, i.e., a thin user level environment (7K SLOC), and the VMM (20K SLOC). BitVisor [84] and VMware ESXi are 194K and 200K SLOC, respectively. KVM is around 200K SLOC, as well as a customized QEMU (140K SLOC). Xen is around 263K SLOC with Dom0 that can be customized to 200K SLOC [47]. Microsoft Hyper-V uses a Xen-like architecture with a hypervisor (around 100K SLOC) and Windows Server 2008 (larger than 400K SLOC).

The binary size of Guardian is around $223KB$, which is much smaller than Xen (around 1,264KB for Xen-4.1.2) and Linux (around 134,134KB for Linux-2.6.33.20) image, and the bootloader core is around $30KB$. Guardian reserves 512KB memory space for TCB images and other critical information. Guardian also provides 11 hypercalls for security services, which is smaller than Xen exported hypercall surfaces (i.e., 46 hypercalls). Note that Guardian only focus on the security services, while these systems (e.g., Xen) usually provide many more

[2]The Hyper-threading mode is disabled since our current hypervisor does not support the multiprocessor mechanism.

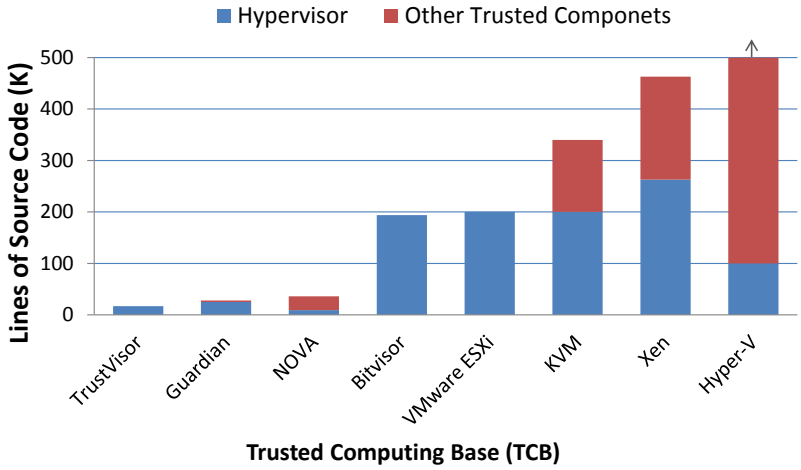


Figure 3.5: TCB size. Larger TCB implies more bugs.

functional services.

3.3.1 System Benchmark

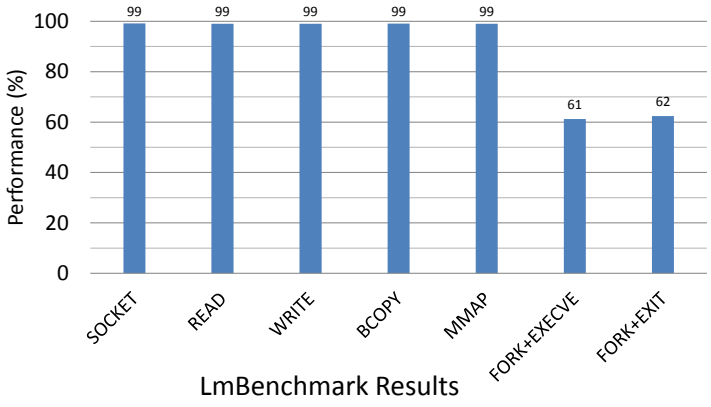


Figure 3.6: The LmBench results on OS operations.

We first measure the overhead on the OS operations using the LmBench suite. Figure 3.6 shows the results: socket (local connection), memory operations (i.e., read, write and bcopy) and some system calls (i.e., mmap, fork+exec and fork+exit). However, fork+exec and fork+exit incur higher performance penalties of 39% and 38%, which are heavily dependent on the Intel EPT performance. We do believe that this could be improved with the performance enhancing of memory virtualization.

We also measure computation performance with Guardian. The results gener-

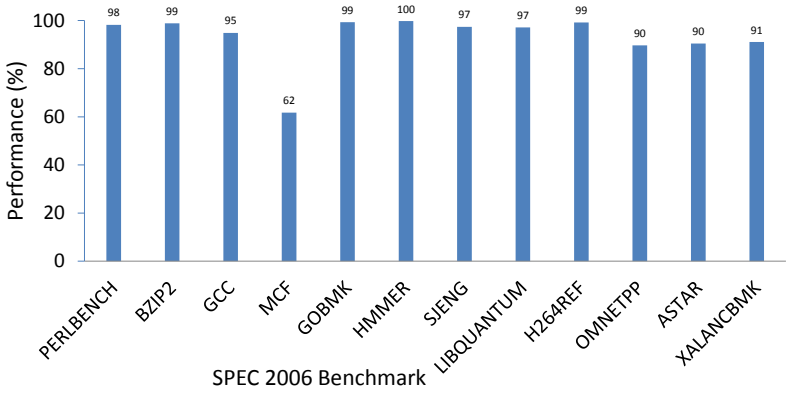


Figure 3.7: The system benchmark comparison results generated by *SPEC CPU 2006*.

ated by the benchmark tool SPEC CPU 2006 (see Figure 3.7) show that Guardian usually only introduces 0.2% - 10.3% performance loss, and may lead to 38.2% performance overhead in some extreme cases (i.e., memory intensive operations with extreme low cache hit rate), which is also dependent on the page operations of current Intel EPT. Again, we believe that it can be improved in the further.

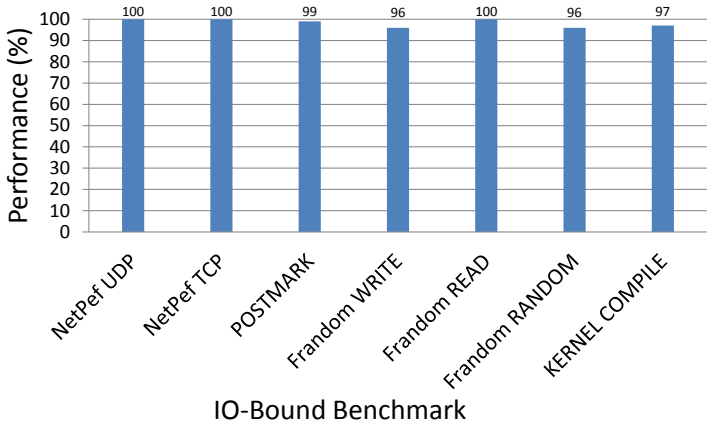


Figure 3.8: The I/O-bound benchmark results.

For I/O-bound benchmark test, we select a range of benchmark tools, including Bonnie, Postmark, Netperf and Linux kernel. For Bonnie, we use a 1GB file and perform sequential read/write (fread/fwrite) and random access (frandom). For Postmark, we choose 20,000 files, 100,000 transactions and 100 subdirectories, as well as all other default parameters. For Netperf, we use another local machine as the Netperf server, and run both TCP STREAM and UDP STREAM benchmarks

to measure basic network performance. For Linux kernel, we compile the Linux-2.6.33.20 with default configuration. Figure 3.8 shows the results.

3.4 Summary

In this chapter, we have proposed Guardian as a security foothold on the end-user systems to enhance their security. Specifically, we introduced Guardian whose integrity and availability were guaranteed by the novel bootup and shutdown technique. Guardian also provided a secure user interface, through which the end-user could update the configurations of Guardian or dynamically activate/deactivate a dedicated security service for the security needs. We have implemented Guardian and the two utilities. The experiment results show that they are efficient and easy to use. Our work demonstrates that computer security can be significantly boosted up by using a tiny and reliable hypervisor.

Chapter 4

Application Protection

In this chapter, we propose AppShield, a novel system which reliably and efficiently protects data secrecy and integrity of a critical application, as well as its execution integrity, against rootkit attacks. AppShield leverages the hardware-assisted virtualization techniques [46] to isolate the application's address space such that all accesses from the kernel are blocked except those explicitly authorized by the application through system calls. The protected application utilizes the main memory in the same fashion as in a normal (unprotected) setting. It accesses the memory with native speed, i.e. without encryption/decryption or being intercepted, and it can request the kernel to (de)allocate memory buffers. AppShield achieves performance isolation since those unprotected applications are not affected and do not have performance loss. We have implemented a prototype of AppShield which consists of a bare-metal hypervisor with roughly $29K$ SLOC and a tiny kernel module of around $2K$ SLOC. We have experimented the prototype with several applications (e.g., Apache and VIM) and run a suite of benchmark tests. The experiment results demonstrate that AppShield incurs insignificant performance costs in CPU computation, disk I/O and network I/O.

ORGANIZATION. In the next section, we define the problem by specifying the threat model, our objectives and the AppShield overview. In Section 4.2, we describe the dynamic address space isolation together with newly identified threats.

The secure and efficient address space switch, and the support of legal data exchanges is described in Section 4.3 and Section 4.4, respectively. The implementation and evaluation are shown in Section 4.5. Finally, we discuss several issues in Section 4.6, and conclude this chapter in Section 4.7.

4.1 Synopsis

4.1.1 Threat Model

In this work, we consider an adversary who remotely controls the OS on the target platform by a rootkit and attempts to attack a critical application by tampering with its data and/or execution. The adversary can run arbitrary code and launch DMA operations in the victim platform. Nonetheless the adversary can not physically control it.

Our aim is to protect a critical application execution integrity and data security against such an adversary. We do not consider protection of its availability. Neither do we protect the application's raw I/O inputs and final data outputs[1]. Side channel attacks are also out of scope of our study.

We suppose that no malicious data input can subvert the control flow of the critical application. It is *orthogonal* to our objectives to enhance code security (e.g., fixing bugs) of the protected applications. The platform's chipset and all peripheral devices are trusted in the sense that they operate exactly following their specifications and do not contain Trojan-Horse circuits or microcode that respond to commands of the adversary.

In our model, a bare-metal hypervisor is trusted since it can be protected with secure boot/DRTM (e.g., Intel TXT) and hardware virtualization technology. Furthermore, the hypervisor can leverage some existing hypervisor protection schemes (e.g., HyperSafe [99], HyperSentry [5], HyperCheck [98]) to further enhance its security. Note that the hypervisor can intercept and emulate the SMM operations so

[1]The critical application may encrypt its disk and network data. Existing secure I/O path schemes like [19, 109] can protect the raw I/O inputs

that SMM-based attacks cannot subvert it.

4.1.2 Desired Properties

It is desirable for a security solution for the stated problem to have the following properties. Firstly, the application's behavior should be preserved by the protection mechanism. The application is not assumed to be a piece of self-contained code and is entitled to issue system calls as in a normal setting. For instance, it can request the OS to allocate a memory buffer even though the OS is not trusted.

Secondly, the security mechanism should have minimum performance impact on the protected application and on the platform as a whole. The performance requirement has twofold implications. Ideally, the protected application should be able to access the main memory with the native speed. Therefore, hypervisor-based interposition and memory buffer encryption/decryption should be avoided since they take a significant toll on memory access delays. Moreover, the mechanism should only incur localized performance overhead, without affecting the performance of unprotected applications and the OS.

Lastly, the TCB of the security mechanism should small and simple, which ensures that the risk of subverting the TCB is kept minimal. This property precludes the approach of using a trusted virtual machine where the TCB encloses an operating system.

In this chapter, we present the design and implementation of AppShield which is the first of its kind meeting the security requirement with all the aforementioned properties. AppShield uses a tiny hypervisor on the bare-metal machine to protect a critical application against the untrusted OS. Its overview is described in the next section.

4.1.3 AppShield Overview

The fundamental idea of AppShield is to isolate the target application's context (registers) and address space from the kernel and other applications, while allowing

it to issue system calls and utilize the memory in a dynamic fashion. The rootkit cannot access its memory space, except those memory buffers explicitly exported to the kernel by its system calls. In the rest of the chapter, we use CAP to denote the critical application that is under AppShield's protection.

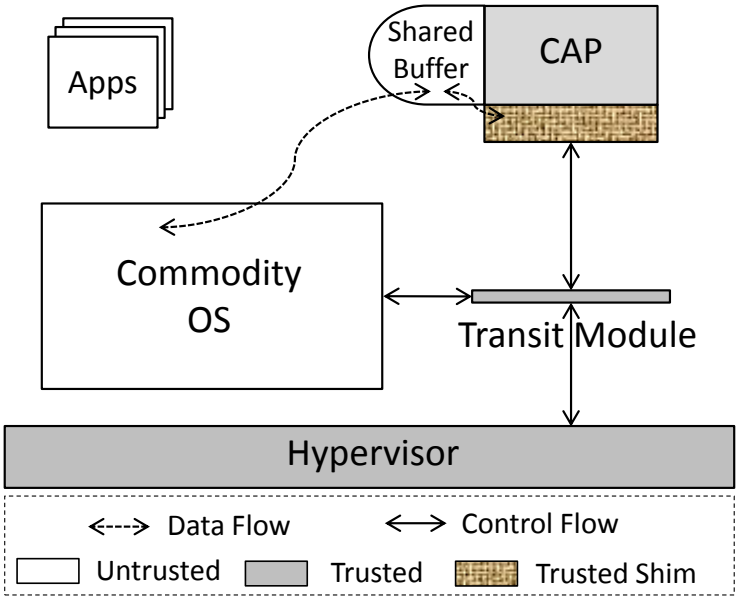


Figure 4.1: The architecture of AppShield. The data flows (dotted lines) between the protected Critical APplication (CAP) always go through the shared buffer and mediated by the *shim* code. The control flows (solid lines) between CAP and the OS are mediated by the Transit Module (TraMod). The execution of transit module are protected by the hypervisor.

Figure 7.1 depicts the architecture of AppShield. It consists of a bare-metal hypervisor, a *transit module* in the guest kernel space and a *shim code* in the user space. Both the transit module and the shim code are safeguarded by the hypervisor to defend against attacks from the guest kernel. CAP runs in an address space isolated from the rest of the guest domain, while the guest OS and those unprotected applications on the platform run unaffected. The page table of CAP is managed by the guest OS, but the updates are intercepted and verified by the hypervisor to defend against various attacks (Section 4.2.2). CAP's system calls are mediated by the trusted shim code which is essentially a wrapper of `libc` libraries. The main task of the shim is to marshal the system call parameters by exporting the data needed by the system call routine into the shared buffer accessible to the kernel.

Since events like interrupts and system calls causes context switches between CAP and the guest kernel, the transit module responds to the event, facilitates the context switches, and prevents the context switch from being manipulated by the rootkit.

4.2 Dynamic Address Space Isolation

Dynamic address space isolation is the bedrock of AppShield. In this section, we first elaborate how the hypervisor isolates a pre-defined address space of CAP. Then, we explain how the isolation is dynamically adapted to the changes of the memory boundary at runtime. While our description follows Intel virtualization technology, the approach is applicable with AMD's as well.

4.2.1 Address Space Isolation

In a nutshell, the physical memory assigned to the guest is divided into two separated regions by the hypervisor. One region is used for CAP while the other is for the guest OS and other applications. The memory dichotomy as depicted in Figure 4.2 is realized by two suites of EPTs maintained by the hypervisor, respectively. In this way, the virtual addresses of the guest OS and other applications are never mapped to a physical address dedicated to the protected application, and vice versa. The hardware enforced address space isolation ensures that the guest OS and the protected application cannot directly access each other, provided that all EPTs are properly set and applied. For the sake of clarification, we use *AppShield EPT* to refer to the ones dedicated for CAP. In the following, we only focus on the EPT configuration. The details of applying the proper EPT are described in Section 4.3 which elaborates the context switches between CAP and the guest OS.

The hypervisor exports two hypercalls for CAP to activate and deactivate the protection. The activation hypercall is issued before CAP's main function is entered. In response, the hypervisor obtains the `CR3` register value from the VMCS and traverses the page table entries belonging to the application, so that it locates all pages within the address space, including the shared libraries. (Note that the

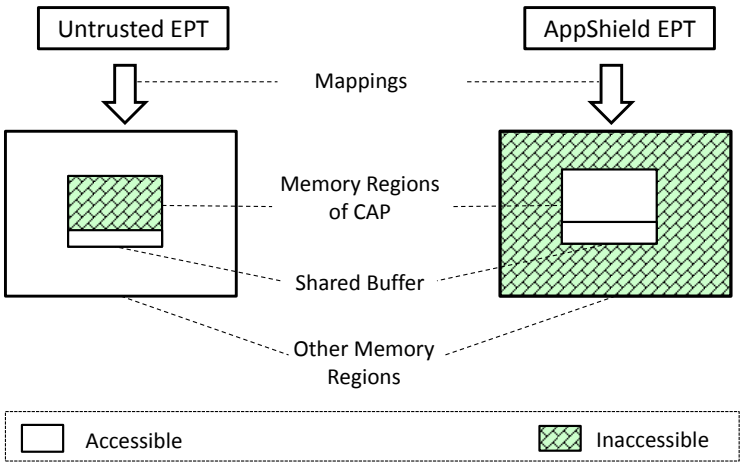


Figure 4.2: Address Space Isolation. With the trusted AppShield EPT, only the memory regions of CAP and the shared buffer are accessible, while with the original EPT, the memory regions except the shared buffer are inaccessible.

page fetching during the traversal forces the guest kernel to load share libraries into the memory.) Both the traversed guest PTEs and the pages pointed by them constitute the physical memory region that needs to be separated from the guest. The hypervisor creates the AppShield EPT for this region and marks the corresponding entires in the original EPT as inaccessible, so that the guest cannot visit the isolated region. Once the application's code and data are isolated, the hypervisor can validate its launch-time integrity, supposing that it has been priorly authenticated by a signature or an HMAC tag.

Through the deactivation hypercall, CAP notifies the hypervisor to disable the protection. In response, the hypervisor destroys the AppShield EPT and restores the entries in the original EPTs. Note that the deactivation hypercall can only be issued by CAP. Any deactivation requests from malicious guest OS and other unprotected applications will be rejected by the hypervisor.

4.2.2 Dynamic Isolation

One of the main challenges of isolating a full-fledge application is that its memory region evolves over time, due to dynamic memory allocation and deallocation as a result of relevant system calls (e.g., *brk*) which are in turn invoked by the corre-

sponding memory usage functions (e.g., *malloc* and *free*) in the `libc` library.

The semantics of these system calls are preserved in AppShield as the guest OS still manages the memory resources for CAP through the guest page table. Although the hypervisor protects the guest page table used by CAP, the guest kernel may manipulate the virtual and/or physical address of the new buffer to attack CAP without direct access to the latter's memory space. We identify several such attacks below.

Address Manipulation

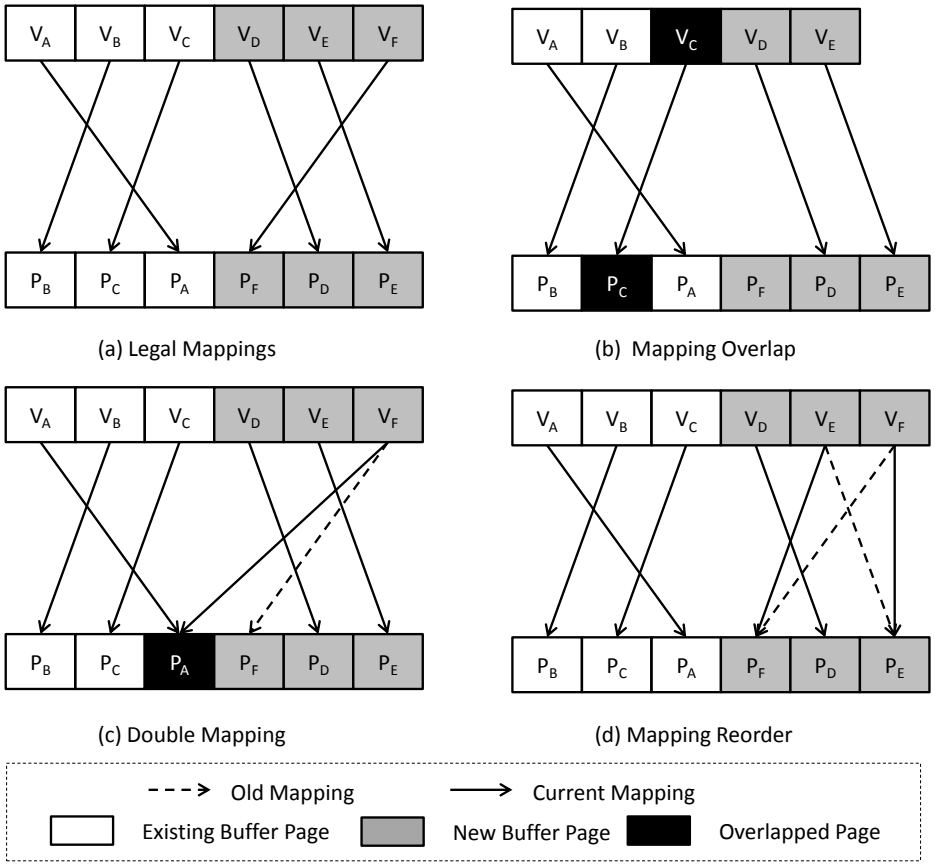


Figure 4.3: Threats for address space isolation.

In general, address manipulation attacks can be launched by the kernel in response to any system calls that result in page table updates. Without loss of generality, we use buffer allocation as an example to illustrate the attacks.

Suppose a CAP's buffer contains three consecutive pages at virtual address

V_A, V_B and V_C respectively and CAP requests a new buffer. In a normal scenario, the newly allocated buffer's virtual address and physical addresses do not overlap with any existing one, as illustrated in Figure 4.3-(a), where they are at virtual address V_D, V_E and V_F. In the following, we show four types of manipulation attacks.

Mapping Overlap Attack. The malicious kernel may overlap two memory regions in the virtual address space. As illustrated in Figure 4.3-(b), the new buffer is set to the pages located at V_C to V_E. The overlapping leads to undesired modifications of data in P_C when the application attempts to update the first page of the allocated buffer. Obviously this threat could break the data integrity, and it may also subvert the control flow of the application, if the overlapping memory is in the application stack and the modifications change the stored return address(es). In fact, the mapping overlapping is a type of Iago attack [15].

Double Mapping Attack. The double-mapping attack maps two or more virtual pages to one physical page in the user space. As shown in Figure 4.3-(c), a write to V_A affects the result of a read operation at V_F. This attack is more stealthy than the mapping-overlap attack, as the physical addresses are transparent to the code running in the virtual space which is not tampered with at all.

Mapping Reorder Attack. The mapping-reorder attack is to reorder the existing address mappings between the virtual addresses and the physical addresses. As shown in Figure 4.3-(d), CAP retrieves wrong data when it reads from V_F. As a result, CAP's data or control flow can be manipulated by the malicious kernel.

Mapping Release Attack. The mapping-release attack is to release one or more existing mappings without any system call requests driven by the protected application. The mapping-release could induce the hypervisor to give up the protection of those pages since they are not in the protected addresses space any more. By doing so, the guest OS can freely access the data on those released pages.

Information Collection

Most applications and shared libraries trust OS by default, and they all miss the verification of the OS behaviors in the memory allocation and deallocation. To fix this loophole, AppShield has to verify if the memory updates follow the requests of the application and are not manipulated.

To verify the OS behaviors in memory updates, we should know the existing memory layout, determine the intent of the application relevant to memory updates and interpret the page table updates operated by the untrusted OS. The existing memory layout (the mapping relationship between guest virtual addresses to guest physical addresses) can be collected from the guest page table of the CAP. The collected information is reliable since it is collected by the hypervisor and the page table has been protected to prevent any update.

To determine the intent of the application relevant to memory updates, one possible way is to allow the hypervisor to intercept all system calls that are potentially used by the CAP to allocate or deallocate memory. In order to correctly interpret the memory updates information (i.e., the based address and the size), the hypervisor has to know the exact semantic meaning of all parameters and return values. It inevitably increases the complexity of the hypervisor and thereby dampens its security. In this chapter, the trusted shim running in the user space closely works with the CAP. Thus, it knows the system calls used by the CAP and their semantic meanings, e.g., the parameter of the *malloc* is the memory size and the return value is the based address of the new allocated buffer. Through several hypercalls, the trusted shim is able to securely synchronize such information with the hypervisor.

To intercept and interpret page table updates, one possible solution is to allocate a dedicated Guest Page Table (GPT) in the address space of the CAP, and the hypervisor or transit module manages its updates. By doing so, the security of the page table is guaranteed but the complexity of the hypervisor or transit module will dramatically increase, which further takes a toll on the overall system security. An-

other possible way is the paraverfication technique [41]. However it requires costly modifications of OS code. To achieve good compatibility and make the hypervisor and transit module small and simple, we choose the solution that is similar to the management of the page table in paravirtualization, e.g., Xen [6]. Specifically, the page table created by the untrusted OS for the CAP is set read-only, but the management is still handled by the untrusted OS. The updates referring to the CAP memory regions are intercepted and verified by the hypervisor. Note that during the validation procedure, the hypervisor gets the original and the new value of the page table slot, together with the virtual address according to the slot position. Considering the updated address and comparing the original value with the new one, the hypervisor obtains the meaning of the current update, and thereby can validate the page table update.

Verification Details

In page table update verification, the hypervisor and the shim code jointly enforce the following policies for protecting the address space of a CAP.

1. The page table of CAP should be non-writable for the untrusted guest OS. Any update should be intercepted by the hypervisor.

2. The newly added memory region should not conflict/overlap with any existing memory regions, no matter the conflicts happen in virtual address space (no mapping overlap) or physical address space (no double mapping).

3. Once the mappings between guest virtual addresses and guest physical addresses are fixed, they are not allowed to re-map (e.g., no mapping reorder).

4. The memory regions can be released only if the CAP requires to release them (no malicious release), and the page data should be cleaned before allowing the guest OS to manage/access it (no data leakage).

Essentially, the mapping overlap attack is the conflicts in the virtual address space. Thus, the verification algorithm can be put into the trusted shim, since it is

protected and aware of virtual addresses. Specifically, the trusted shim is able to know all memory regions used by the CAP by collecting such information in the memory-related system calls. For example, the trusted shim can know the size of the memory-mapped region through the second parameter of *mmap* and the base address through the return value. Such information stored in an ordered list is inaccessible for the untrusted guest OS since the address space of the CAP is isolated by the hypervisor. For each new allocated memory region, the trusted shim verifies it with existing ones. If there is no overlap, it then updates the maintained list and passes the execution flow to the CAP; otherwise it will issue a hypercall to the hypervisor to inform the policy violation.

To defend against double mapping and mapping reorder attacks in the page table updates, the hypervisor has to interpret the old mapping M_o and the new mapping M_n, and analyze the intent of this update. If the guest OS is to build a new mapping (i.e., M_o is empty and M_n points to a guest physical page), the hypervisor verifies if the new pointed physical page is occupied before. If it is already occupied, it is a double mapping attack; otherwise the update is approved. If the guest OS aims to remap/reorder the mappings (i.e., both M_n and M_o point to guest physical pages), the hypervisor directly rejects it.

If the guest OS aims to free an old mapping (i.e., M_o points to a guest physical page while M_n is empty), the hypervisor verifies if CAP requires the guest OS to release this memory page. The information about the released memory pages is provided by the trusted shim through hypercalls. Those potentially released memory pages are stored in a list in the hypervisor space. By searching the list, the hypervisor decides if the current page is the one that CAP aims to release. If it is not, the hypervisor rejects the update; otherwise it approves it and updates the list by deleting the corresponding record. Note that the data on the releases memory page is cleaned by the trusted shim once it gets the release requests from the CAP.

4.3 Secure Context Switch

Events like system calls, interrupts and exceptions, lead to context switches between CAP and the kernel. Different from traditional user-kernel context switch, the switch between CAP and the kernel involves address space switches, since they run in two address spaces.

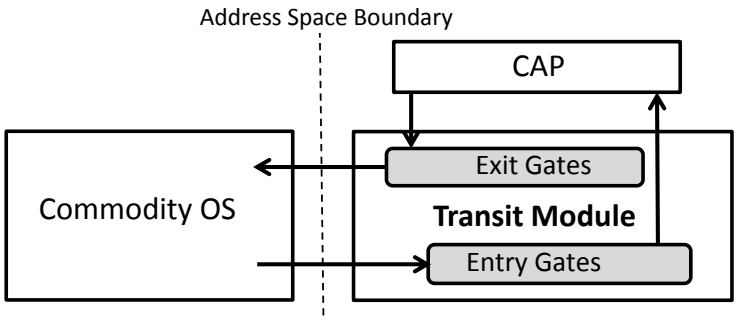


Figure 4.4: Control flow between the CAP and the guest kernel.

When CAP is in execution, the transit module in AppShield handles all interrupts and prevents the kernel from exploiting the context switch to attack CAP. Its main tasks are to facilitate the context switch and to safeguard CAP's context information. It also notifies the hypervisor to perform address space switch. As shown in Figure 4.4, when an interrupt is raised, the control flow leaves from CAP to the kernel. Once the event is processed by the kernel, the flow goes back to CAP. We proceed to elaborate the details of context switch.

4.3.1 Transit Module

The transit module is a self-contained kernel module with its execution being protected by the hypervisor using the mechanism described in [82]. Specifically, the memory regions occupied by the transit module is isolated by the hypervisor, such that the untrusted commodity OS can not modify the data and the code. The control flows of the transit module *always* start from the pre-defined addresses.

The transit module has two sections (Figure 4.5), which are page aligned for facilitating memory protection. The first section is the *public section* which contains

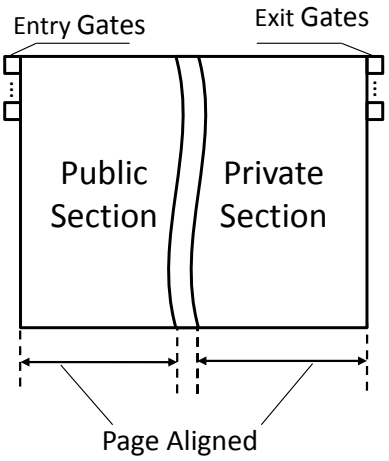


Figure 4.5: The format of transit module

information that is read-only for the transit module and the commodity OS. The second section is the *private section* which contains private data. Accesses to the private section are only allowed if they are from the transit module; other accesses originated from outside of the transit module are blocked by the hypervisor.

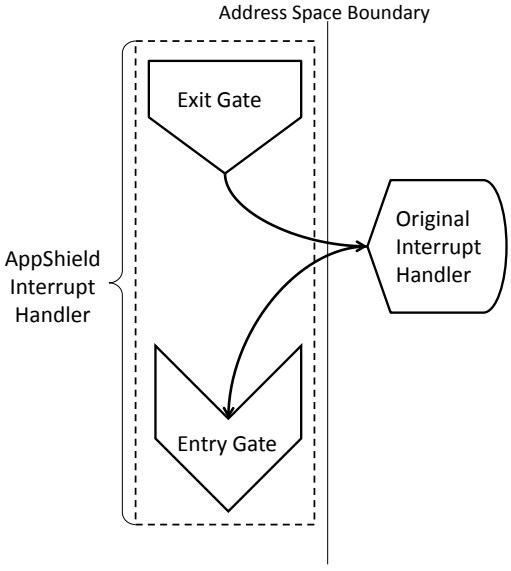


Figure 4.6: The AppShield interrupt handler.

The transit module consists of a set of interrupt handlers called *AppShield interrupt handlers*. An AppShield interrupt handler is composed of two code stubs (Figure 4.6). One is called the *entry gate* which is located in the public section and the other is called the *exit gate* which is in the private section. The control flow of

the transit module always starts from one of the entry/exit gates. The exit gate handles the context switch from CAP in protection to the guest kernel while the entry gate handles the switch back to CAP. More details are presented in Section 4.3.3.

4.3.2 Event Capture

We do not use the hypervisor to intercept the interrupt events as this method significantly affects the platform performance. When AppShield is activated, the hypervisor loads the AppShield Interrupt Descriptor Table (IDT) which is dedicated to CAP under protection, and protects it from be modified by the guest kernel by setting its region as read-only.

The AppShield IDT contains the pointers pointing to the AppShield interrupt handlers. The hypervisor installs the AppShield IDT to the CPU occupied by CAP by setting its IDTR register. Consequently, the AppShield interrupt handlers become the first responders to interrupts on the CPU. When the guest OS is running, it uses the original IDT and interrupt handlers. The switch of the two IDTs follows the switch of the address space. As illustrated in Figure 4.7, the original IDT is uninstalled and the secure IDT is installed for the CAP execution.

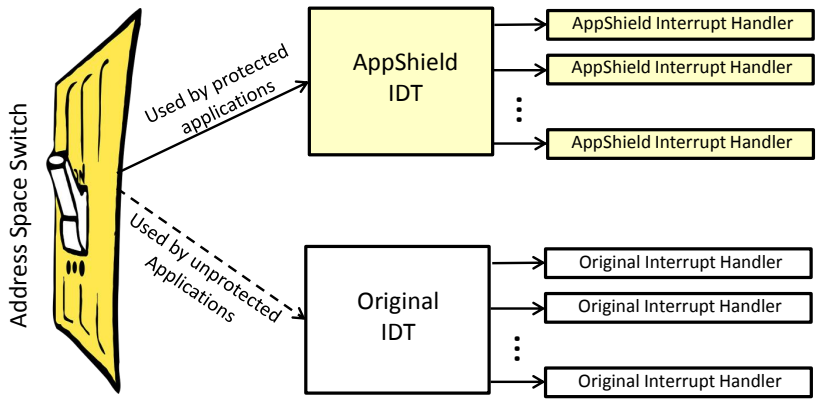


Figure 4.7: Performance Overhead Localization. When the context switches to CAP, the normal IDT is uninstalled and the secure IDT is installed.

By using two sets of interrupt handlers, our design achieves performance overhead localization, because the transit module is only invoked when CAP is interrupted. AppShield is not involved when other applications and the guest OS are

running.

4.3.3 Context and Address Space Switch

Figure 4.8 depicts the control flow of event handling with two context switches at the exit gate and the entry gate. When an interrupt is raised during CAP's execution, the exit gate of the AppShield interrupt handler kicks off the context switch. Under the protection of the hypervisor, the exit gate first prepares a buffer and saves CAP's context in the transit module's private section. It then creates a dummy context for the kernel to execute within. Note that the dummy context should *not* be randomly generated since some context information is used by the kernel to serve the application. For instance, the EIP should point to the corresponding interrupt handler so that the original handler can serve the interrupt. Specifically, we only need to hide the information in the general registers (i.e., EAX, EBX, ECX, EDX, ESI, EDI, EBP) since they may contain sensitive CAP data. In the case of system call context switch, we also need to keep the parameters in the corresponding registers. Moreover, to allow the execution flow to come back to the transit module, the return address of the dummy context is set to point to the corresponding entry gate. In the end, the exit gate then issues a hypercall to inform the hypervisor to restore the original page tables so that the interrupt handler in the guest kernel can properly execute.

Once the guest interrupt handler finishes its process, the control is returned to the entry gate. The entry gate issues a hypercall to request the hypervisor to restore the AppShield EPT. After ensuring that the request is indeed from the legitimate entry gate, the hypervisor restores the AppShield EPT and installs the AppShield IDT, so that the entry gate can properly restore the saved context and resume the interrupted CAP execution.

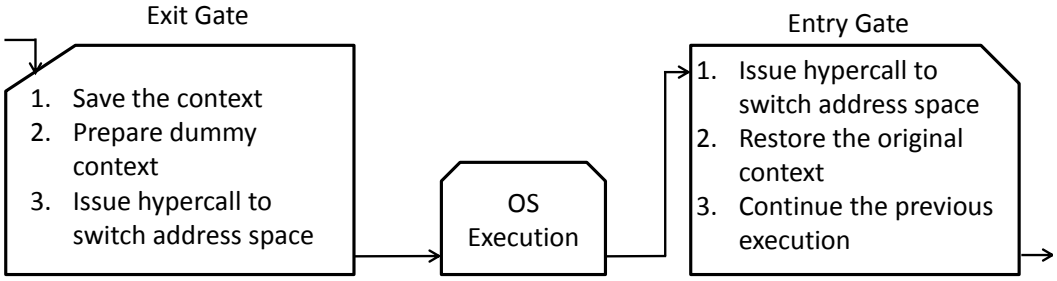


Figure 4.8: A typical address space switch always starts with an exit gate and ends with an entry gate. The commodity OS handles the events that trigger the address space switch.

4.3.4 Special Considerations

Fast-System-Call Cost Localization

The platforms equipped with new processor and chipset support fast system call mechanisms (i.e., SYSENTER/SYSCALL, SYSEXIT/SYSRET), which are independently proposed by Intel and AMD, respectively. The SYSENTRER/SYSCALL traps the CPU to the kernel mode and the SYSEXIT/SYSRET transfers the CPU back to the user mode. In this chapter we choose one pair (i.e., SYSENTER/SYSEXIT) to illustrate.

The $SYSENTER$ instruction sets the registers (i.e., CS, EIP, SS and ESP) according to values specified by the operating system in certain Model-Specific Registers (MSR), and triggers the CPU to trap into the kernel mode. To localize the performance overhead to CAP, the hypervisor also prepares two sets of MSR registers. One set is used for CAP, where the EIP value in the corresponding MSR (i.e., *SYSENTER_EIP_MSR*) is modified to point to the new system call handler prepared by the transit module. By doing so, all fast system calls will be intercepted by the transit module. Another set is used for the unprotected applications as usual. The two sets of registers are switched following the switches of the address spaces. Note that the context backup and restoration are still handled by the pairs of the exit and entry gates.

Multi-Thread Execution

AppShield supports multi-thread execution of CAP. The child threads could be user threads, which are completely maintained by CAP in user space, or light weight processes scheduled by the guest OS and sharing the same address space with their parent.

The user threads do not have their own contexts since they do not have the kernel structure for scheduling. Therefore, they are transparent to AppShield. In contrast, light weight process threads may have multiple user contexts for CAP, since each of them has its own corresponding structures (e.g., kernel stack) for scheduling. These threads may run in parallel and trap into the guest OS simultaneously. Therefore, by using the base addresses of their kernel stacks as the identifiers, the transit module can distinguish each of them, and save/restore the respective contexts.

4.4 System Call Adaption

The system call from CAP to the guest kernel exposes some CAP data since they are passed to the guest as parameters. AppShield provides a *spatio-temporal* protection [22] for the data involved in the system call. It ensures that the guest OS can only access the authorized data (spatial protection) during the execution of the system call (temporal protection). The previous sections have explained that temporal protection is achieved by address space isolation and secure context switch. In this section, we describe how AppShield enforces spatial protection through system call adaption.

A majority of system calls do not need the OS to access the application address space to get further information, since all needed information in the parameters (e.g., *close*) or even without needing any parameters (e.g., *getpid*). These calls are passed to the guest OS without any adaption.

Those system calls whose parameters contain pointers (e.g., a pointer pointing to the file name in *open*), need adapting. To ensure spatial protection, researchers

have proposed two possible solutions. The first approach [17] is to interact with the hypervisor multiple rounds to safely move the *decrypted* data into a shared/public buffer. The other approach [22] does not allocate a new buffer. Instead, it decrypts the data in the original buffer and allows the OS to directly access the buffer. Both solutions use expensive cryptographic technique, which dramatically reduces the application performance if the application frequently issues system calls. In addition, the multiple round interaction (with the hypervisor) is another source of the performance loss. We summarize the time cost of the parameter marshalling in a system call in two typical schemes (i.e., Overshadow and SecureME) together with our scheme in Table 7.1. Overshadow needs 8 context switches and costly cryptographic operations, and SecureME also needs cryptographic operations together with 2 context switches. For our scheme, we only need 2 context switches for one system call.

	Crypto. Operations	Data Movement	Context Switch (#)
OverShadow	$\surd$	$\surd$	8
SecureME	$\surd$	$\surd$	2
AppShield	X	$\surd$	2

Table 4.1: The time cost of the parameter marshalling in a system call. Our scheme is relatively efficient because we give up the costly cryptographic operations and reduce the switch times.

In our scheme, the trusted shim creates a shared region (buffer) in its user space, and issues a hypercall to inform the hypervisor that the shared region is accessible for the guest OS. In this way, the guest OS can only access the data within the shared region, but cannot access any other regions within the user space of the CAP, achieving the spatial protection.

To adapt system calls, the shim developers should understand the semantic meaning of each system call. Specifically, they should know the meaning of the parameters and the return values. In addition, they should know the direction of the data exchange, e.g., from application perspective, the buffer referred to by the parameter is for receiving data from the guest OS, or caching the data that will be sent

out. Getting such semantic information, they are ready for the system call adaption. Specifically, for the data that the CAP attempts to send out, the shim simply moves the data into a buffer allocated in the shared region, and updates the corresponding parameter to refer to the new buffer. To receive data from the guest OS, the shim should reserve a buffer in the shared region. The shim then saves the base address of the original buffer, and updates the corresponding parameter to refer to the reserved one. When the system call returns, the shim copies the received data into the original buffer and continues the execution.

4.4.1 System Call Emulation

There are several system calls whereby the system call adaption technique is not applicable to resolve the conflicts between the system call purpose and our security requirements. Specifically, such system calls are not designed for exchanging data. Instead, they are for introspecting or manipulating the application by accessing or modifying internal status.

The first is *Futex* (i.e., fast user mutex), which provides a method for an application to wait for a value at a given address, and a method to wake up other applications waiting on a particular address. The implementation of Futex not only directly accesses the process memory, but also binds some information (e.g., a hash bucket) with the address. Therefore, if we simply apply the system call adapting technique to Futex, the semantic information may be bound to a wrong address, which may lead to the failure of Futex.

The rest are the system calls used in the *signal-handling*, where the guest OS needs to prepare a temporary execution context for the application and transfers the execution control to a pre-registered handler to handle the corresponding signal. The critical security issue here is that the guest OS needs to be authorized to manipulate the application context. Such authorization may be exploited to reveal and tamper with the application data, e.g., involve a function to send plain text outside. To revoke the authorization from the guest OS, we have to emulate it.

Configurations	Descriptions
CPU	Intel i7-2600 with 3.40GHZ
Memory	3GB DDR3 1333MHZ
Network Card	Intel Device 1502 with 1Gbps
Disk	ATA 7200RPM
OS	Ubuntu 10.04 with Kernel 2.6.32.59

Table 4.2: The configurations of the experiment machine.

4.4.2 Ptrace

The *ptrace* system call is not allowed for CAP since its working mechanism requires the guest OS to directly read the content of the user space, or to modify the data or even code of the specific addresses, which is *completely* conflicts with our security requirements. We should not emulate this system call since it opens the door for the malicious guest OS to read/write the whole address space of the CAP.

4.5 Implementation and Evaluation

We have implemented a prototype of AppShield on a PC whose specification is listed in Table 4.2. The prototype consists of a dedicated hypervisor running on the bare-metal hardware, and a Linux loadable module as the transit module. The code base of the hypervisor is around $29K$ SLOC with $218KB$ binary size. The transit module consists of around $2K$ SLOC, and the trusted shim is around $1K$ SLOC.

Trusted Shim

We do not modify the source code of the application and the shared libraries, instead we create the shim as a wrapper of libc, and allow it to intercept the function calls that are supposed to call the libc functions. Specifically, on the Linux system, an application usually needs shared libraries at runtime, and the dynamic linker loads those shared libraries in whatever order it needs them. However, when you set $LD\_PRELOAD$ to a shared library, that file will be loaded before any other libraries, including the libc library. Preloading a library means that its functions will be used before others of the same name in later libraries, allowing a function to be

System Calls	
Files	open, close, read, write, chdir writev, access, fstat64, uname, poll, fcntl statfs64, fstatfs64, getdents64, getdents stat64, lseek, _llseek, getcwd, fchdir, ioctl
Network	bind, listen, accept, sendto, recvfrom, accept4, select connect, send, recv, getsockname
Memory	mmap2, munmap, mremap, brk, mprotect
Process	getpid, gettid, getgroups32, set_thread_area getuid, geteuid, getgid, getegid exit_groud, tgkill, getrlimit, exit
Time	time, clock_gettime, gettimeofday
Others	futex, rt_sigaction, rt_sigprocmask, sigaltstack

Table 4.3: Supported system calls.

intercepted. We use this feature in our implementation, saving the cost of the source code modification.

The trusted shim needs to do some initializations and preparations for the protection and the interception, such as allocating the shared buffer, and informing the hypervisor to protected the application. However, those functions for intercepting system calls are passively invoked, meaning those functions will not execute until the application explicitly call them. To solve this problem, we resort to another feature - constructor function. A constructor function marked with *.init* will be called by the dynamic linker when the library is loaded. The trusted shim supports 56 system calls (listed in Table 4.3) in the current implementation.

We evaluate the impacts of AppShield by running both macro- and micro-benchmark kits.

4.5.1 Micro Benchmark

In the micro benchmark, we evaluate the cost of the address space switch (Table 4.4). An address space switch event can be divided into three parts: protection mode switch, context backup and restoration. The protection mode switch includes a hypercall, IDTR and EPT switching. The context backup consists of saving 17 registers (including general, flag and control registers) and creating a dummy con-

Operation	Time (μs)
Out of Protected Address Space	1.72
Back to Protected Address Space	1.33
Context Backup	0.11
Context Restoration	0.08

Table 4.4: The micro-benchmark results for address space switch.

text. The context restoration is to load all the saved registers. The cost of domain switch is relatively high, because it contains the costly memory access from hypervisor space to guest space, i.e., inserting the return address to the kernel stack. All three costs constitute the latency for the system to handle a particular interrupt or exception. The cost for a system call is for address-space switch cost and parameter marshalingt. The parameter marshaling cost varies for different system calls. For instance, there is no such cost for *getpid*, while we need it to copy data from user space to the shared region in *write*. Thus, we do not measure them individually, but choose to evaluate the whole application performance overhead in macro benchmark.

4.5.2 Macro Benchmark

AppShield Impacts on Performance

SPEC CINT2006 [25] is an industry-standard benchmark intended for measuring the performance of the CPU and memory. We executed SPEC CINT2006 in two setups: system with virtualization, and the system with AppShield. Figure 4.9 shows the results.

Comparing the impact of running the workload in a system with a bare-metal hypervisor, we calculate the overhead added by the additional virtualization layer. Based on the virtualization impacts, AppShield imposes an additional 0.01% slowdown on average. The primary source of virtualization overhead is VM exits due to interrupts and privileged instructions [35].

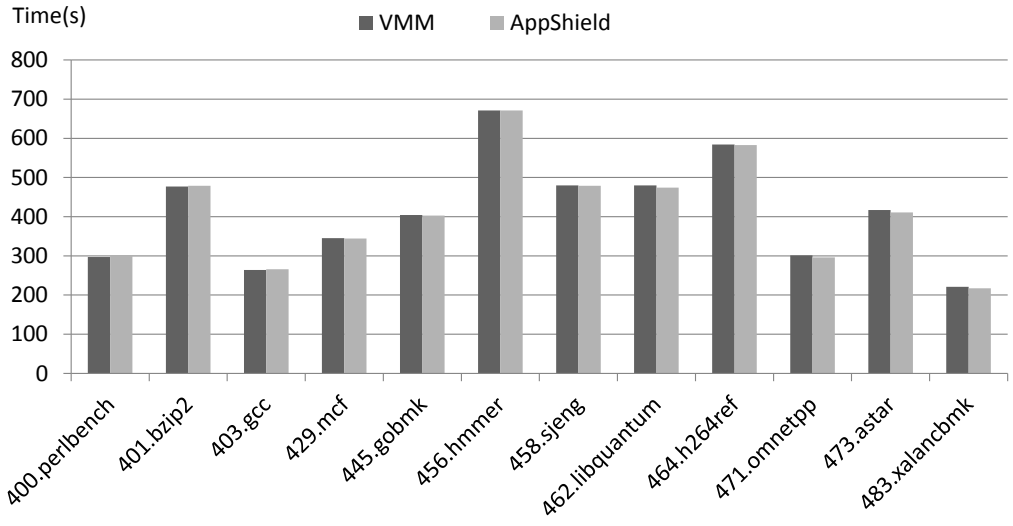


Figure 4.9: SPECint 2006 Result. AppShield introduces insignificant slowdown comparing with virtualization.

Computation Effects

We measure the AppShield protection on computation programs. In our experiment, we measure three encryption algorithms (i.e., AES, RC4 and RSA), which is adopted from *OpenSSL 0.9.8k* project. We run these algorithms to encrypt/decrypt messages with different lengths, from 32bytes to 2048 bytes. The measurement results in Figure 4.10 show that the protection effects on the computation programs is quite small.

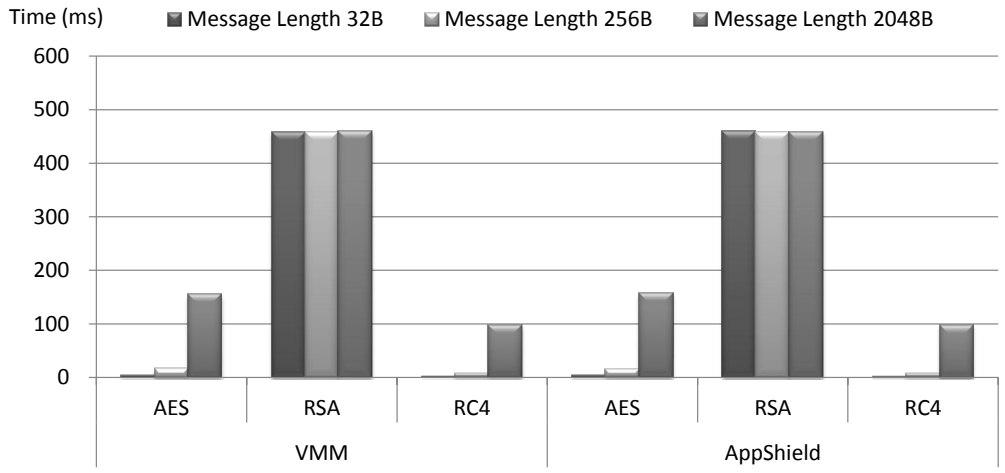


Figure 4.10: The effects of AppShield protection on computation.

Disk I/O Benchmark

The disk I/O benchmark includes three sub-benchmarks to evaluate the overhead in disk reading, writing and copying. Disk I/O benchmark reads/writes data from/to files with different sizes. In our experiments, the file size is 64MB, and the read-/write granularity is from 512B to 4MB. Experiments with a larger file and a smaller buffer result in more system calls, and consequently introduce more context switches. However, with the increasing of the buffer size, the performance is better, which is also proved by the experiment results in Figure 4.11. Note that the overhead is mainly introduced by data copy and context backup/restoration in parameter marshaling.

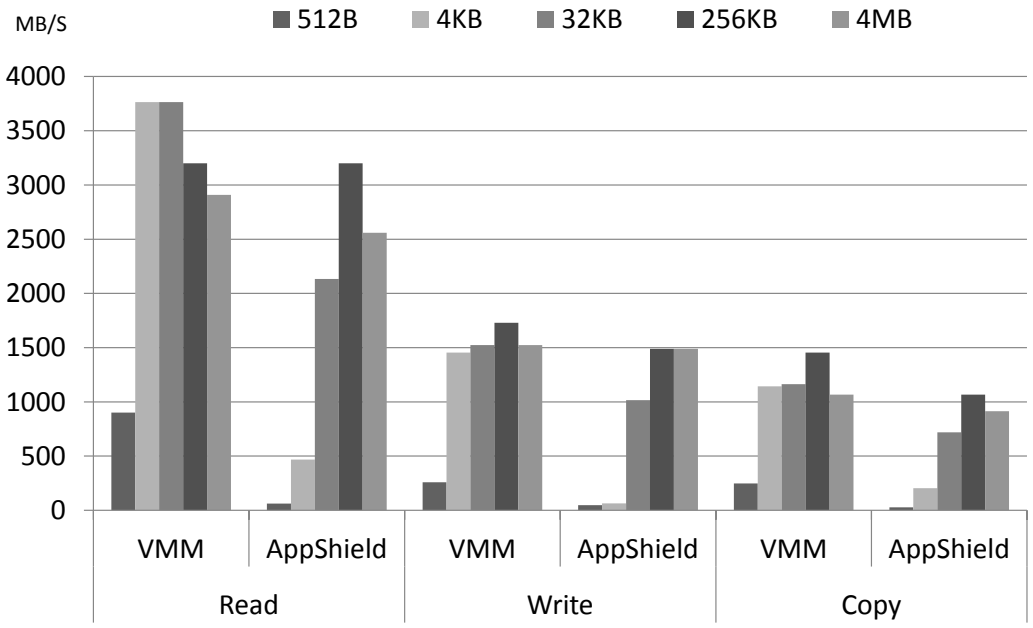


Figure 4.11: The disk I/O Benchmark.

Network I/O Benchmark

We measured the network performance with the Apache web server. The Apache is configured as worker mode with one main process and 20 threads. We run the standard *ab* matchmarking tool included in the Apache utility tools. We execute 10,000 web requests, at the concurrency level of 100 to fetch the default *index* page. The web client and the Apache server are in the same LAN. The Apache web server

		Linux	AppShield	Overhead
Apache Throughput		320.65 req/s	316.84 req/s	1.01x
Connection Time (ms)	Processing	160	163	1.01x
	Waiting	131	135	1.03x

Table 4.5: The benchmark results of Apache.

serves requests with 1.20% overhead in throughput, and about 3.05% overhead in waiting time and 1.86% overhead in processing time. The overhead is reasonable since Apache may cache the frequently requested pages, without issuing disk I/O for each request.

4.6 Discussions

I/O Data Protection

To keep the hypervisor and the transit module small and simple, we do not integrate the device drivers into them, while choose to reuse the legacy ones in the untrusted OS. It means that the untrusted OS have the chance to reveal or tamper with the input data of the CAP. For the file and network data, we can use cryptographic techniques to protect them, e.g., all file and network data are encrypted with the CAP private key. However, the data like keyboard input and mouse input are passed from the devices in plain text, meaning that the untrusted OS is able to get the content or even arbitrarily modify them. To protect such data, we can integrate the trusted path [19, 109] with our AppShield technique.

Fine-Grained Protection

Currently, we focus on the whole application protection, while the AppShield technique can be adapted to protect self-contained high-assurance components. To achieve this, we can split the application into low- and high-assurance partitions, and only protect the high-assurance components. In addition, the high-assurance components that are aware of the existence of the AppShield can explicitly communicate with the hypervisor to request more fine-grained protections or certain special services, e.g., the online transaction service.

Verifiable Protection

End users usually require a proof (verifiable protection) to indicate the state of the protection. There are several approaches proposed to provide a secure feedback channel. Bumpy [59], ZTIC [50], Lockdown [97] and Trusted Path [109] attempt to use a dedicated (extra) hardware device (e.g., USB token and mobile) as the trusted monitor, while KGuard [18] and Guardian [21] build a visual verification on the display. We can integrate the the visual verification into our hypervisor since it does not need dedicated devices. Note that the visual verification is only involved when the first time the CAP is isolated.

4.7 Summary

In this chapter, we have presented the designed and implementation of AppShield, which reliably and flexibly protects critical applications with complete isolation, rich functionalities and high efficiency. We have implemented the prototype of AppShield with a small bare-metal hypervisor. We have evaluated the performance impacts on CPU computation, disk I/O and network I/O using micro and macro benchmarks. The experiments show thatAppShield is lightweight and efficient.

Chapter 5

Generic Protection On I/O Flows

In this chapter, we aim to protect data flows between applications and devices against an untrusted kernel throughout the entire I/O lifecycle. In particular, we focus on those devices that render raw data, e.g., sound cards and printers, or generate raw data for applications, e.g., seismic sensors and fingerprint scanners. We are less concerned with disks and network adaptors, because these devices deal with derived data from applications. Therefore, a straightforward solution to protect the disk I/O and the network I/O is to encrypt the data before and after I/O operations.

In this work, we present DriverGuard, a holistic and compact I/O protection system making use of a combination of cryptographic and virtualization techniques. We implement DriverGuard with slight changes on the device drivers and the Xen hypervisor. Our experiments with several I/O devices demonstrate that DriverGuard imposes little overhead to the system and causes unnoticeable delays to user applications. DriverGuard is complementary to many user space protection schemes such as Overshadow [17], and SP^3 [106]. A composition of DriverGuard and a user-space protection scheme can protect the whole lifecycle of data processing.

Our work is also remarkably different from secure I/O [84] and driver code security [78]. Secure I/O copes with those attacks misusing the I/O mechanism (especially DMA operations) for illegal memory accesses. Driver code security tackles software attacks, such as return-address attacks [14] and code injection attacks [55],

which gain the root privilege by subverting drivers. Although these attacks do not necessarily target at the I/O data, they are one of the threats considered in our study. Our work is similar to the trusted path proposed by Zhou et al. [109]. Their trusted path aims to assure the *secrecy* and *authenticity* of data transfers through a trusted path from the *new inserted user-level* driver to the device, while our work focuses on the protection of the *secrecy* of the I/O data through a trusted path built upon the *legacy* drivers.

ORGANIZATION The next section describes the problem as well as the threat model, security requirements and main challenges in Section 5.1, and explain the design rationale in Section 5.2. We describe the design overview, privileged code block and design details of DriverGuard in Section 5.3, 5.4 and 5.5, respectively. Section 5.6 discusses the automatical PCB identification and the full path I/O protection, and Section 5.7 shows the evaluation of DriverGuard through experiments and performance measurements. Finally, we conclude this chapter in Section 5.8.

5.1 Problem Definition

In this section we state our goals together with the threat model, and present the possible attacks and the main challenges we are facing.

5.1.1 Our Goals

There are several approaches [17, 106] proposed to protect the data in the user space. Therefore, in this chapter we focus on the I/O flow protection in the kernel space. Specifically, we attempt to propose an approach to defend against attacks that get the value of the I/O data from device interfaces and kernel-space buffers. More specifically, we focus on the protection of the *raw I/O data* that is generated from or send to devices. Disk I/O and network I/O are not in the scope of our study, because neither disks nor network adaptors produce or render raw data. In fact, data stored in disks or transmitted across networks are actually generated by user applications. Therefore, they can be protected using user-level encryption techniques (e.g., using

SSL to protect the network data). Note that the devices mentioned in the chapter are hardware/physical devices rather than virtual devices.

The second goal is to propose a *generic solution* to protect all kinds of I/O flows on different machines and different devices. We attempt to make our system to be compatible with commodity operating systems and legacy applications.

5.1.2 Threat Model and Assumptions

In our threat model, we consider the malicious software residing in the guest as the adversary. The malware may compromise the kernel through attacks like ROP [80, 9, 14] and code injection [55]. As a result, the adversary can take full control of the guest, e.g., launching arbitrary code and issuing any DMA requests.

We assume that malware can not subvert the hypervisor. This assumption is reasonable since the TPM-based secure boot scheme can guarantee the load time integrity of the hypervisor, and the virtualization technology can prevent malicious software and illicit DMA accesses driven by the guest OSes in runtime. In addition, some proposed hypervisor-protection schemes [99, 98, 5, 69] can be applied to ensure the hypervisor's security.

We trust the end user, and assume that the adversary can not physically control the system. We assume that the hardware devices always behave according to their specifications. We also assume the system firmware is trusted as in [45, 91, 93, 97]. The modern BIOS has a built-in hardware lock mechanism to set itself as read-only and only accepts signed updates, so that the OS cannot tamper with it. Due to the complexity of the x86 platform (e.g., optional ROM), this assumption may not always be true. Nonetheless, it is still possible to validate the system firmware by the proposed attestation approach [52] or by a trusted system integrator. Furthermore, for the computers in an organization, the security-savvy system administrator can simplify the system boot settings, such as disabling unnecessary option ROMs.

In this work, we only focus on uniprocessor platforms. Attacks from multi-core or multi-processor are out of scope of our discussion. Note that it is not our

interest in this chapter to study how to check the trustworthiness of a device driver. We assume that a trusted authority[1] signs every device driver to be installed. The hypervisor can validate the integrity of device drivers by verifying the signatures. Neither the denial-of-service attacks nor the side channel attacks (e.g., [85]) are in the scope of our work.

Although the security and functionality of DriverGuard are independent of user space protection, the benefits are maximized if DriverGuard joins schemes such as Overshadow [17] and SP^3 [106] to safeguard the entire I/O data life cycle covering both kernel and user spaces. We will discuss this issue in Section 5.6.2.

5.1.3 Attacks

According to the characteristics of driver operations, we spell out attacks targeting at the I/O data. From the above typical I/O flow, we can find out many possible attack targets to get the value of the I/O data. We summarize all attack targets into three categories: device I/O interface, the kernel space data buffer, and the user-space data buffer.

- **Attacks on device I/O interfaces.** The device I/O interfaces include PIO, MMIO and DMA descriptors. For PIO and MMIO, a rootkit may launch the I/O-port or MMIO mapping attack [109] to intercept or manipulate the device I/O, or directly access the interface to get the I/O data. It may also attempt to modify the DMA descriptor to induce the device sending or fetching the I/O data to or from the memory regions controlled by the rootkit.

- **Attacks on kernel space data regions.** This type of regions include all driver allocated memory regions. A rootkit can keep probing and reading the target I/O data, or be triggered by some special event (e.g., external interrupt) to access the I/O data directly from these regions. It is hard for the kernel to defend against such attacks because the rootkit has the same privilege as the

[1]To avoid increasing the TCB size, we suppose that the platform administrator signs every driver to be installed in the platform. The hypervisor is pre-configured with the administrator's public key.

kernel. Another attack is that the rootkit calls the driver's legitimate routine to access the data.

- **Attacks on user space data regions.** The user space regions are wildly open for a kernel rootkit. Once the I/O data is in the user space, the rootkit is able to bypass kernel- and user-level protections to directly access it. Such attacks can be defended by serval user-space protection approaches [17, 106].

5.1.4 Security Requirements

Given that the locations of the I/O data can be categorized into two types: device interface (including PIO, MMIO and DMA) and main memory, we summarize all required security properties of the I/O flow protection on each of them.

For the data in the device interface, we require that malicious code can not access or manipulate the data. The security properties are stated as:

- $SP0$: The physical addresses and I/O ports of all device interfaces can *not* be updated once they are fixed by the BIOS during the system boots up.
- $SP1$: Any access on the data or to update data transfer parameters through the device interface should be intercepted and verified.
- $SP2$: Only accesses from trusted code blocks are granted.

For the data in the main memory, we require that only the trusted code blocks are able to *read* the protected I/O data, and the executions of the trusted code blocks are protected. The security properties are summarized as follows:

- $SP3$: If the I/O data in a memory buffer is readable (plain text), access control must be enforced and only granted the trusted code blocks can access it.
- $SP4$: If the I/O data in a memory buffer is unreadable (cipher text), any code blocks are able to access it without triggering any verification mechanism.

- $SP5$: If a trusted code block is interrupted to give up CPU during its execution, its execution context must be saved and restored when it occupies CPU again.

5.1.5 Challenges

We now discuss the challenges in designing a system that provides the guarantee of confidentiality of the I/O data over the lifetime of the system. The first challenge is the complexity of the I/O sub-system. Different devices have different interfaces to communicate with the system. For instance, cameras use USB interface while the PS/2 keyboard is attached to the system with the PS/2 interface. Furthermore, for the same device with the same version, different platforms (e.g., Linux or Microsoft Windows) have different driver implementations. The diversity and complexity dramatically increase the difficulties to build a generic solution to protect all I/O flows.

The second challenge is the complex and intensive interactions between drivers and the kernel. Most driver functions are dependent on the kernel exported functionalities. For instance, the driver memory allocation and deallocation are heavily dependent on the kernel memory management component. The heavy dependence and intensive interactions make it extremely hard to distinguish if an access on the protected I/O data is driven by the benign driver or by the compromised kernel.

The third challenge comes from the power of attackers. Once attackers compromise the kernel, they are able to gain the kernel (highest) privilege, which allows attackers' code to freely access any memory regions and I/O ports. On the other hand, it is hard for the buggy monolithic kernel to completely defend against software attacks due to the large size and numerous attack surfaces.

5.2 Design Rationale

A straightforward approach is that the hypervisor arbitrates whether a control flow can access the I/O data. It requires the hypervisor to introspect driver operations, which is difficult to implement due to the semantic gap (e.g., lack of details of driver

operations) between the hypervisor and the driver. Considering the complexity of I/O operations, the workload on the hypervisor will inevitably expand its code size, and may significantly downgrade the whole system performance.

Isolation is a widely used method to protect program executions. To apply isolation on I/O data protection, one may propose location isolation or execution isolation. Location isolation is to place device drivers and the kernel's I/O subsystem into a separated domain, e.g., a driver domain or Dom0 in Xen, or the hypervisor's space, e.g., VMware, so that malware in the guest kernel can not attack them directly. These approaches are efficient in terms of I/O performance. Nonetheless, the resulting protection is weak because the TCB size is increased significantly due to the drivers and the I/O subsystem.

In the execution isolation, the device drivers still reside in the untrusted guest kernel while their executions are escorted in a secure environment established by the hypervisor, similar to TrustVisor [56] and Overshadow [17]. The generic execution isolation is not applicable for I/O data protection, because I/O operations are featured with frequent hardware interrupts and intensive driver-kernel interactions. Note that if the I/O subsystem is also enclosed in the execution isolation, it suffers from the same drawback as in the location isolation approach.

We adopt the idea of execution isolation, however, at a micro-level. It is well-known that most of the driver code is for housekeeping purposes, such as error handling, resource allocation and cleaning up [33], with only a small portion dealing with I/O data transferring. We further observe that among the code for data transferring, only a few code blocks, e.g., an encoding function, need to process the I/O data, while the majority of them just move the data from one memory location to another without necessarily knowing the content. Based on these observations, we design DriverGuard as a fine-grained I/O protection mechanism, which enforces access control on the device interfaces and encrypts the I/O data once it is moved into memory. To let the device driver work properly, DriverGuard distinguishes those security-sensitive driver code (around 1% of the driver code according to our

experiments) from the rest. Only these identified code blocks are granted to access device interface, and access decrypted I/O data. Other code blocks is only able to access encrypted I/O data. In the meantime, DriverGuard protects the execution of security-sensitive code block to prevent malicious code from accessing the I/O data. Different from the hypervisor introspection technology, those access controls do not impose comprehensive semantic logics on DriverGuard. Hence, its performance is on par with the location isolation solution, however, the security strength is much stronger.

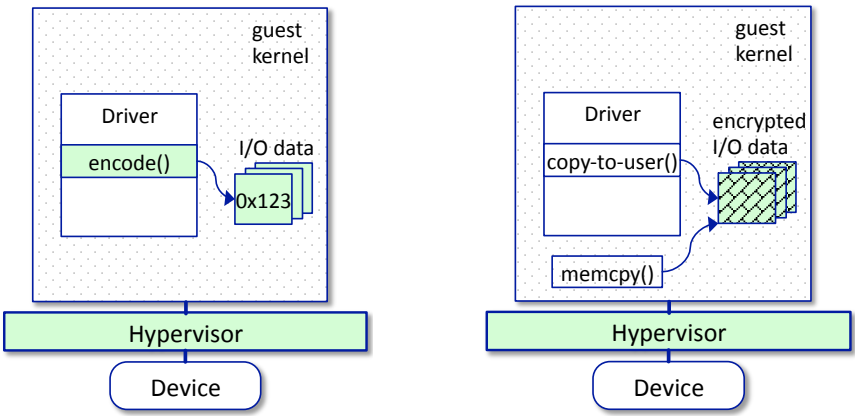
5.3 Design Overview

By and large, DriverGuard is constructed using three lightweight protection techniques as the building blocks: cryptography, access control and runtime protection. We use cryptographic techniques to protect all I/O data without interfering with most of the driver and the kernel executions. For regions holding data which cannot be protected by encryption, we resort to DriverGuard to enforce access control. The plaintext data can only be accessed by a few designated driver code blocks, which are trusted and whose executions are safeguarded by our runtime protection mechanism. We refer to these code blocks as *privileged code blocks* (PCBs) in the rest of the chapter. By protecting the execution of PCBs, we successfully ensure the whole I/O data security with minimal overhead since PCBs only constitute a tiny fraction of the driver code.

5.3.1 Protection Mechanism

A high level view of DriverGuard's protection mechanism is as follows. Once the hypervisor boots up, it fixes all physical addresses and I/O ports of all device interfaces by setting read-only on the configuration space registers, and rejects any update requests from the guest OS [109] (achieve $SP0$). All device interfaces related to protected I/O flow are enforced access control by the hypervisor, so that any access from the guest must be trapped into the hypervisor (achieve $SP1$). If

the access is from a PCB (i.e., command-PCB or computation-PCB), the hypervisor grants the access, otherwise rejects it (achieve $SP2$). Receiving the I/O data from device interface, a PCB may attempt to read the content of data to do some computations, such as encoding or decoding operations. Before the PCB is off the CPU, the PCB is designed to either require the hypervisor set access control back on the data if it is readable[2] (achieve $SP3$) or encrypt the data into cipher text with the key (achieve $SP4$, and see more in Section 5.3.4), which is generated by the key-PCB and only accessible by PCBs. Non-PCBs are free to move the ciphertext to anywhere without any constrains from the hypervisor. Figure 5.1(a) and Figure 5.1(b) illustrate the difference between a PCB's and a non-PCB's I/O data accesses. Since the PCBs never actively give up the CPU until its execution flow ends, the off-CPU event must be triggered by external interrupts or exceptions. Based on this observation, the hypervisor enables interception mechanisms on all interrupts and exceptions during the PCB execution. If the interception mechanism is triggered, the hypervisor restores protection on the I/O data. Furthermore, the hypervisor also protects the PCB execution context to avoid indirect data leakage (achieve $SP5$).



(a) A PCB, e.g., *encode* function, accesses the I/O data in plaintext.

(b) A non-PCB, e.g., the *memcpy* function, accesses the encrypted I/O data.

Figure 5.1: The concept of privileged code block (PCB).

In Section 5.1.2, we assume that every driver is signed by a trusted entity such as the platform's administrator. To ensure the initial integrity of the PCBs in a

[2]The PCB is able to get enough semantic information to know if the I/O data is readable or not.

driver, we further assume that they have been explicitly labeled in the driver code before being signed and installed. Therefore, the signature on the driver code also ensures the integrity of PCBs. (We will discuss PCB identification methods in Section 5.4.1.) Next, we explain the design details of three building blocks and leave the discussion of their integration in Section 5.5, since it involves the details of I/O operations.

5.3.2 Access Control Over Critical Regions

Since we do not rely on encryption-capable devices, encryption is not applicable for data used by the hardware. To cordon off illicit accesses to the data, we utilize the hypervisor's access control mechanism. In general, the data regions are classified into memory regions and I/O ports, for which we apply different access control methods by leveraging the hardware features and the virtualization techniques available in the platform.

To intercept accesses to a protected memory region, DriverGuard sets the attribute bits in the corresponding Page Table Entries (PTEs), clears the corresponding IOPL bits, and sets up the I/O bitmap to intercept accesses to an I/O port. Note that the protected memory addresses are machine addresses not guest physical addresses (or named pseudo physical addresses). We use *checkpoints*[3] in the rest of the chapter to refer to both the IOPL bits and the PTEs marked by the hypervisor for the purpose of access interception. Although the aforementioned protection techniques are used in many existing schemes, e.g., [17, 67], we are confronted with two new problems. First, given a memory buffer, the hypervisor must make sure that the kernel can not bypass the checkpoint to access the region, which is challenging for memory regions allocated by the kernel. Secondly, the hypervisor must ensure that the sensitive I/O data is indeed placed in the region *with* a checkpoint. The first problem demands a careful page table walk checking while the second demands the I/O control integrity checking.

[3]Our definition of *checkpoint* has no relation with the *checkpoint* for rollback in distributed systems.

5.3.3 Cryptographic Components

We introduce to the device driver a symmetric-key encryption function and a decryption function, both of which can be called by any code. However, any write access to the function code is denied by the hypervisor. We also add a key generation function to the driver as a PCB. The security of the I/O data relies on the secrecy of the driver's key, rather than the secrecy of the decryption function, which complies with the famous Kerckhoff's principle. The driver's secret key is securely generated based on a secret random seed supplied by the hypervisor. The secret key is securely stored in a kernel space buffer priorly appointed by the driver and can only be accessed by the driver's PCBs. This prevents any unauthorized code from decrypting the driver's data, even though the decryption function can be called arbitrarily.

5.3.4 PCB Execution Escorting

The third building block in DriverGuard is the runtime protection mechanism that prevents a PCB's execution from deviating its expected behaviors. The protection is requested at the PCB's entry and is relinquished at the exit via hypercalls. The hypervisor agrees to admit a control flow into the escorting only when the request is issued from the driver's PCB, and agrees to discharge a flow from escorting only when the request is issued from the PCB presently under escorting. The PCB is registered to DriverGuard during the kernel boot up process (details in Section 5.5).

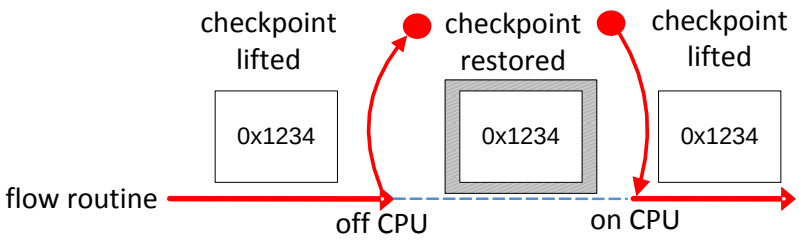


Figure 5.2: An illustration of runtime protection, where $0x1234$ is an exemplary memory address with a PTE checkpoint.

The PCB under the escorting is granted by the hypervisor to access the critical

data such as the driver's secret key and the I/O data, or to issue I/O commands. In our design, the hypervisor temporarily restores the access on those regions for the PCB, and withdraws the access right at the exit of escorting. Therefore, no duplicated exceptions or page faults will be raised despite that the PCB may access the same region multiple times within one escorted execution. An escorted PCB can be scheduled off from the CPU for various reasons. In that case, the hypervisor intercepts these events and restores all checkpoints. Meanwhile, it also securely saves the driver's runtime stack and sets up a breakpoint for the PCB's upcoming CPU occupation. As a result, other code's accesses to the protected regions are denied. Figure 5.2 depicts a scenario of escorting.

5.4 Privileged Code Block

We consider three types of PCBs in a driver. One is *computation-PCBs* which refers to the driver code blocks making computation on the I/O data, e.g., an encoding function. The second is *command-PCBs* which refers to the driver code blocks issuing data transfer parameters to the device. This type of code is security sensitive because their executions determine the locations of plaintext I/O data. The third is *key-PCBs* which refers to the driver code blocks initializing the driver's encryption key. Each driver generates its own key in the driver initialization step, such as in `module_init`.

There are several properties of PCB summarized as follows: 1) It is self-contained in the sense that there is no extra function calls to kernel functions; 2) It does not contain indirect call or indirect jump (e.g., no call using function pointer), meaning that the control flow is static; and 3) It does not have any dynamic data dependence except for the parameters. These properties call for driver developers' prudence in driver coding such that the driver code is friendly to PCB identification.

5.4.1 Identifying PCB

Given that the *key-PCB* is added by the DriverGuard scheme, we only illustrate how to identify the other two types of PCB from the driver code. Following the I/O data flow, the sophisticated driver developers are able to identify all functions that operate on the I/O data, and thereby label all PCB candidates in these functions. If some PCB candidates are not naturally satisfy the above listed PCB properties, there are some guidelines for the driver developers to modify them into PCBs.

Obviously, it is easy to achieve the second property by carefully programming. To achieve the first property, the developer can replace external function calls with its own code if they are simple (like *inline* functions). If they are hard to be replaced, the developer may either move these function calls out of the PCB candidate if they do not effect the behavior of the driver, or divide the PCB candidate into two PCBs with the function call as the separator. In order to achieve the third property, the developer could assign the dynamic dependence data to the static or global data variables. To facilitate the protection on these variables, developers are able to put them in a particular region (e.g., a pre-reserved page) with compiling flags. Note that the labeled driver can be distributed after the PCB-labeling work is done.

5.4.2 PCB Format

A PCB is always capsulated by a pair of hypercalls for escorting. The entry hypercall is a *start_escort* hypercall that requires DriverGuard to start to protect the execution of the PCB, and the exit hypercall is an *end_escort* hypercall that informs DriverGuard to end the PCB execution protection. There are two possible formats for a PCB. One format of the PCB is ended with encryption on the protection data. Such PCBs are usually in the intermediate parts of a driver, where the driver does some process operations on the I/O data whose output is ready for the later stage. The other is ended with protection requirement which is to request DriverGuard to block all accesses on the protected regions. Such PCBs usually directly work

with device (e.g., updating I/O buffer for DMA transferring) or user-level applications (e.g., copying data into user space). We illustrate their different formats in Figure 5.3.

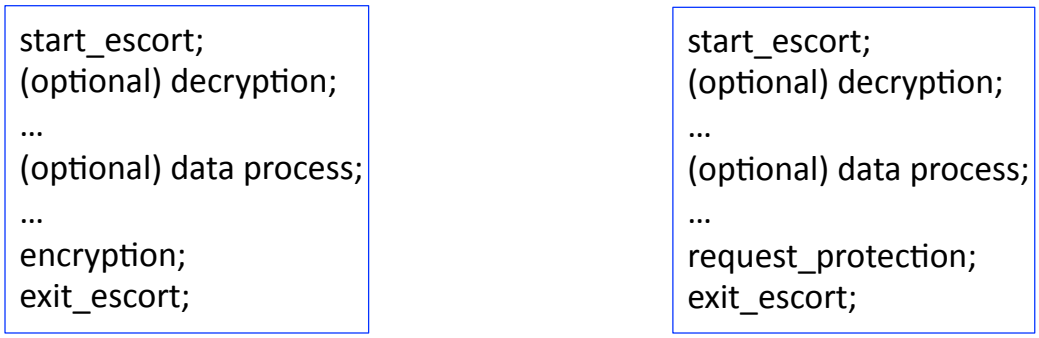


Figure 5.3: The two types of PCB format. (a) A PCB ending with encryption. Thus, the hypervisor does not need to enforce access control on the data; (b) A PCB ending with protection requirement. Therefore, the hypervisor must enforce access control on the data to restrict the access.

We assume that the PCB interface is trusted and well-designed/implemented, without malicious intention to leak I/O data to outside. In fact, it is true in almost all cases especially for the ones in the driver interfaces since they are usually well defined in the specification.

5.5 Design Details

We build DriverGuard on top of the Xen hypervisor to protect the drivers running in a Linux guest domain. We systematically examine every step in I/O operations, from the device discovery to the application's (or device's) data fetching. In order to adaptively protect the driver operations, the hypervisor needs to store certain context information about the driver. We start with driver context initialization since it is performed by the hypervisor during the guest domain bootstrapping.

5.5.1 Driver Context Initialization

The context information of drivers are securely stored in three types of tables in the hypervisor space. A *device table* specifies the management relation between a driver

and a device by paring their identifiers. For every protected driver, the hypervisor maintains a *PCB table* and a *region table*. The former stores the entry and exit addresses of all PCBs of the driver while the latter specifies the memory regions and the I/O ports to protect. There are five types of regions in the region table: 1) the application buffer; 2) the memory buffer allocated by the driver for data processes; 3) the I/O data buffer such as DMA buffers ; 4) the device interface, including the I/O ports or MMIO regions and DMA descriptor queues; and 5) the buffer holding the driver's secret key. Figure 5.4 depicts their locations in the system.

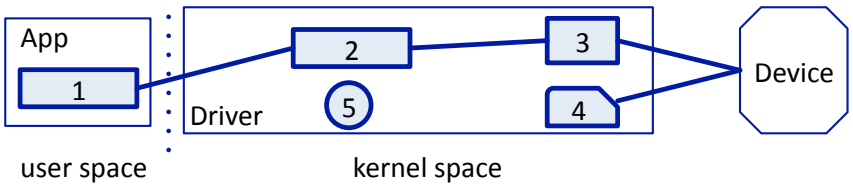


Figure 5.4: An illustration of five types of regions with same numbering in the description.

Device Table Initialization When a guest kernel image is uncompressed, the hypervisor inserts a hook function to the kernel to inform the hypervisor about the device-driver association via a hypercall. The hypervisor then initializes the device table accordingly. The hypervisor also sets the checkpoints for the kernel structure maintaining the device-driver association. Whenever a driver takes the ownership of a device, the hypervisor intercepts the event and updates the device table properly.

PCB Table Initialization We assume that all PCBs in a driver have been manually identified and delimited by a pair of hypercalls, i.e., an escorting-entry hypercall and an escorting-relinquish hypercall. The hypervisor scans the driver code to record the addresses of escorting-entry hypercalls and of the respective escorting-relinquish hypercalls. It puts these pairs into the PCB table. In Section 5.6.1, we will discuss how to automatically identify PCBs.

Region Table Initialization The regions used by a driver can either be the default ones chosen by the manufacturer/the kernel or set by the driver. In the first case, the hypervisor updates them when the driver is loaded as in the device

discovery step. In the latter case, the driver informs the hypervisor via a hypercall about the protected regions or I/O ports.

Driver Key Initialization Each driver has a dedicated key-PCB to initialize its own encryption key. In the key-PCB, key initialization algorithm first issues a hypercall to get a random seed from hypervisor, and then saves the generated key into its key buffer. The key generation process is only run once, and escorted by hypervisor (see escorting details in section 5.5.3).

5.5.2 Checkpoint Deployment

Given a memory region or an I/O port, the hypervisor sets up the corresponding checkpoint to intercept and verify potentially malicious accesses. The detailed deployment method is dependent on the virtualization environment.

Memory Region Checkpoint

For a memory page A, the hypervisor walks through the page tables through the $CR3$ register to locate the corresponding PTE pointing to it. The hypervisor sets the attribute bits on the PTE to specify different access rights. To set a page *read-only*, the _PAGE_RW bit is cleared; and to set a page *non-access*, the _PAGE_PRESENT bit is cleared. Note that all protected regions are in kernel space and all processes share one kernel space mapping. In the paravirtualization setting, only the hypervisor is able to update page tables. In the hardware-assisted virtualization setting, a Shadow Page Table (SPT) or Extended/Nested Page Table (EPT/NPT) is used to translate virtual addresses into machine addresses, and the SPT/EPT/NPT is only updated by the hypervisor. Although the mechanism of the SPT is a little bit different from the one of EPT/NPT, i.e., the SPT requires the hypervisor to control over the guest page table while the EPT/NPT do not need, the page-access-checking mechanisms are essentially the same, since the final access right to a page is determinate by the page table, i.e., SPT/EPT/NPT, handled by the hypervisor. Therefore, those checkpoints can not be removed by the malicious kernel. Note that in the hardware-assisted virtualization environment, the hypervisor enforces access on the

machine address, not the pseudo physical address or virtual address.

Given that the granularity of the memory protection is in the page-level, we should carefully deal with the I/O data buffers that are not page-aligned or mixed with other data in a single page. There are two options to solve the problem. One is to request I/O buffer at the length of pages. In fact, to accelerate the performance, many device drivers have such allocation feature. For example, the USB camera driver allocates a large memory pool in page level for data caching. The other option is to let the hypervisor emulate the operations that access other data. In DriverGuard, we choose the second option to avoid changes on driver code. Although emulation incurs performance loss, the likelihood of its occurrence is low. This is because the checkpoints are only deployed on device interfaces used immediately after or before I/O, and other data are protected by encryption.

Legitimacy of Memory Region When the hypervisor attempts to set up a checkpoint, it checks whether the machine memory page can be reached by another unauthorized PTE. In other words, the kernel is not allowed to bypass the checkpoint to visit a machine memory page. Therefore, the hypervisor must ensure that there exists no unchecked virtual-to-machine address translation for memory pages with checkpoints.

We leverage the hypervisor's memory management mechanism to tackle this issue. The Xen hypervisor maintains a `page_info` structure for every machine memory page. The `count_info` field in this structure records the number of usages of a machine memory pages. For a page allocated to a guest, its `count_info` is actually 2, because the hypervisor itself is holding it[4]. Therefore, on setting up a PTE checkpoint, the hypervisor checks if the corresponding counter is 2. In addition, we modify the hypervisor's `do_mmu_update` and `do_update_va_mapping` to prevent the kernel from crafting a trapdoor path for existing checkpoints. These functions are used by the guest kernel to update page

[4]Before a machine memory page is allocated to a guest OS, the hypervisor holds it first and sets the `PGC_allocated` bit. Therefore, the page's `count_info` is already 1 before being allocated to the guest.

tables. In this way, for any page table update, the hypervisor checks whether the requested update increases the usage counter of any machine memory page with a checkpoint.

In the hardware-assisted virtualization environment, the hypervisor does not set two PTEs pointing to the same machine address. To enforce this property, the SPT/EPT/NPT update algorithm can be extended to verify it. Note that the legacy hypervisors, such as Xen, do not export any interface for the guest to manage the SPT/EPT/NPT.

I/O Port Checkpoint

I/O ports is separated from the memory address space, and the accesses to such I/O ports need a set of special instructions, e.g., inb and $outb$. A successful access must go through the IOPL checking and I/O bitmap checking. Any access will be blocked once its priority is lower than the priority specified in the IOPL. Even if the access pass the IOPL checking, it is still blocked if the corresponding bit is set in the I/O bitmap. To prevent the malicious guest kernel from accessing the protected I/O ports, the hypervisor clears the IOPL bits of *EFLAGS* of the guest's CPU. Namely, it sets the I/O privilege level to 0, such that the hardware always checks the I/O bitmap for PIO instructions because the paravirtualized kernel runs in Ring 1. Then, the hypervisor sets the bits corresponding to the protected I/O ports such that a PIO instruction will cause a general protection exception.

It is relatively easier to set up I/O checkpoint in hardware-assisted virtualization. More specifically, the hardware-assisted virtualization technique supports that the hypervisor itself can intercept all instructions that access a particular I/O port through a dedicated I/O bitmap in the hypervisor space. Therefore, the hypervisor simply activates the I/O bitmap mechanism by setting the 25th bit of the processor-based VM-Execution control vector and then sets the bits in the bitmap corresponding to all protected I/O ports.

5.5.3 PCB Execution Escorting

PCB Admission

A driver's PCB starts with the hypercall which takes as the parameter the buffer address it requests to access. To admit a PCB, the hypervisor checks whether the hypercall is issued from the instruction whose address is registered in the PCB table. If not, the hypervisor rejects the request.

For an admitted PCB, the hypervisor protects its stack as follows. The hypervisor allocates a dummy stack for the PCB. Therefore, an admitted PCB has two runtime stacks. A genuine stack is used for the PCB's execution while the dummy stack is used for untrusted code sharing the same execution flow due to interrupts. The usage of dummy stacks will be explained in the next subsection. Figure 5.5 below describes the details of the PCB admission algorithm, where InEscorting is a flag bit indicating the current execution state.

Admission Algorithm:

1) Fetch the EIP value stored at the top of the current guest kernel stack, which is the return address of the hypercall.
2) If EIP does not match any entry in the PCB table, return error.
3) If the address of requested buffer is legitimate, then
    a) set InEscorting to 1;
    b) If the guest's kernel stack segment is not a dummy stack, then
        (i) allocate a dummy stack at the reserved space.
        (ii) save the machine addresses of the dummy stack and the present stack as (MA'_{ss}, MA_{ss}). Return 0.
    c) else, switch to the corresponding genuine stack. Return 0.
6) Return -1 as an error message for admission failure.

Figure 5.5: Algorithm for PCB admission.

Escorting

Once a PCB is admitted by the hypervisor, its execution is escorted and the checkpoints for the buffers are temporarily lifted. The essence of escorting is that the hypervisor intercedes whenever the PCB is scheduled off from the CPU, which occurs due to the hardware interrupts. This situation opens the door to the kernel attacks,

because the kernel may occupy the CPU and could access the PCB's runtime stack and data. To defend against such attacks, the hypervisor should be able to enforce access control on the data and stack before the potentially malicious kernel occupy the CPU. In the virtualization environment, the hypervisor is able to configure the system to give priority to itself to occupy the CPU. Specifically, all hardware interrupts are sent to the hypervisor prior to sending to the guest domain. Therefore, the hypervisor is able to 1) restore the checkpoints and 2) replace the runtime stack with the dummy stack allocated in PCB admission. The hypervisor also sets a breakpoint to intercept the events that the PCB is re-scheduled to the CPU.

We explain below how the hypervisor handles an interrupt through interrupt handler `do_IRQ` and a debug exception through the debug exception handler `do_debug` in addition to its normal process.

Interrupt To switch to a dummy stack, the hypervisor only replaces the content of the PTE for the present stack with the machine page number of the dummy stack allocated during admission. This change is transparent to any guest process, since the address in the ESP register remains the same. Hence, the guest kernel is not able to access the true stack while the subsequent execution can use the dummy stack without being affected. The algorithm for stack switching and checkpoint restore is shown in Figure 5.6.

IRQ-handler Algorithm:

(1) If $InEscorting = 0$, return.
(2) Restore the checkpoints that are removed during escorting.
(3) Switch to the dummy stack, by setting the PTE for the guest's stack base to point to MA'_{ss}.
(4) Set $InEscorting = 0$.
(5) Set a local breakpoint at the instruction pointed by EIP. Save the address pair in EIP and ESP.
(6) Return and pass the control to the default interrupt handler.

Figure 5.6: Interrupt handler for escorting. When there is an interrupt interrupting the execution of the execution of a PCB, the interrupt handler restores the protection on the escorted data, and saves the context of current escorting PCB.

Debug Exception When a debug exception occurs, the hypervisor's *do_debug* func-

tion is called before the event is forwarded to the guest kernel. All breakpoints used by the hypervisor are local breakpoints. Therefore, they are triggered only for the present process. There are two types of local breakpoints used in DriverGuard.

Setting a breakpoint at the EIP is to intercept the event of PCB resuming. For this type of breakpoint, the hypervisor enters into escorting only when both EIP and ESP values match the previously saved pair. The details are shown in the following algorithm in Figure 5.7.

Debug-handler Algorithm: Breakpoint address stored in *EIP*, the stack address stored in *ESP*

/* Enter into Escorting */
(1) If there exists a saved (EIP', ESP') pair, s.t. $ESP' = ESP$ and $EIP' = EIP$, then
  (a) remove the breakpoint at *EIP*;
  (b) Restore to the genuine stack by replacing the stack PTE with MA_{ss}.
  (c) Set $InEscorting = 1$, and return 0.
(2) Return -1 as an error message.

Figure 5.7: Exception handler for escorting. When a previous interrupted PCB resumes, the exception handler restores the PCB execution context.

PCB Exit

To exit from the hypervisor escorting, the PCB issues another hypercall. The hypervisor checks if InEscorting is set. If not, it returns an error message; otherwise, it clears InEscorting flag. The PCB should also issue a hypercall to protect its data if the data are left in plaintext. The hypervisor sets no more breakpoints and processes interrupts and exceptions in the normal way.

5.5.4 Data Region Access Control

A potentially malicious access to a memory region with a checkpoint causes a page fault and an access to an I/O port with a checkpoint throws out a general protection exception. Therefore, we modify the hypervisor's page fault routine `do_page_fault` and the general protection exception handler `do_general_protection`. In the former, the hypervisor gets the address of the trapped instruction from EIP and the address being checked from CR2, while in

the latter, the I/O port number is enclosed in the instruction.

If the access is granted by the hypervisor, the event will not be forwarded to the guest kernel. In that case, The legitimate flow continues to execute the intercepted instruction without being re-scheduled due to the page fault as the guest kernel does not observe this exception. For unauthorized accesses, the page fault or exception is passed to the guest kernel. DriverGuard is compatible with memory mapping for page sharing because the checkpoints are deployed at the PTEs. A buffer mapped to two addresses has two PTE checkpoints. In the following, we elaborate the details of region access control according to all types of regions except the control region.

I/O Buffer The addresses of I/O buffers are obtained within an escorted command-PCB. Since the I/O buffer contains the data to/from the device, they are not protected by encryption. The hypervisor blocks all accesses not from an escorted PCB. For an input buffer containing the data from the device, the driver always encrypts the data before moving them to other locations, whereas for an output buffer the driver must decrypt the data after copying them to the output buffer.

Driver Buffer Driver buffers temporarily hold data for processing. When the data in those buffers are encrypted, the hypervisor does not set up checkpoints for them. Only when the escorted PCB is temporarily scheduled off from the CPU, the hypervisor sets up the checkpoints against all accesses as the data are in plaintext. In this case, the PCB notifies the hypervisor about the buffer address.

Key Buffer The key buffer holds the secret encryption key used by the driver. The hypervisor allows the key to be read only from the instructions from the encryption/decryption functions and is currently in escorting mode. Thus, a non-PCB can not access the encryption key.

5.5.5 Device Control Protection

As explained in Section 5.5.1, the control region's addresses can be obtained in the initialization phase for certain devices. The hypervisor denies all write accesses to the region not from an escorted PCB. Furthermore, it is also crucial to maintain the

consistency between the I/O buffer address specified in an I/O command which is sent to the control region and the buffer addresses requested by the device driver. This is because the kernel may manipulate the I/O command such that the device uses an unprotected I/O buffer for transferring. To defend against such attacks, the driver's command-PCB informs the hypervisor the locations of the I/O buffers in use, such as the DMA buffer and the DMA descriptor queue. The hypervisor inserts them in the region table and sets up the checkpoints accordingly. Therefore, it ensures that the I/O buffer in use is always protected.

5.5.6 Device Configuration Space Restriction

The physical addresses and the I/O ports of all devices are decided by the device configuration registers. All these configuration registers are located in the north-bridge chipset [31]. There are two possible methods to access them. One is through I/O ports. The I/O port $CONFIG\_ADDRESS$ (i.e., 0x$CF8$) is used to select a dedicated device whose configuration space is updated through the I/O port $CONFIG\_DATA$ (i.e., 0xCFC). The other method is through MMIO. Typically there is a continuous $256MB$ memory region reserved by the system for all devices. Any access to this region will trigger the chipset to propagate the configuration throughout the whole system. In order to avoid the configuration space conflicts (e.g., MMIO mapping attack) between different devices, the privileged code (e.g., the hypervisor) is able to verify the update requests by setting access control on the above I/O ports and the reserved memory mapped region.

In order to defend against I/O-port and MMIO mapping attacks, DriverGuard restricts the updates on the physical addresses and I/O ports of device interfaces. According to the above descriptions, DriverGuard sets checkpoints on the I/O ports 0x$CF8$ and 0xCFC, or the reserved memory region. The details of the checkpoint refers to Section 5.5.2. Any update (write) operations will be rejected. This restriction does not lower the runtime performance of the system since the configuration operation is normally done once at the bootup phase of the system.

5.5.7 User-Space Device Driver Support

When a device is managed by a user-space driver, the I/O data is directly transferred between a user space buffer and the device interface without any intermediary kernel space buffers. According to our threat model in Section 5.1.2, the user space memory regions are well protected by schemes like Overshadow. However, those schemes are not sufficient for I/O protection because they do *not* protect the device interface. Thus, we need to instrument the driver code with hypercalls to fix the protection gap. The inserted hypercalls update the information of the device interface to DriverGuard and request it to enforce access control on the device interface. Recall that all the user space driver code is protected. Thus, we do not need to identify PCBs for user-space drivers.

5.6 Discussions

In this section, we discuss the automatic PCB identification process, and the whole life cycle of protection on the I/O flows with the cooperations of DriverGuard and other user space approaches.

5.6.1 Automatically Identifying PCB

Ideally, a fully automated PCB identification algorithm can discover PCBs in a device driver with no false positives and no misses. False positives lead to unneeded overhead while misses result in loopholes for the adversary to attack. However, it remains as an open problem how to design such a PCB identification algorithm for a driver's source code or binary code. In this chapter, we make analysis of the challenges and propose a best-effort solution.

Recall that we defined three types of PCBs in Section 5.4. We only need to discover computation-PCBs and command-PCBs, as the key-PCBs are new function inserted to the driver. Labeling command-PCBs is straightforward, since they are featured with special instructions (e.g., inb and $outb$) involving the device interface

(e.g., I/O ports) or special memory region (e.g., MMIO). Many existing techniques can be used to identify the related statements, e.g., interprocedural points-to analysis technique [38] and the slicing techniques [86, 101, 62]. For instance, in the slicing techniques, we select the device interface (e.g., the I/O ports) as the seed, and the slicing tool can find out all statements that *directly* access the seed.

The computation-PCBs are the code blocks computing on the I/O data (e.g., mapping the scan code into key code in the keyboard driver). Identifying the computation-PCBs is a challenging task since it involves code and data semantics. To the best of our knowledge, existing code analysis techniques (e.g., forward and backward slicing, and thin slicing) are not sufficiently intelligent to distinguish code semantics. Another challenge is the abundant usage of function pointers in drivers, which makes it infeasible to determine execution flows through a static code analysis. This issue is aggravated by the fact that most drivers are essentially a collection of disjointed functions, instead of a single executable. The executions of driver functions are usually integrated with kernel execution. It is therefore difficult to map out all possible execution flow given existing code analysis techniques.

We propose a semi-automatic method for computation-PCB identification, with automated tools for coarse-grained scope defining on a large scale of code and human efforts for fine-grained refinement on small scale code fragments. Given a driver's source code which is a set of functions, the basic idea is to firstly pick up functions related to I/O data, then identify PCBs within each chosen function. In a nutshell, the procedure is divided into three steps: 1) to select functions which potentially contain PCBs; 2) to identify PCB statements in each function selected in previous step; and 3) to form all PCB blocks from the statements discovered in Step 2.

Step 1: Function Candidate Selection Drivers are highly structured code to conform to hardware interface specifications, such as providing file operation interfaces like `open`, `read` and `close`. According to hardware specifications, I/O flows

must originate at those designated interfaces. Therefore, those interface functions which do not process I/O data, as well as functions solely called by them, are excluded from our search scope. For instance, the interface function $poll$ corresponding to the $select$ system call usually does not access the I/O data, while allows a program to monitor and wait until one or more of driver/device states become ready for some class of I/O operations. The interface functions like $read$ and $write$ usually handle I/O data for the requests of the applications. For ease of presentation, we use $\mathcal{F}$ to denote the set of interface functions with I/O flows.

We then map out the execution flows (and therefore I/O data flows) starting from functions in $\mathcal{F}$, so that all dependent functions are examined. For this purpose, we first manually identify the I/O data used in $\mathcal{F}$, because the exact locations of I/O data in those functions are implementation specific, e.g., in the function parameters or predetermined buffers. Then, we use the I/O data as the seed to perform dynamic taint analysis [64, 48] to identify functions involved in I/O flow[5]. Since the dynamic taint analysis does not guarantee covering all execution paths, manual efforts are needed to check missed functions. Lastly, we extend $\mathcal{F}$ to enclose all functions picked up either manually or by the tool. For each function in $\mathcal{F}$, we identify computation-PCB in the next step.

Step 2: PCB Statement Identification In each selected function, we attempt to identify from the function body the PCB-candidate statements where I/O data are involved. Using the I/O data in Step 1 as the seed, we apply the slicing tools [86, 62] to label all seed-related statements in the function body. Note that the resulting statement set contains non-PCB statements for two reasons. Firstly, according to [74], slicing techniques may introduce false positive in statement discovering. Secondly, it is likely that some statements correctly identified by the slicing tools are not for I/O data computation, since the slicing technique does not take code semantics into consideration. For instance, statements that copy I/O data between memory buffers

[5] Although the method for dynamic taint analysis is applicable to drivers in principle, we have not found any existing tool suitable for this task.

do not satisfy the definition of computation-PCB. We suggest to manually examine the slicing results to filter out non-PCB statements.

Step 3: PCB Formation The last step is to organize the PCB statements in Step 2 into PCBs and instrument them with hypercalls. If each PCB statement is treated as a PCB block, its performance toll will significantly rise up. For each function in $\mathcal{F}$, our algorithm scans the statements identified in Step 2 with several rounds of iterations. In the first iteration, adjacent PCB statements are grouped into one PCB block. In the second iteration, the algorithm attempts to merge separated PCBs in order to reduce the total number of PCBs. If two PCBs are in the same basic block (i.e. a straight-line sequence of code with one entry point and one exit) and the number of non-PCB statements between the PCBs are less than a predetermined parameter κ, then these two PCBs are merged together with the non-PCB statement in between into a new PCB. Note that κ is used to tune the balance between the size of PCB and the number of PCBs. This iteration continues until no new PCB is generated. In the end, two hypercalls are inserted for each formed PCB as described in Section 5.4.

It is better for the driver developers to do the PCB identification since 1) they know best, and 2) it may increase the market share due to the extra security services. In addition, the identification process is only done once, and the results can be delivered anywhere. For a particular system, the hypervisor does not need to maintain a universal list (including all PCBs), while it manages the PCB list only for the drivers loaded in the system. Maintaining the PCB list that are never used will lead to unnecessary cost and may increase the TCB size, especially for the hypervisor.

5.6.2 Full I/O Path Protection

As illustrated in Figure 5.4, a full I/O path in general consists of the user-space buffers allocated by the application, the kernel-space buffers allocated by the driver and/or kernel, and the I/O interface buffer such as a DMA buffer. DriverGuard ensures the security of the latter two while a user-space protection scheme (e.g.,

Overshadow [17], SP^3 [106] or SecureME [22]) secures the first type of buffers. To have a seamless integration, the key issue is to ensure that the kernel segment of the I/O path correctly joins the intended application's user space segment, as a device is shared among multiple applications.

This issue has twofold implications. One is that the location of the user space buffer allocated by the target application must be securely passed to the driver, such that the driver can deliver (fetch) I/O data to (from) the right place. The other is that the data during the user-kernel space transition must be securely handled. When the I/O data is transferred between the application's buffer and a kernel buffer, it should be ensured that no security gap exists during the transition. In other words, both buffers should be protected either by encryption or hypervisor-based access control while allowing data flow between them.

We propose below a design integrating DriverGuard with Overshadow as illustrated in Figure 5.8. Note that Overshadow makes use of a cloaked shim which is introduced as a trusted component in user space. The shim code in Overshadow plays the role of protecting data exchange between the application and the kernel using system calls. We propose to add a new function named as *shimguard* in the cloaked shim. Shimguard in the cloaked shim handles inbound and outbound data before and after system calls and correctly locates the application buffer for I/O data.

Before an application issues a system call to activate an I/O operation, the shimguard is involved by the cloaked shim. The shimguard first updates the identity of the application, the identity of the buffer and the identity of the target driver to the hypervisor. The identity of the application is the unique *address space identifier* (ASID) maintained by the hypervisor as proposed in Overshadow. If current ASID is the trusted application, the hypervisor accepts the hypercall, otherwise it will reject the hypercall. The identity of the buffer is its memory region represented in machine address. The start and end virtual addresses of the buffer are provided by the shimguard using the hypercall, and the corresponding machine addresses are

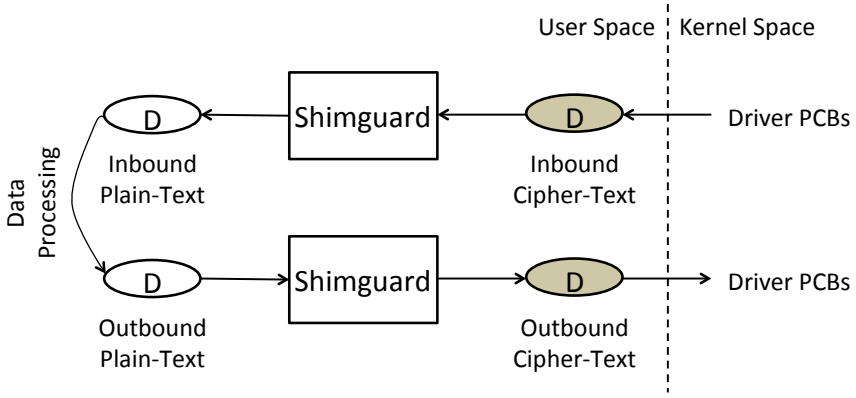


Figure 5.8: The *shimguard* helps Overshadow and DriverGuard to protect the whole life cycle of the I/O data. Note that the I/O data denoted as D in the shaded regions are encrypted either by driver PCBs or by shimguard.

collected by the hypervisor. Note that the virtual addresses alone can not used as the identity of the buffer since they may represent a different buffer in another virtual pace. The identity of the device driver can be got by using the name of the device provided through the hypercall since the device and driver mapping relationship is maintained by the hypervisor. The hypervisor generates a unique AES encryption key for the received 3-tuple of identities. Both the 3-tuple and the corresponding key are inserted into a table in the hypervisor space. For clarity purpose, we use *shim-key* to denote the AES key save in couple with the 3-tuple identifiers.

We use a read operation as an example to illustrate how to protect the I/O path starting from the device interface to the application buffer. The protection over the kernel space segment is the same as described in previous sections. When the PCB of the driver moves the I/O data into a user space buffer denoted as $Addr$, it requests the shim-key for $Addr$ from the hypervisor. The hypervisor releases the shim-key on the condition that there exists an entry in the previous table containing both the requesting driver's identity and the machine address of $Addr$. If successful, the driver PCB encrypts the I/O data and transfer the cipher text to $Addr$. When the cloaked shim is triggered to fetch the data from $Addr$, the shimguard function requests the shim-key from the hypervisor. The hypervisor releases it on the condition that there exists an entry in the previous table containing both the requesting appli-

cation's ASID and the machine address of $Addr$. If successful, the shim deciphers the encrypted I/O data and passes it to the application.

The operations for outbound data flow is similar to the description above. Note that the shim-key in different from the the encryption key used by the driver described in previous sections. The shim-key is application specific and is the same as that used in Overshadow, whereas the driver has its own encryption key. The key generation and management are trusted as it is generated by the hypervisor in the hypervisor space and protected in the guest space.

5.7 Evaluation

We implement DriverGuard and run experiments on six peripheral devices to evaluate its security and performance. The devices are a USB keyboard, a web camera, a fingerprint reader, a sound card, a printer and a graphic card.

5.7.1 Security Analysis

Driver Security

As we know drivers are usually buggy. Attackers are able to compromise the driver through these vulnerabilities to hijack the control flow or data flow of the driver to attempt to get the I/O data. Fortunately, attackers can not get the I/O data as long as the integrity of driver's PCBs are kept under the protection of the DriverGuard. Even if attackers completely control other parts of the driver, they are only able to access encrypted I/O data or are directly rejected since all these accesses are not from PCBs. Smart attackers may attempt to call a PCB to get I/O data. However, this attempt would fail because 1) the control flow of the PCB is static, and 2) I/O data is either encrypted or set protection by the hypervisor according to the design of the PCB (Section 5.3.4) when the control flow is out of PCB . The confidentiality of the I/O data is dependent on the trustworthiness of PCBs, not other parts of the driver or kernel. Therefore, attackers can not get any benefits from buggy device drivers.

ROP Attack

The Return-Oriented Programming (ROP) attack is a very powerful attack since it does not need to inject malicious code but drives legitimate code to do malicious behaviors. However, the ROP attack can not get the protected I/O data in our system due to the design of the PCB and the DriverGuard protection. More specifically, in our design, the executions of PCBs are protected by the hypervisor and the control flows of PCBs are static. Therefore, attackers can not hijack any PCB control flows. Furthermore, only the execution flows that start from *recorded start_escort* hypercalls are able to access decrypted I/O data. Any other execution flows that have no escorting request or with *unrecorded* requests are rejected.

DMA Attack

Our scheme relies on IOMMU to defend against DMA-based attacks, whereby a rootkit instructs a DMA device to read/write a memory region. IOMMU can defend against this type of attacks if the checkpoints are set on I/O page tables as well. If IOMMU is not available, an alternative approach is to intercept DMA request with shadow DMA descriptor mentioned in BitVisor [84]. Nonetheless due to its high runtime cost, the shadow DMA descriptor is more amiable to slow devices with infrequent usage, e.g. a fingerprint reader.

Interrupt Spoofing Attack

The interrupt spoofing attack are proposed in [109], which attempts to induce the device driver operating on incomplete or inconsistent data by processing spoofed interrupts. Obviously, the interrupt spoofing attack may lead to the device driver's misbehavior. However, it can not help attackers to access the I/O data, since it is only readable for trusted PCBs.

Side Channel Attacks

Side channel attacks (e.g., [102, 85]) can be used by the adversary to infer secret data. Since the adversary in our model refers to malwares residing in the guest

OS, the hardware-based side-channels such as power consumption are not feasible for the adversary. The adversary can launch other attacks by observing the timing difference between two I/O events (e.g., keystrokes) or the contents in a CPU cache, which is weak than the adversary considered in chip-card security. In addition, existing side channel attacks mainly target cryptographic data, such as a decryption key or a password. It is unknown whether generic I/O data is subject to these attacks as well.

Our current design does not take side-channel attacks into consideration. To counter these attacks, DriverGuard should deploy a special AES implementation resisting side-channel attacks. The hypervisor should clean up the CPU caches whenever a PCB is scheduled off from the CPU. It can also generate random I/O events to defeat timing analysis. The main challenge is how to deal with side-channel attacks without increasing the hypervisor's complexity and incurring more overhead.

Attacks on Multi-core Platform

On a multi-core platform, it is possible that while a PCB runs in one core accessing the I/O data, the subverted guest kernel on another core can also access them using the same page table used by the PCB. This attack can be countered using hardware-assisted virtualization supporting EPT or NPT.

The hypervisor prepares a dedicated EPTs for PCBs so that they have access permissions to those protected checkpoints. The non-PCB code such as the untrusted guest kernel use the normal EPT/NPT, in which the checkpoint regions are set as inaccessible. Whenever a PCB starts to occupy a CPU core, the hypervisor installs the dedicated EPTs (e.g., triggered by the *start_escort* hypercall) for the corresponding core. When it gives up the CPU core, the normal EPTs are restored (e.g., triggered by the *end_escort* hypercall). Since instructions on other cores do not have the dedicated EPTs, they cannot access the protected region when the PCB is in execution. Note that the EPTs are solely managed by the hypervisor. The guest

kernel does not the privilege to manipulate the EPTs.

5.7.2 Security Evaluation

To validate the design of DriverGuard, we evaluate its effectiveness in several experiments.

Known Attacks

To the best of our knowledge, the only publicly known kernel-level attacks on I/O devices are keyloggers. We have downloaded and tested three kernel-level keyloggers and none of them can successfully acquire the keystrokes. The first keylogger[6] directly reads the keyboard I/O ports 0x0060 and 0x0064 using a fake interrupt handler. It fails because the fake interrupt handler does not belong to the authorized keyboard driver PCBs. Thus, it cannot access the I/O port or get the decryption key. The second keylogger[7] modifies the keyboard driver's data structure and installs a malicious function handler. Since the malicious function handler is not admitted by the hypervisor as a PCB, it can only access encrypted keystrokes without being allowed to use the secret key. The third keylogger[8] modifies the system call table to replace *read* function with a malicious one. The malicious *read* function first calls the original *read*, and then steals data from the user space buffer that is passed as parameter. This rootkit fails because the *read* function only copies the encrypted keystroke. After the driver places the ciphertext in the application buffer and opens it for the application, the hypervisor denies all kernel level accesses.

Synthetic Experiments

We introduce three synthetic attacks to read protected I/O data, and the results show that the DriverGuard successfully prevents all of them. More specifically, in the first experiment, we attempt to modify one byte in the protected MMIO region. DriverGuard catches the write operation through a page-fault exception. DriverGuard

[6] `http://www.phrack.org/issues.html?issue=59&id=14`

[7] `http://goo.gl/DpOBc`

[8] `http://packetstormsecurity.org/files/view/25677/kernel.keylogger.txt`

rejects the operation once it verifies the caught operation is not from an identified PCB. In the second experiment, we try to read the protected I/O data in a driver buffer, where the data is encrypted. We are only able to get the cipher text since the newly introduced code is not able to get the encryption key to decrypt it. In the third attack, we introduce a piece of code to call a PCB to get the protected I/O data. The attack fails as the data is encrypted when the execution flow is out of the PCB.

5.7.3 Usage of PCB

In our experiments, we manually identify all PCBs on the source code of device drivers and the drivers in the kernel's I/O subsystems, e.g., a host controller driver. It is straightforward to identify command-PCBs and key-PCBs, because key-PCBs are introduced by DriverGuard while command-PCBs are the code accessing port I/O, MMIO or structures used by devices (e.g., frame list of UHCI). Identifying computation-PCB requires the semantic knowledge of the code. We trace the I/O data to spot code segments computing on the I/O data. Note that code segments for copying or moving data are not PCBs.

Table 5.1 lists all the involved drivers (except for the graphic driver since it uses user-level driver) used in our experiments and the number of PCBs in each of them. We find that a driver typically has only around ten PCBs and each PCB has approximately 15 lines of code without making function calls (except the encryption and decryption functions). The total PCB code only account for $1 \backsim 3\%$ of the driver code. The tiny size of PCB and its simple logic allow for high security assurance, as compared to protecting the execution of thousands of lines of driver code.

5.7.4 Performance Evaluation

We experiment with DriverGaurd on a PC with Intel(R) Core(TM)2 Duo CPU E7200 @2.53GHz, 4GB main memory, running Xen 4.0.0 and a PV guest domain with Linux kernel 2.6.31.13. DriverGuard adds around 1.7K SLOC to the Xen hypervisor. Our performance evaluation includes a cost measurement of Driver-

Driver	**Size** (LOC)	**# of PCBs**	**Avg. PCB Size** (LOC)	**Device**
keyboard	4964	11	17	keyboard
HID*	12771	13	10	keyboard
UVC driver	7838	7	11	camera
EHCI*	10011	6	15	camera
HDA-Intel	47825	8	6	sound card
Sound-core*	18722	5	4	sound card
devio	1628	7	12	printer, finger-print reader
UHCI*	7600	5	14	printer, finger-print reader

Table 5.1: The number of PCBs and the average size for each driver used in our experiments. The drivers labeled with stars are those within the kernel's I/O subsystem. The PCB size includes the hypercalls and the calls to the encryption and decryption functions.

Guard's component functions and a set of application tests with six devices. We remark that the I/O characteristic is favorable to our scheme as peripheral devices are usually much slower than the CPU. Therefore, DriverGuard does not affect the driver performance since the device speed is the performance bottleneck.

We choose 128-bit RC4 as the encryption cipher in our implementation rather than AES encryption, because RC4's compact code is easier to protect and does not significantly expand the PCB size,

Component Cost Evaluation

We instrument the DriverGuard code to measure the CPU cycles consumed by its main components including the escort hypercalls, the interrupt handler `do_IRQ`, the debug handler `do_debug`, the page fault handler `do_page_fault` and the general protection exception handler `do_general_protection` The results are shown in Table 5.2. Note that the encryption cost within a PCB comprises the overhead of the secret key access which incurs one page fault and the hypervisor's checkpoint removal.

We also test the time cost induced by DriverGuard to data movements in the guest domain, including I/O port data and memory buffer transferring. The cost is

Components	CPU cycles
do_IRQ	844
do_debug	739
do_page_fault	961
do_general_protection	1813
Encryption 1KB within a PCB	23355

Table 5.2: Cost of DriverGuard components

due to the interceptions triggered by the checkpoints. We choose two commonly used functions: *inb* and *memcpy*. *inb* reads one byte from a serial port while we run *memcpy* to copy 12K bytes from one buffer to another. Both the port and the memory buffer are protected by DriverGuard checkpoints and the functions are allowed to access. The test results are shown in Table 5.3.

	memcpy	*inb*
Normal (in CPU cycles)	1884	2738
DriverGuard (in CPU cycles)	3026	2939
overhead (%)	1142 (60.62%)	201 (7.3%)

Table 5.3: Time cost induced on the guest domain data access

As shown in Table 5.3, the overhead for protecting memory data flow is rather high (about 60%). Therefore, a severe performance drop will be seen in I/O flows involving frequent memory data movement. In fact, most device drivers are optimized to reduce the number of memory copying. A widely used practice is for the driver to maintain a large cache buffer and memory copying is invoked only when the cache is full.

Driver Performance Measurement

We test three input devices (keyboard, camera and fingerprint reader) and three output devices (printer, sound card, and the graphic card). For each device, we evaluate the performance overhead and latency for the device drivers and user applications.

Keyboard When a user presses a key on the keyboard, an interrupt is generated by the hardware. The interrupt handler of the HID driver is invoked to get the key

code and to send it into a tty buffer through the keyboard driver. Following that, an event is raised to trigger *sys_read*, which has been sleeping on the event. When being waked up, *sys_read* transfers the key value from the keyboard driver's buffer to a user space memory address. In our experiment, we measure the time cost of the interrupt handler which moves the data from the keyboard to the tty buffer. The results are shown in Table 5.4. Although the protected keyboard I/O is slower than the unprotected one, it does not affect the application because the overhead (i.e., $0.085ms$) is still negligible as compared the speed of *human keystrokes*.

	key code transfer
Normal	0.053ms
DriverGuard	0.138ms
overhead (%)	0.085ms (160.40%)

Table 5.4: overhead of a protected keyboard I/O

Camera The web camera in our experiment is managed by the default Linux UVC driver. When the camera is opened by an application, it continuously collects video data and sends them to the application. The UVC driver's interrupt handler moves and decodes the captured data from the camera into a video frame, which resides in the driver's buffer mapped to the user space. The user application can directly use the frame data like normal user-space data without any kernel-to-user-space data movement.

We run a command line program called *capture-example*[9] which reads the camera data continuously. We measure the time overhead of the UVC interrupt handler and the application's waiting time for getting new data, which is a key factor to the quality of the generated video stream. The results are shown in Table 5.5.

The interrupt handler's cost grows 10 times when under the protection of DriverGuard. The main overhead is due to the encryption on the camera data, which are 4 pages long. Nonetheless, the drivers spends much more time in waiting for the

[9]It can be downloaded from `http://v4l2spec.bytesex.org/spec/capture-example.html`.

camera's data generation. Thus the cost of the interrupt handler does not cause the overall performance degradation. We test video chatting using Empathy 2.30.2, which is an graphic instant messenger. The experiment results do not show noticeable delays to the human users.

	interrupt handler	waiting time
Normal	0.023ms	33.24ms
DriverGuard	0.259ms	33.38ms
overhead (%)	0.236ms (1026.09%)	0.14ms (0.42%)

Table 5.5: The performance of a USB camera

Fingerprint-Reader Our fingerprint reader is the Upek Touchchip fingerprint sensor. In our evaluation experiment, we choose *Fingerprint GUI* [10] as the application which uses the default Linux driver *devio* to communicate with the fingerprint reader. When the fingerprint reader is active, the driver's interrupt handler continuously loads the collected fingerprint data into its buffers, which are then fetched by *Fingerprint GUI* by calling the *ioctl* function. In our experiments, we measure the whole I/O session of fingerprint collection. The results are shown in Table 5.6.

Printer The printer in our experiments is HP Officejet 7210 and the device driver

	fingerprint collection
Normal	2.61s
DriverGuard	2.63s
overhead (%)	0.02s(0.77%)

Table 5.6: The turnaround time of fingerprint collection.

in use is *devio*. We use OpenOffice to print documents via a print-process running in the background. The print process opens the printer and issues *ioctl* to send data to the printer. After sending out the data, the print-process waits for a signal sent back by the printer to close the printer. In our experiments, we measure the turnaround time between the printer open and the printer close. The results are shown in Table 5.7.

[10] `http://www.n-view.net/Appliance/fingerprint/index.php`

	1 page	2 pages	4 pages	8 pages
Normal	15.74s	27.36s	56.65s	120.75s
DriverGuard	16.19s	27.73s	58.15s	122.40s
overhead (%)	0.45s (2.86%)	0.37s (1.35%)	1.50s (2.65%)	1.65s (1.37%)

Table 5.7: The turnaround time of file printing

Sound Card The sound card in our test is Intel Corporation 82801I (ICH9 Family) HD Audio and the driver in use is *HDA Intel*. We run the application *Totem* which plays MP3 files. Totem places its sound data into a user space buffer, which is mapped into the DMA buffer specified by the driver. When the music is in playing, Totem directly sends data into mapped DMA region in user space, and issues *ioctl* to synchronize and update information. The hardware fetches the data from the DMA buffer directly without the driver's involvement. Hence, DriverGuard is only involved in protecting the control region so that the kernel can not change the location of the DMA buffer in use.

Specifically, DriverGuard sets the sound card MMIO, the status and control region read-only after the probing stage. It rejects any update on the DMA descriptor base address. DriverGuard also denies any access to the DMA buffer from the kernel. Therefore, there is no cost for DriverGuard during music playing, though the cost in opening the sound card is high, which is shown in Table 5.8.

	sound card open
Normal	7.8μs
DriverGuard	12.3μs
overhead (%)	4.5μs (57.7%)

Table 5.8: overhead of the protected sound-card opening

Graphic Card We test DriverGuard with the graphic card. Since Xen 4.0.0 in our testing platform does not support Direct Rendering Manager (DRM), we have to run a guest Linux without DRM where the X Window sever directly manages all display outputs. The X Window server runs in the user space and does not use any kernel-level drivers. In a nutshell, it simply copies the display data to a designated

memory buffer reserved by BIOS for the graphic card.

According to the design, we implement a loadable kernel module as the Trusted Loadable Module, which collects the reserved physical region, the user-space mapping region and the page table base address of the X Window server during the system booting. To defend the kernel's data stealing, DriverGuard grants the X Window server to access these protected regions and denies all accesses from the kernel or other user processes. We test the display performance with *x11perf*, which is a

	10subs	100subs
Normal	45.6 μs	55.5 μs
DriverGuard	45.8 μs	55.5 μs
overhead (%)	0.2 μs (0.44%)	0

Table 5.9: Graphic card performance evaluation with x11pref.

graphic card performance measurement tool. We run the command *x11perf -repeat 100 -reps 10 -subs 10 100 -circulate* to measure the graphic card performance with and without DriverGuard protection. The results in Table 5.9 show the performance overhead is rather small. The reason is that when the X Window server accesses the protection region, there is only one page fault exception which is for the first access. After lifting the checkpoints, further access will not trigger any exception until it is switched off.

5.8 Summary

We have proposed DriverGuard which is a hypervisor-based system protecting I/O flows between devices and applications, especially for devices generating data or rendering data. DriverGuard protects I/O device control, I/O data transfer and a driver's data processing, against attacks from the untrusted guest kernel. It is featured with fine granularity protection, strong security assurance and low overhead. It only adds around 1.7K SLOC to the Xen hypervisor and a few lines to the driver code. DriverGuard can work jointly with user-space data protection schemes to safeguard the entire data lifecycle.

Chapter 6

Dedicated Protection on User Passwords

Password based authentication is the primary method for a remote server to check a user's identity. In a typical web authentication, a user password is transferred from the keyboard to the kernel, then to the browser before being sent out over the network to the web server through an SSL channel. One of the main threats to password authentication is kernel/application keyloggers which steal the password from its transferring path.

Any countermeasure to keyloggers must cope with both the attacks on the application which forwards the password to a remote server, and the attacks on the I/O path, namely from the keyboard to the application. Virtualization based isolation is the main approach as used in [26, 36, 32, 10], where either the browser or the entire OS is isolated as a protected environment. This approach usually incurs significant cost due to the large code to isolate and the security assurance is not strong, though it addresses other related security problems, e.g., phishing attacks. Another approach, as suggested in Bumpy [59] and BitE [58], is to use an encryption-capable keyboard to protect the I/O path and rely on the latest processor features to isolate the application. However, most commodity platforms at present are not equipped with the needed keyboard.

In this chapter, we propose a novel system to protect passwords against keyloggers in remote authentication without using a special keyboard or isolation protection like [26, 36, 32, 10]. Note that in the typical remote authentication setting, it is unnecessary for the user's platform (including the OS and the application) to know the *actual user password* as long as it can forward the authentication information to the server properly. Therefore, the high level idea of our work is that a hypervisor intercepts the user's password input; and whenever the application needs to submit the password to the server through an SSL channel, it traps to the hypervisor which performs the desired encryption. In other words, the normal SSL connection between the application and the server is split into non-cryptographic operations and cryptographic operations, such that the latter are accomplished by the hypervisor holding the password.

In our system, the cleartext password is never exposed to the operating system or the application. As a result, a keylogger can only get a ciphertext version. The system is highly efficient because no extra computation or communication cost is incurred as compared to normal password authentication, except the keyboard interception and the trapping. It is entirely transparent to the operating system, though the application needs to have a plug-in in order to split the SSL operations. Furthermore, the system is user friendly as it results in little user experience change. (Note that anti-phishing is not in the scope of our work.)

In the rest of the chapter, we present the design and implementation details of our password protection system named as *KGuard*. It is for password based web authentication using Firefox. We also report its performance in experiments with commercial websites such as Gmail. A novel building block of our system is a secure user-hypervisor interaction channel that allows a user to authenticate a hypervisor, which in itself is of research value as it addresses one of the challenges recently identified in [108]. KGuard can be extended for other password authentication systems (e.g., SSH) by replacing the browser plugin with the one for the application.

In the next section, we present an overview in Section 6.1 with the emphasis on the methodology used in our design. In Section 6.2, we describe the details of our design. The implementation details and performance results are shown in Section 6.3 and Section 6.4, respectively. We discuss several important issues in Section 6.5 and conclude this chapter in Section 6.6.

6.1 Overview

This section presents an overview of our work. We explain the design criteria and the rationale we follow, including the trust model and a high level explanation of our approach. We also show the architecture of the proposed system.

6.1.1 Design Criteria

Ideally, a password protection system should meet the following criteria. Firstly, the protection should offer the strongest security assurance. It should be able to defeat attacks from rootkits which subvert the operating system, as kernel rootkit keyloggers are not uncommon in the cyberspace.

From the practicability perspective, the protection should induce little or no modification on the operating system and is fully compatible with existing browsers. This is due to the fact that proprietary operating systems such as Windows and Mac OS are more widely used than open-source operating systems.

Furthermore, the password security should not be attained at the price of the easy-of-use of password authentication. On the user side, the protection scheme should be as simple as possible and does not require user possession of extra devices, such as a USB token and a mobile phone. On the server side, no changes should be needed. Last but not the least, the protection system should incur low cost. The cost is measured in terms of both the time delay during the password authentication session and the overall computation load on the platform. It is crucial that the user should not experience noticeable delay in an authentication session.

6.1.2 Design Rationale

In order to meet the criteria, we carefully assess a variety of design options. The foremost issue to consider is the trust model, i.e. which component in the platform can be considered as trustworthy.

Trust Model

We do not trust the operating system and applications running on top of it, in the sense that they can be compromised and attempt to steal user passwords. Therefore, safeguarding user password necessitates a root of trust which should not be subverted by rootkits. One candidate for the root of trust is the TPM chip [96], which is expected to resist all software attacks. Nonetheless, despite of its high security assurance, the TPM chip offers rather primitive and inflexible functionalities and is slow in computation. These drawbacks make it ill-suited for password protection.

In this work, we choose the hypervisor (a.k.a. virtual machine monitor or VMM) as the root of trust, as in [78, 84, 17]. The main benefit is that it allows us to develop desirable protection functions within the hypervisor, and therefore facilitates the design and the implementation. The hypervisor is not as secure as the TPM chip since several attacks have been discovered to compromise some versions of hypervisors [94, 27, 49, 69]. However, the security of the hypervisor can be ensured by three measures. Our design is based on hardware-assisted virtualization, such as Intel VT-x and AMD V, which significantly reduces the virtualization code of the hypervisor. In addition, TPM-based authenticated bootup can verify the integrity of the hypervisor when being launched. Thirdly, the hypervisor in our system is only for protection in a normal personal desktop setting, rather than a cloud server with a full-fledged virtualization for multiple VMs. Therefore, those unneeded services from the hypervisor are turned off so that only a minimal attack surface is exposed to the guest OS.

A secure hypervisor is capable to dynamically protect memory regions and I/O ports against direct malware accesses. In addition to that, the hypervisor also uses

IOMMU to enforce the similar policies against malicious DMA operations launched by malware.

Protection Method

There exist several candidate methods to protect user passwords against rootkits. One is to follow the isolation approach as shown in [56]. The execution of routines processing the password is isolated from the rest of the platform to cordon off attacks. This method is not compatible with our design criteria because of its low performance. The frequent interrupt caused by user keystrokes for password inputting induces the expensive system thrashing between the protection mode and the regular mode. In addition, the isolation approach faces the difficulty of extracting appropriate Pieces of Application Logic (PAL) due to the complexity of the kernel's keyboard input processing and the browser's web page processing. Another possible method could be to escort the password data flow as shown in DriverGuard [19]. Nonetheless, this approach requires code modifications on the drivers, which does not satisfy our compatibility requirement. Moreover, DriverGuard by itself does not guarantee the security of password in the application level.

In this work, our method is based on the characteristics of password authentication. Firstly, passwords are typically sent to a remote server through SSL/TLS. It is *not* necessary for the local host to know the password in use. Secondly, passwords are fed to the system through keystrokes which can be intercepted by the hypervisor.

Based on these two observations, the basic idea of our protection method is to intercept the password keystrokes and then securely inject them back to the SSL/TLS connection established by the browser, however, with its cryptographic operations performed by the hypervisor. Therefore, the password is encapsulated using the web server's public key following the SSL/TLS specification without any exposure to the operating system or the browser.

Security Properties

The main challenge of realizing the proposed protection method is the gap between the hypervisor and the security-conscious user. In existing platforms, a user only interfaces with the operating system through the application, e.g., a browser.

This gap entails three problems to solve. The first is about the timing for protection. It is undesirable for the hypervisor to intervene in all keyboard inputs. Ideally, the protection is only activated by the user whenever needed. The *on-demand* protection brings up the second challenge: how the user is assured that the hypervisor is protecting the password input. Note that the operating system may cheat the user by simulating the hypervisor's behavior. Last but not the least, the hypervisor's SSL traffic assembling must use a proper public key certificate for encapsulation. Ideally, the hypervisor is capable of verifying whether the certificate belongs to the intended web server.

In this work, we design a dynamic secure channel for user-hypervisor interaction which bypasses the operating system. While the hypervisor's protection mechanism is dormant, the channel allows a security-conscious user to activate it through a key combination. In addition, the channel allows the user to verify whether it is indeed active. Note that it is not necessary for the hypervisor to authenticate the origin of the keystrokes, because a faked activation key combination, e.g., from the malware instead of the user, does not lead to password leakage[1].

For the aforementioned third problem, our design achieves the same level of security as the standard browser's dealing with SSL certificates, because a certificate misuse is essentially the traditional man-in-the-middle attack on SSL. Similar to the browser's certificate verification, the hypervisor ensures that the certificate is genuine and matches the SSL connection.

[1]The faked activation key combination can be considered as a denial of service attack. It will be quickly spotted by a user because as shown later, the hypervisor will respond to the user with a secret message pre-shared with the user.

6.1.3 Architecture

We consider a platform with an operating system running on top of a hypervisor. A user uses a web browser to login to a remote server by supplying the password. KGuard is designed to protect the user password from being stolen by kernel/application rootkits. The architecture of KGuard consists of three components:

1. A secure user-hypervisor interaction channel allows the user to activate or deactivate the password protection and authenticate the hypervisor. A user toggles the protection by pressing a prescribed key combination. In response, the hypervisor securely displays (on the screen) a secret message pre-shared with the user.

2. A routine in the hypervisor intercepts user keystrokes after the protection is activated. It also validates the authentication server's public key certificate supplied by the browser and encapsulates the password using encryption.

3. A browser plugin splits the SSL connection for password submission. Specifically, it requests the hypervisor to perform the needed cryptographic operations in an SSL connection and handles other non-cryptographic operations by itself.

Note that the hypervisor only performs cryptographic operations. It does *not* establish any SSL connection with the server. In a web authentication, the browser may establish multiple SSL connections. Only the one submitting the password is split by the plugin to get the needed cryptograms from the hypervisor. The benefit of this design is that it does not entail extra computation and communication cost and it can keep the hypervisor small without including the support for SSL.

The following diagram illustrates the architecture of our password protection system.

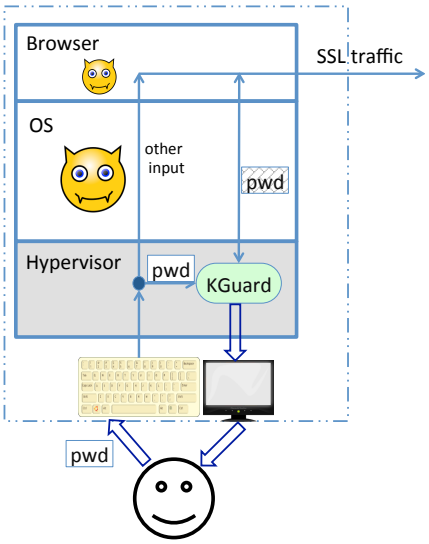


Figure 6.1: The Architecture of The Password Protection System.

6.2 Design Details

In this section, we provide the details of our design for each of the components mentioned above. Note that we use the terms "KGuard" and "hypervisor" interchangeably in the rest of the chapter since KGuard is a part of the hypervisor.

6.2.1 User-Hypervisor Interaction

The user-hypervisor interaction channel is a duplex channel. In one direction, a user sends an activation command to the hypervisor by requesting the operating system to issue a hypercall. In the other direction, the hypervisor (on receiving the user's command) securely displays a secret message on the screen. Therefore, the user can verify whether the hypervisor receives the command or not.

Hypervisor Protection Activation

There exist several approaches for activation. One alternative design is for the hypervisor to listen to a prescribed hardware event, such as keystrokes, plugging a USB device etc. These methods can bypass the operating system. Nevertheless, it requires extra work from the hypervisor which has to keep listening to all events and filter them properly. In our system, we do not favor this approach because 1)

we aim to minimize the load on the hypervisor, especially when the protection is not needed; and 2) bypassing the operating system is not necessary because no *data* is sent to the hypervisor for activation. In addition, the user can verify the activation by checking the returned secret message from the hypervisor.

In our design, the operation system is the medium transferring the user's activation command to the hypervisor. Specifically, we design an application routine, e.g. a browser extension, and install a new module to the OS, e.g. a virtual device on Windows. The application routine listens to a prescribed key combination. When the event is captured, it issues a system call which triggers the new module to issue a hypercall. This process is depicted in Figure 6.2.

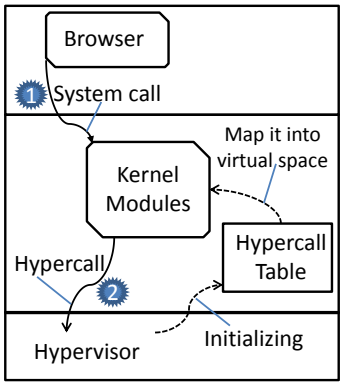


Figure 6.2: The hypercall mechanism in a HVM domain.

Note that the guest OS in an HVM is not aware of the existence of the hypervisor. The dash lines in Figure 6.2 illustrates how the hypercall mechanism is step up. The hypervisor initializes a hypercall table and then the installed OS module maps the table into the kernel space. The module exports an interface (i.e., a system call) to applications. After getting input parameters from applications via system calls, the module is invoked by the kernel automatically and forwards these parameters to the hypervisor through hypercalls, in the same fashion as the system call mechanism.

In response to the activation hypercall, the hypervisor clears the keyboard input buffer, starts to intercept the keyboard strokes as described in Section 6.2.2, and authenticates itself to the user as shown in the next subsection.

Visual Verification of Hypervisor Protection

The verification of hypervisor protection requires an output interface. To ensure its security, the output should not be captured or manipulated by malware in the guest OS. Otherwise, the guest can impersonate the hypervisor and give the user an illusion that the protection is activated.

The basic idea of our visual verification is that the hypervisor securely outputs to the monitor a secret text message chosen beforehand by the user. Note that without involving the operating system, the monitor automatically and periodically fetches the display data *directly* from a memory region called the *display buffer*, whose location is determined by the hardware [42], and then it renders them on the screen. The hypervisor shows the secret message to the user by writing it into the display buffer. To prevent the operating system from attacking the secret, the hypervisor clears the _PAGE_PRESENT attribute bit of the corresponding page table entries. As a result, any guest access will be denied by the hardware.

The details of the visual verification are described below. Initially, the user chooses a random text message as his/her long term secret shared with the hypervisor. When the hypervisor boots up, the secret message is passed to the hypervisor as a booting parameter, which is the reason why the secret has to be text. Once taking control, the hypervisor stores the secret message into its own space. Since the hypervisor boots up before the operating system, the OS is not able to access this secret. To display it on a monitor in the graphics mode, the hypervisor derives the graphic version of the secret message by using the corresponding font bitmap for each character.

After receiving the activation hypercall, the hypervisor substitutes a part of the display buffer with the secret graphic data. As a result, the user secret message is displayed on the screen. The location of the message on the screen depends on its offset in the display buffer. Note that it is unnecessary to choose random locations. In addition, the hypervisor properly sets the attribute bits of the page table entries

covering the graphic secret. Secret uploading and attribute bit setting up are an atomic operation. In other words, the hypervisor occupies the CPU without yielding it to the operating system until the attributes are set.

The hypervisor then sets up a timer whose duration is configured by the user during bootup. When the timer expires, the hypervisor restores the original display data, and finally returns the page access rights back to the guest OS.

Hypervisor Protection Deactivation

Protection deactivation requires a stronger authentication on the user than protection activation, since malware may attempt to impersonate the user to terminate the protection. Note that once the protection is activated, the hypervisor has cleared all previous data in the keyboard input buffer and intercepts all new keystrokes. As a result of the interception, no software can access the keyboard input buffer, either directly or through DMA operations, as explained in Section 6.2.2. Only the physical keyboard strokes can place inputs to the buffer.

Therefore, the hypervisor in KGuard is pre-configured with a deactivation command. Once it intercepts the command during its protection, it switches to the no-protection state by releasing the access control on the keyboard input buffer.

6.2.2 Keystroke Interception

After getting the activation key-combination command from the user, the hypervisor starts keystroke interception. Since the key stroke code is directly delivered to the guest's memory by the hardware using DMA, keystroke interception means that the hypervisor retrieves the keyboard scan code *before* the guest.

One potential approach is for the hypervisor to intercept all interrupts and intervenes if needed. The main drawbacks of this approach are twofold. This approach may fail because the guest OS can keep scanning the keyboard input buffer without waiting for the interrupt. Therefore, the guest OS may have the luck of getting the data prior to the interrupt. Secondly, the interrupt number can be shared by several devices. The hypervisor has to determine whether the interrupt is for the keyboard.

Furthermore, the interrupt by itself does not provide sufficient information for the hypervisor to locate the data.

Since locating the keyboard input buffer is an indispensable step, we let the hypervisor intercept the guest access on the keyboard input buffer, rather than interrupt interception. This method reduces the burden of the hypervisor as the guest OS manages all interrupts and is forced by the hardware to alert the hypervisor for the scan code retrieval. For this purpose, the hypervisor sets up page-table based access control on both the keyboard I/O control region storing I/O commands and the keyboard input buffer storing the scan code. IOMMU is also configured such that no DMA command can be issued to access these protected regions. Consequently, both the guest OS's keyboard I/O command issuance and its data retrieval are intercepted by KGuard. For the I/O control, KGuard emulates the operations; for the data retrieval, it replaces the user keystroke with a dummy one and saves the original input into a buffer in the hypervisor space.

The actual access control mechanism for the keyboard input buffer depends on the keyboard interface. A PS/2 keyboard usually uses PIO to transfer data whereas a USB-keyboard uses DMA. It is easy to deal with port I/O keyboards. The technique for controlling I/O port has been demonstrated in [19].

The access control for USB-keyboard is more complex due to the USB architecture.

Figure 6.3 explains the main data structures used by the so-called *Universal Host Controller* hardware. Locating the keyboard input buffer starts with a 32-bit register called FLBASEADD. Its content is the base address of a list of *frame pointers*. A frame pointer points to a list of *Transfer Descriptors (TDs)*. A TD specifies the necessary I/O parameters for one DMA operation, including the input buffer address. After completing one keyboard I/O, the guest OS must either update the current TD or insert a new TD in order to read the next keyboard input. The keystroke interception for a USB keyboard follows the steps below.

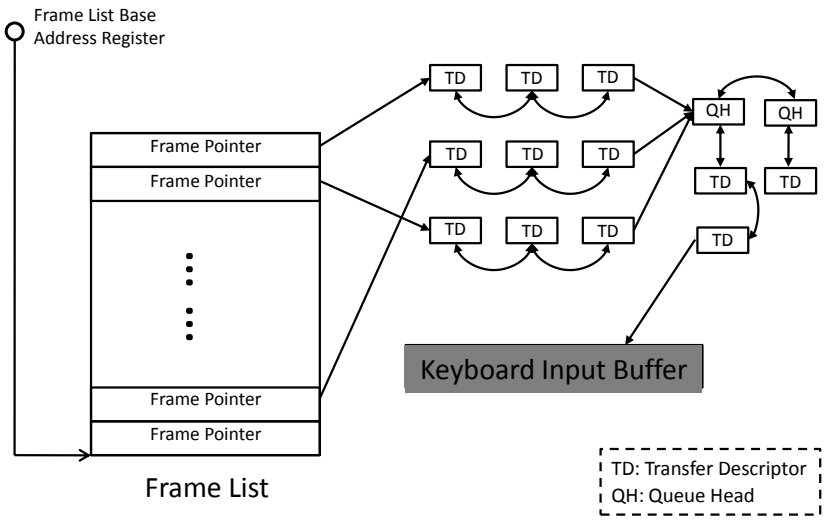


Figure 6.3: Data structures used by the USB-keyboard. The (grey) input buffer indicates that it is set inaccessible. Other (white) parts of the whole data structure are set read-only.

Step 1. KGuard freezes the present frame list and all TDs by setting FLBASEADD and all memory regions occupied by the frame list data structure as *read-only* using I/O bitmap and page table respectively. Therefore, any attempts from the guest OS to relocate the input buffer will be monitored by KGuard.

Step 2. KGuard locates the keyboard input buffer following the path used by the host controller. The keyboard input buffer is then set as *inaccessible*.

Step 3. When the guest OS attempts to read the keyboard input buffer, a page-fault is generated and passes the control to KGuard which saves the scan code (which is one password character) in the input buffer and replaces it with a dummy one, and sets the buffer as *read-write*. The guest OS can have a full access to this buffer.

Step 4. When the guest OS prepares for the next keyboard I/O by updating the TD, a page-fault is generated. In response, KGuard emulates the update operation. To prevent malware from providing faked keystrokes, the hypervisor clears the content in the keyboard input buffer, which ensures that the data fetched in Step 3 is indeed from the keyboard.

Note that KGuard responds differently on the keyboard input buffer and the I/O region because one keyboard I/O only involves one TD update but may incur multiple accesses to the buffer depending on the driver's needs. Our approach avoids unnecessary hypervisor involvements.

We further remark that the keyboard interception is only activated based on the user's command. With the cooperation from the user, the incurred cost is therefore minimal to the platform's overall performance and it is reasonable for KGuard to treat all the intercepted keystrokes as the password. Even in case that the user and KGuard are out of synchronization, no user secret is compromised and the user can easily reset the protection.

6.2.3 Handling SSL Session

A normal web authentication may involve one SSL session comprising one or multiple SSL connections. Typically, when the user clicks a button for password submission, the browser sends out the encrypted password with other necessary information through an SSL connection.

In our system, the browser is deprived of the privilege of handling the password, because the encryption of the password and other authentication information must be performed in the hypervisor space, instead of in the untrusted guest domain. For this purpose, we design a dedicated browser extension for posting authentication information to the server through SSL. To achieve both security and compatibility, the extension is only responsible for non-critical operations in the SSL connection, while all cryptographic operations, such as master key generation and data encryption, are exported to KGuard.

The extension captures the login event and initiates a new SSL connection with the server. All keys used in this SSL connection are *newly* derived and only known by KGuard and the server. Note that this new connection will be immediately closed after the login event. Therefore, the browser does not need to maintain any extra connection. In the new SSL connection, the extension obtains the server's public

key certificate. At the same time, it prepares a data blob containing all the data needed by the web server (except the password), e.g., the user name. It then submits to the hypervisor the data blob together with the server certificate. The hypervisor merges the blob with the intercepted user password, and encrypts them following the SSL specifications, on the condition that the provided public key certificate is valid. On receiving the resulting ciphertext from the hypervisor, the extension prepares the SSL data and sends them to the server. If the authentication succeeds, the server usually returns a URL with some cookies, which are decrypted by the hypervisor and forwarded to the extension. The extension then sets the cookies and redirects the browser to the URL. Now the extension terminates its SSL connection. Since neither the extension nor the browser possesses the keys for the SSL connection used for password submission, this SSL connection cannot be reused by the browser.

To avoid verbosity, we do not recite how the hypervisor generates the master key and performs the encryption, because it strictly follows the SSL/TLS specification. Out of the same reason, we do not explain how the extension prepares the data blob and the SSL traffic. However, it is worthwhile to elaborate how the server's public key certificate is validated by the hypervisor. Since we do not trust any software in the guest domain, the certificate forward by the extension to the hypervisor can be a malicious one. If the adversary has the corresponding private key, the hypervisor's password encryption will be decrypted by the adversary. We leave the details of the browser extension in Section 6.3 because it is browser specific and more relevant to usability than security.

Server Certificate Verification

Certificate verification has long been considered as a thorny problem due to the trust on the public key infrastructure. The problem is even more complicated in our case because limited information is provided to the hypervisor for the sake of minimizing the hypervisor's size. Note that phishing detection is *not* within the scope of our study. Therefore, the criterion of a certificate's validity is not whether

it matches the web server the user intends to login. Instead, a certificate is deemed as trusted as long as its root CA is trusted by the user.

In our system, the user may choose to trust all pre-loaded root CA certificates or import CA certificates she trusts. Once the user obtains a repository of trusted (root) certificates, the crux of our system is how the user securely passes them to the hypervisor. The difficulty is that the hypervisor does not have a file system and the whole guest is not trusted. The solution we propose relies on an additional trusted platform, or alternatively, the user may consider his/her platform in the initial state is trustworthy. On such a trusted platform, cryptographic tools such as OpenSSL, can be used to compute a HMAC key H_k and computes HMACs for each of the trusted certificate. Then, the user imports all trusted certificates as well as their corresponding HMAC tags into a file on the untrusted platform running with KGuard. During the platform's rebooting, the HMAC key H_k is passed to the hypervisor as a parameter. Therefore, the hypervisor knows whether a certificate is trusted by the user by checking its HMAC tag. Instead of using HMAC, the user may also apply digital signatures and pass the public key to the hypervisor, though this approach is not preferred because of its longer key and higher computation cost. Note that these above procedure is only executed *once*, i.e. for the first time using KGuard. All HMAC tags in the file are able to be reused after rebooting.

In runtime, the certificate verification proceeds as follows.

Step 1. The browser extension receives the public key certificate from the server and composes a certificate chain such that the last certificate in the chain is a trusted certificate imported by the user. For ease of description, we denote the certificate chain as $(Cert_0, \cdots, Cert_k)$ where $Cert_0$ is the server's certificate and $Cert_i$ is the issuer of $Cert_{i-1}$ for $1 \leq i \leq k$. In most cases in practice, $k = 1$ or 2. Note that only $Cert_k$ is the trusted certificate while all others are not. It is not necessary to obtain the issuer for $Cert_k$ even if it is not a root, because it is already trusted.

Step 2. The extension transfers $(Cert_0, \cdots, Cert_k, \sigma)$ to KGuard, where σ is the HMAC tag for $Cert_k$. In addition, the extension transfers the server's host name to KGuard. The transferring is accomplished by a hypercall.

Step 3. In response, KGuard first checks whether σ is a valid HMAC for $Cert_k$ using the HMAC key provided by the user during bootup. If the checking fails, KGuard rejects the certificate chain and aborts.

Step 4. KGuard then verifies the certificate chain in the same ways as the browser's verification, by treating $Cert_k$ as a trusted CA. Namely, it checks $Cert_i$'s signatures with the public key in $Cert_{i+1}$ for $0 \leq i \leq k-1$, and make sure that they are not expired, and checks whether $Cert_0$'s subject name matches the given server hostname (domain name). If all certificates pass the checking, KGuard accepts $Cert_0$ as the server's public key and uses it to encrypt the pre-master secret key in the current SSL connection.

The hypervisor calculates an HMAC value of each certificate in the verified certification chain, and returns them back to the guest if the certificate chain passes all checks. The browser inserts the certificate with its HMAC tag into the trusted certificate repository. This is to save the hypervisor's verification time when this certificate is reused in the user's future logins. Note that the new website certificates are accepted once the root certificate is trusted by the user.

6.2.4 Security Analysis

The security of the proposed password protection mechanism relies on the security of the hypervisor and the user cooperation. With the assumption on both conditions, the user-hypervisor channel ensures that the password is typed in only when KGuard is in position for keystroke interception, which saves the real password in the hypervisor space. The hypervisor and the guest space isolation enabled by the virtualization techniques prevents the guest from accessing the password. When the browser runs an SSL connection to submit the password, all cryptographic opera-

tions are performed by the hypervisor. The browser and the guest OS only get the ciphertext of the password. The hypervisor security is discussed in the Section 6.5.

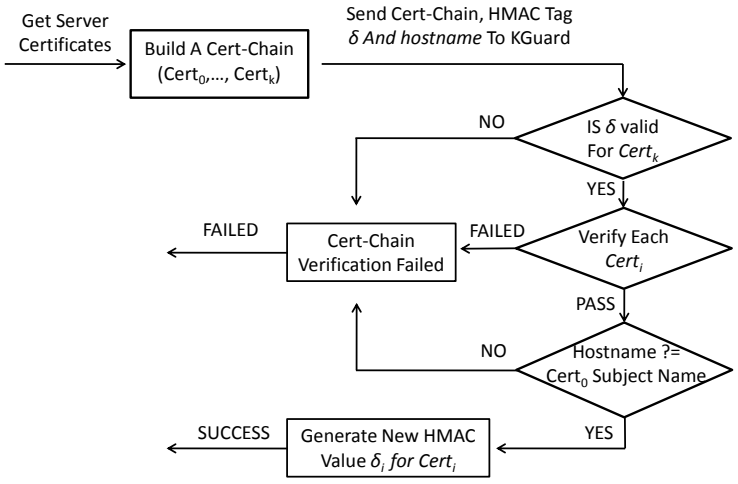


Figure 6.4: The certificate chain verification.

The hypervisor calculates an HMAC value of each certificate in the verified certification chain, and returns them back to the guest if the certificate chain passes all checks. The browser inserts the certificate with its HMAC tag into the trusted certificate repository. This is to save the hypervisor's verification time when this certificate is reused in the user's future login sessions.

6.3 Implementation

6.3.1 KGuard in Hypervisor

We have built a prototype of KGuard on Xen 4.1.0 on a desktop with an Intel(R) Core(TM) i7 CPU-860 @2.80GHz processor and 4GB main memory. We choose a USB-keyboard as the experiment device. The implementation of KGuard does not depend on the design of Xen and can be easily migrated to other hypervisors.

KGuard consists of around 1500 SLOC for its main functions except cryptographic functions. We import the needed crypto functions (about 5000 SLOC) from [76]. The main cost is due to AES and RSA algorithms which need about 3500 SLOC. Nonetheless, comparing with the Xen code base (around 225,000 SLOC),

we only increase the code size 2.885%. In fact, most of the code in Xen are not used by our system. Therefore, it is one of our future work to customize Xen for KGuard.

Visual Verification

One of the implementation issues about the user's visual verification of the hypervisor verification is to choose a proper secret message. It is similar to a password in the sense that it should not be random enough to resist dictionary attacks, and it should be easy to remember. Since the user does not type in the message at runtime, the message can be much longer than a password. For instance, we choose the string "$ApBlE@8s\_BaeuTifu10O$" as the user secret in our experiment.

Another issue is the position of the text message on the screen. We do not change the position for two reasons. Firstly, it does not enhance the security. If malware can breach the access control, it may grab the entire display buffer data. Secondly, from the usability perspective, it is inconvenient for users to find the message over the whole screen. We choose the top-left corner of the screen as the location because it is less likely to be overlapped with the web page in use.

The third concerns in visual verification is the performance overhead due to the slow speed of the display memory. It requires twice display memory access for the hypervisor to save the present content and to write the secret message. In our implementation, we use the following trick to save one display memory access. We do not save the original data. Instead, we impose the font bitmap of characters in the message upon the existing content. By performing the XOR operation, all the bits corresponding to the characters are flipped. As a result, the shape of the character is displayed on the screen. Although the content is not saved, it can be recovered by running the XOR operations again.

Note that our current implementation requires to work with the VGA compatible graphics cards.

6.3.2 Browser Extension and Plugin

Benefiting from the virtualization features of the Intel processor, we launch a hardware virtual machine (HVM) running Windows. The HVM guest domain runs a installation of Windows 7 Professional version with default configuration. We choose the popular firefox (version 3.6) as the test browser, and extend it with a plug-in and an extension.

The main part of the browser plug-in is based on CyaSSL v2[2]. It interacts with the hypervisor using hypercalls to build a separated SSL channel with a web server. Specifically, The plug-in interacts the hypervisor in the SSL handshake phase for four times: to transfer the server certificate chain; to provide the key materials for pre-master key generation; to provide the authentication data for encryption; and to provide a finish-message to terminate the SSL handshake phase. The plugin finishes the SSL protocol and forwards the server response data to the browser extension.

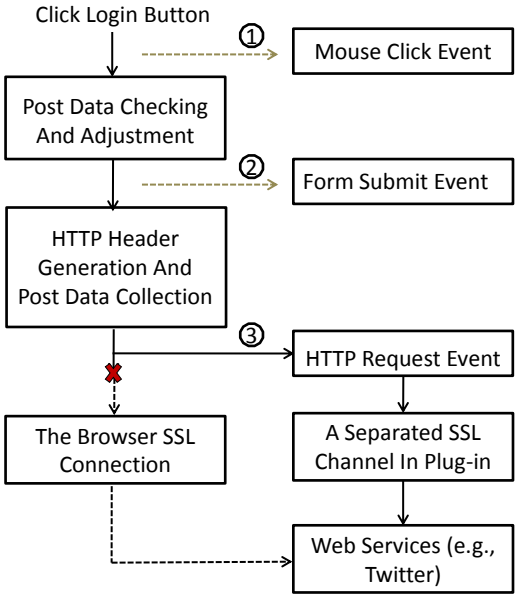


Figure 6.5: Three events generated during authentication. The third one is intercepted by the extension.

The browser extension is implemented using Firefox XML User interface Language (XUL) and JaveScript. One of the tasks of the extension is to listening to the

[2]CyaSSL is a C-Langue SSL library for embedded and realtime operating systems, and in regular desktop and enterprise environments [54]

user activation key combination and then sends a hypercall to KGuard. The other two tasks are to integrate the password protection with the browser. The first task is to intercept the authentication data submitted to the server. Since KGuard is transparent to the browser, it proceeds as usual in password submission though with a dummy password.

The events generated by Firefox after the login button is clicked are shown in Figure 6.5. We choose to intercept the *HTTP Request Event*, the last event right before Firefox is about to pass the data to the SSL layer. The benefit of this choice is that this event implies that the browser has prepared all the data (including the HTTP header) expected by the web server. Therefore, the extension does not need to handle the nuisance of gathering all kinds of POST data required by the web server.

The second task is to navigate the browser to the destination URL that is in the server response packages. After receiving the response packages returned by the plug-in from its own SSL channel, the extension extracts the cookies and the redirection URL by parsing the header and body. It updates the cookies in the browser, and requests it to refresh the current page to the redirection URL. For the following connections, no matter whether they are HTTPS or HTTP connections, the browser will send the request with corresponding cookies, and continue the web session as normal. Note that the browser is not aware of the existence of the separated SSL connection, thanks to the statelessness of HTTP and HTTPS protocols.

6.3.3 Hypercall Support In HVM

In the Windows kernel space, we build a virtual device module using the Windows Driver Kit (WDK) [61]. The module first uses the instruction *CPUID* to find registers that contain the size and the location of the hypercall table. Then it maps the hypercall table into its own memory space. Using the mapped hypercall table, the module is able to issue hypercalls to communicate with the hypervisor.

The module also exports a *DeviceIOControl* interface for application usage. Ac-

cording to the $dwIoControlCode$ parameter in the *DeviceIOControl* interface, the module can request different services by issuing different types of hypercalls to the hypervisor.

6.4 Performance Evaluation

We have run experiments and evaluated the performance and usability with legitimate web servers, including Google, Groupon, Twitter and Amazon, and Microsoft Hotmail. We divide the total authentication session into two phases to facilitate the evaluation. The first phase is user password input and the second is password submission. We have measured the time overhead in each of them. Note that our protection is the "on-demand" mode, therefore, there are no extra cost for the system when the protection is inactive.

6.4.1 Overhead for Password Input

Table 6.1 lists the time costs for the procedures taking place during a user's password inputting. The password input phase begins with protection activation and ends with protection deactivation. The main overhead is due to the hypervisor's responses to the activation/deactivation command and its interception of keyboard strokes. The activation cost mainly includes a guest system call, a hypercall, a series of access control setup, and two accesses on the display memory. The deactivation cost only includes the removal of access control on the relevant regions. The keystroke-interception cost is the CPU time spent for intercepting one keystroke. It includes two exceptions, emulation of the refreshing of TD and processing the keystroke.

Note that the user secret message is written to the display memory, instead of the main memory. Its speed is much slower than the main memory chip. Therefore, the secret message displaying dominates the overhead of protection activation. Nonetheless, it is still negligible to the user as compared to the human keystroke speed. The removal of the secret message is not considered as the overhead, be-

cause with a high likelihood, it is completed between the user's two keystrokes.

Components	Protection Activation	Protection Deactivation	Keystroke Interception	Displaying Message
Time	$1.71ms$	$3.5\mu s$	$0.12\mu s$	$1.67ms$

Table 6.1: The performance overhead for password input protection in KGuard.

6.4.2 Overhead for Password Submission

In the password submission procedure, we evaluate the extra operations introduced by our scheme, i.e. those not appearing in normal web authentication. The extra operations include the extension's HTTP Request event interception and extracting data from the login (POST) request, which cost about $4ms$ in total. Note that the extension is written in JavaScript, whose best timing granularity is in milliseconds. The extra operations also include transferring data between the guest and the hypervisor; HMAC verification for the certificate's trustworthiness. The measurement results are listed in Table 6.2.

	Event interception and data extraction	Data transferring cost during in hypercalls	HMAC computation
Time	$4ms$	$1.38ms$	$0.02ms$

Table 6.2: The performance overhead of each component for password submission.

We have also measured the turnaround time to evaluate the overall delay a user may experience with KGuard. The turnaround time refers to the period from the moment when the login button is clicked, to the moment when the browser begins refreshing the page. We have tested KGuard with Twitter and a local web server which resides in the same platform with the browser so that no network delay variation disturbs the results. The results are shown in Table 6.3. Note that the results from the tests with Twitter are not sufficiently accurate due to the large variance of network round trip time.

	Login without KGuard	Login with KGuard	Extra Cost
Twitter	$1.10s$	$1.11s$	$10ms$
Local Web Site	$201ms$	$207ms$	$6ms$

Table 6.3: The overall performance measurement in the login procedure.

6.5 Discussions

6.5.1 Hypervisor Security

The hypervisor security is the bedrock of the proposed password protection system. It is known that both the code size and the interfaces affect the hypervisor security. According to [7, 66], the size of the source code is proportional to the number of vulnerabilities (bugs). We choose Xen for our prototype building instead of the other mainstream hypervisor VMware ESXi, because the former has a smaller code size according to [88] and is open source. In principle, KGuard can also be built on those tiny hypervisors developed by researchers, such as SecVisor [78], BitVisor [84] and Nova [88]. Unfortunately, they are not supported by the Intel processor used in our platform.

As mentioned in [63], interfaces are the main source of critical errors. In the current Xen hypervisor, all default hypercalls for a HVM domain are only used during HVM loading. Therefore, we turn off all of them to enhance security to minimize the attack surface.

In the future work, we aim to reduce the hypervisor code size by removing unnecessary code. Besides the basic hardware virtualization functions, our initial study shows that the functionalities required by KGuard include: 1) memory management, including data transferring and address translation between the guest and the hypervisor; 2) access control on all I/O ports and memory regions; 3) interceptions on interrupts and exceptions; 4) basic crypto algorithms, such as RSA, AES and SHA1; 5) certain instruction emulations; and 6) asynchronization support (e.g., timer).

6.5.2 Trusted Certificate Updates

The user may need to insert or delete entries in the trusted certificate repository. It is relatively straightforward to add a new trusted certificate. The user simply calculates the HMAC value on a clean system and adds the certificate and its HMAC into the repository.

However, it is costly to revoke a trusted certificate from the repository. One solution is that the user chooses a new HMAC key and re-computes the HMAC tags for all trusted certificates excluding those revoked ones. Once the new key is updated to the hypervisor, the revoked certificates will not pass the verification. Alternatively, the user can prepare a Certificate Revocation List (CRL) whose integrity is protected by the HMAC tag. Whenever the plugin sends the server certificate to the hypervisor, the CRL is attached. The hypervisor then checks whether the certificate in use is on the CRL. Both methods have pros and cons. The former requires more user involvement while the latter increases the hypervisor's code size and causes more runtime overhead.

6.5.3 Sensitive Keyboard Input Protection

The KGuard system proposed in this chapter focuses on password protection. We can easily extend it to protect other sensitive inputs from the keyboard, such as CAPTCHA, credit card numbers or driver license numbers. KGuard is able to intercept and replace the sensitive inputs whenever the user activates the protection. By inserting them back into an SSL/TLS connection or forwarding them to a trusted domain, all sensitive inputs are free from malware attacks.

The challenge is to maintain the user's experience. For a normal password input, the browser only displays a string of '$*$'. The user feels the same even if KGuard replaces the original password with dummy ones. However, for other types of inputs, the user may feel discomfort when seeing dummy characters instead of the expected ones. Another issue on the user interface is how a user determines the correctness

of the input, since a wrong key may have been pressed accidentally. One possible solution is that KGuard echoes each input on the screen in the same ways as in the visual verification. Alternatively, KGuard can display the entire input string and ask for user confirmation. This method does not work well for protecting a large amount of sensitive inputs (e.g., private document editing) due to the heavy load on the hypervisor and the slow responses. In addition, it would add too much code into the hypervisor and possibly weakens the security strength.

6.6 Summary

We has presented a virtualization based password input protection system, which is composed of a novel user-hypervisor interaction channel, a keyboard stroke interception mechanism, and a hypervisor-based SSL client. Our method does not require specialized hardware and is fully transparent to the operating system and the browser. The prototype implementation and testing have demonstrated that the protection system incurs insignificant overhead on the platform and maintains the user-friendliness of password authentication in web services.

Chapter 7

Framework For Security Services

Based on the above proposed systems, and many other existing virtualization-based security systems, such as the systems proposed in [84, 17, 16, 41, 109, 56, 89], we summarize the common security primitives into our secure foothold (Guardian), upgrading it to be a framework/template of secure foothold for personal systems that could be directly used or customized by end users according to their demands to harden their systems, without needing to build them from scratch. To demonstrate the framework, we create four security utilities, i.e., hypervisor-based firewall, device monitoring, software runtime attestation and user present attestation for password authentication. The experiment results show that we can simply add a few lines of code to achieve all these security services.

7.1 Architecture

An overview of the framework architecture is illustrated in Figure 7.1. At the center of Guardian is the event dispatcher which interacts with a secure user interface and four security primitives. The architecture also comprises an event log, Guardian's long term secrets and state information. The functioning of Guardian is event-driven. To respond to an event from the guest or the hardware, the dispatcher dispatches it to the proper facility for processing. Note that bridging the semantic gap is not the purpose of Guardian.

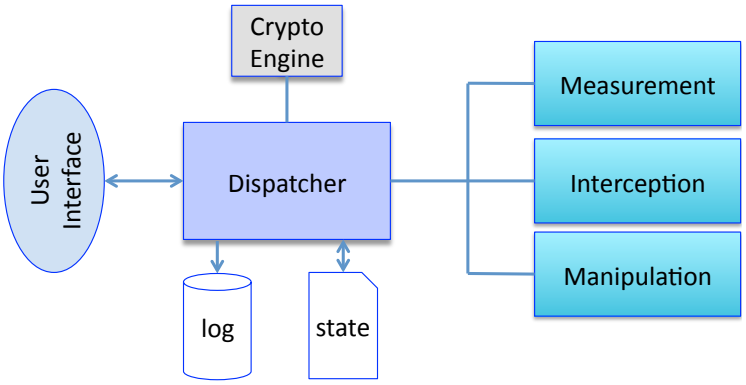


Figure 7.1: The architecture of Guardian.

7.1.1 Security Primitive

The crypto engine supports pseudo random number generation, SHA-1, AES and RSA operations. The other three security primitives consist of measurement, interception, and manipulation, which are used to access and manipulate the states of the system resources and enforce access control on them. These four primitive security functions are the building blocks for constructing high-level security utilities shown in Section 7.2.

Measurement Similar to the measurement in TPM based attestation [73], Guardian's measurement computes hash digests on memory regions or hardware resources, e.g, I/O ports and MSR values. As compared to TPM based measurement, our measurement is performed using a relatively shorter trust chain since the former usually requires a kernel level attestation agent. Furthermore, since Guardian executes in the host mode with the full privilege to access all resources in the platform, the trustworthiness of the results is the same as the trustworthiness of Guardian. In the TPM's case, its measurement can be trusted only when the attestation agent is trusted to run in a secure environment, which has weaker assurance than the TPM chip itself.

Interception Interception is often used for two purposes. One is to interpose on the guest execution so as to enforce access control policies or to monitor the guest's behavior. Guardian can set PTE attribute bits on the EPT and IOMMU

tables to regulate accesses to all memory regions, and can configure VMCS for register accesses.

The other usage of interception is for Guardian to handle external events prior to the OS. The events typically intercepted are the interrupts by peripheral devices and those by software, e.g., a system call. Most interceptions rely on the supported from the hardware virtualization technology, for instance, the events listed in [46]. Note that rather than intercepting all events, Guardian only performs interception based on its configuration and user commands.

Manipulation The manipulation function is for Guardian to modify the platform status or set up the guest context. For instance, by manipulating the VMCS, Guardian can change the CPU mode of the guest or change the access permission on a particular I/O port. Manipulation also includes emulation. Guardian supports two types of emulation: instruction emulation and event injection.

In short, measurement, interception and manipulation are the three basic security tools Guardian can use. As explained above, their functioning solely relies on the hypervisor and the hardware feature. The guest OS does not have the privilege to tamper with the operations.

7.1.2 Event Log

The event log is for the end-user's postmortem on attacks or incidents. In our design, Guardian records those security sensitive events. For instance, the build-in camera is turned on only for a fraction of a second, or the network card is turned to the promiscuous mode.

Note that Guardian does not support any file systems. Therefore, the logs are stored in the hypervisor partition in the hard disk. Guardian protects them from being read, modified or deleted by the guest. When booting up, it moves the logs from the disk into a protected memory region in the hypervisor space. To add a new log entry during runtime, it encrypts the log and writes to the memory region. Upon power off, it writes all encrypted logs back to the hypervisor-partition. Note that the

logs do not belong to the TCB, although they are protected in a similar way.

7.2 Security Utilities

When designing security utilities based on Guardian, we endeavor to deal with threats plaguing normal end-users and system administrators. To this end, we propose the following utilities. More specifically, we provide two local services, i.e., device monitor and software runtime attestation, two network utilities, i.e., user presence attestation and hyper-firewall. We also evaluate their performance. The experiment results show that the modifications on Guardian is small and they all introduce insignificant performance overhead.

7.2.1 Device Monitoring

A rootkit can misuse a peripheral device without the user's consent. For instance, it can quickly turn on the camera of a laptop to take a picture of the user and then turn it off. In a stealthy manner, it can also turn a network adaptor into the promiscuous mode so as to sniff the entire LAN traffic. We develop a Guardian utility to monitor the states of the camera and the network interface. In case of risky device usage, the end-user is alerted via the hypervisor-user interface or a beep sound. Note that the beep cannot be stopped by the adversary, because Guardian is able to intercept all accesses to that device.

Camera Control. Our design considers an external camera attached to the platform through a USB interface. (It can also be extended for a built-in camera.) The USB port is controlled by an EHCI [44] or UHCI [43] controller. In either case, a *frame list*, with its base address specified by the *PERIODICLISTBASE* register, is used to queue I/O commands. To enable the camera, the driver must insert a *transfer descriptor* or TD to the frame list. The host controller automatically fetches it from the queue and responds properly.

Upon the user's activation command, the camera control utility makes use of the interception primitive to set read-only on the region for the base register, the

frame list and the TD queue. If it detects a new TD with the open command UVC_SET_CUR for the camera, it alerts the user through a beep sound.

NIC Promiscuous Mode Control. The control on the network interface is simpler than EHCI. The Unicast Promiscuous Enabled (UPE) bit and the Multicast Promiscuous Enabled (MPE) in the Receive ConTroL Register (RCTL) are the flags that turns on the NIC's promiscuous mode. The monitoring utility intercepts the accesses to RCTL. Once the UPE bit or the MPE bit is set, an alert is raised to the user.

Note that Guardian and its utilities are not burdened with the complicated task of device management, for instance, to block illegal operations. This is to keep the hypervisor size small and more reliable.

Device Monitoring Evaluation

The device management component consists of 1.2K SLOC. Currently Guardian supports to monitor camera and network card working modes. It can be extended to support other similar devices, such as a microphone.

We experiment with a USB Logitech web camera attached on an EHCI host controller. Note that the monitoring has no effect on the camera's performance as the scheme does not intercept runtime commands and data transferring.

The network card mode monitor is built upon the Intel 82579LM Gigabit Network Card, whose registers are accessed using MMIO. The experiment results produced by network benchmark tool *netperf* [70] prove that the monitor service almost does not affect the network I/O throughout. Note that the device management service does not require any modifications in the guest kernel or device drivers.

7.2.2 Hyper-firewall

Recent attacks have shown that both application-level and OS-level firewalls can be disabled by rootkits. One solution proposed recently is the VMwall [87], which isolates the firewall in a separated domain (i.e., the Dom0 in the Xen setting). However, this approach dramatically increases the TCB size and requires the user to run

two domains concurrently.

We propose in this section a more elegant and stronger solution called *hyper-firewall* as the firewall functions in the hypervisor space. The basic idea is that a Guardian utility interposes on network I/O. It drops illegal packets if their TCP/IP headers are not compliant to the firewall policies set by the end-user through the secure UI. Since Guardian does not comprise any NIC driver, this utility does not significantly increase Guardian's code size. The main challenge is how to intercept network packets in an efficient way. Before presenting the details, we briefly explain the network I/O mechanism.

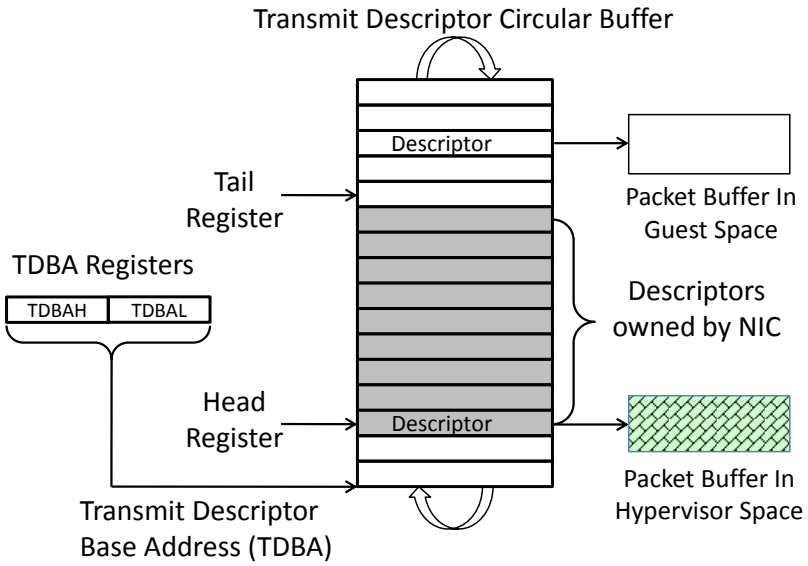


Figure 7.2: The transmit descriptor circular queue used by the NIC.

The packet transmission mechanism is illustrated in Figure 7.2. The NIC makes use of a ring buffer (essentially a circular queue) to store *transmit descriptors* which point to the packets to transmit. The ring buffer has its base address saved in the TDBAL and TDBAH registers, has its size saved in the TDLENL and TDLENH registers, and has a head register and a tail register pointing to the queue head and tail respectively. The NIC always dequeues the descriptor pointed by the head register, and then fetches the corresponding packet. After retrieval, it advances the head pointer. The tail pointer is maintained by the device driver. To send a new packet,

the driver enqueues one or multiple descriptors. Then, the tail pointer is also advanced. The NIC only uses the descriptors between the head and the tail. It stops transmission when the two pointers collide.

The packet receiving mechanism is analogous to the transmission mechanism. It also has a ring buffer storing *receive descriptors*, and has its own base address registers, length registers, and the head and tail registers. Initially, the driver allocates a set of fixed length DMA buffers, and enqueues the corresponding descriptors into the ring queue. When receiving packets, the NIC stores them into those pre-allocated DMA buffers, updates the corresponding descriptors, and advances the head pointer accordingly. Finally, it throws out an interrupt to notify the driver to fetch the packets according to the descriptors. Since the packet sending and receiving mechanisms are different, we design two interposition schemes, respectively. Note that the registers used by NICs may be different. To support all NICs, we can provide a profile which can provide necessary information for Guardian to understand register meanings.

Outbound Packet Filter Guardian uses the EPT to intercept all write accesses the TDBAL, TDBAH, TDLENL and TDLENH registers so that Guardian can always locate the legitimate ring buffer. Similarly, it sets up the EPT and IOMMU tables, such that the head register can only be updated by the NIC[1], and all accesses to the tail register are intercepted by Guardian. Lastly, it sets the entire ring buffer as read-only.

When a write access to the ring buffer is intercepted by Guardian, it checks whether the write overwrites an existing descriptor which has not been fetched by the NIC. If so, the access is blocked; otherwise, Guardian emulates the write. When a write access to the tail register is intercepted, Guardian performs the following. (1) It checks whether the packets pointed by the descriptors between the present tail and the new tail are compliant with the firewall policies; (2) It copies all legal packets

[1]In the current hardware specification, the driver is not able to instruct the NIC to update the header register

to the hypervisor space and updates those descriptors accordingly so that the NIC can fetch them from their new locations; for illegal packets, it sets the packet-length field in their descriptors as zero; (3) It emulates the tail update.

Once the packets are moved to the hypervisor space, their descriptors are not allowed to be changed. Note that packets are much smaller than a memory page. Therefore, relocating them into the hypervisor space avoids undesirable page faults as compared to protecting them in the guest space.

Inbound Packet Filter The inbound packet filter mechanism is similar to its outbound counterpart. By enforcing access control on those control registers and the ring buffer for the receiving descriptor, Guardian locates the DMA buffers allocated by the driver. To retrieve a packet, the driver first fetches the receive descriptor which triggers a page fault. Guardian then performs the packet inspection according to the firewall policies, and drops illegal ones.

Hyper-firewall Evaluation

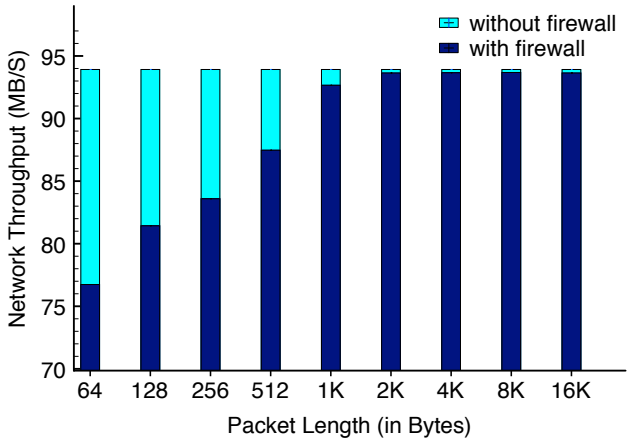


Figure 7.3: The benchmark results with and without hyper-firewall.

The packet filter service is built on the Intel Corporation 82579LM Gigabit Network Card, and does not add any code into the guest OS. Current hyper-firewall supports adding policies on inbound and outbound packets. For the outbound packets, hyper-firewall restricts the region of the target destination (e.g., external IP addresses), and for the inbound packets, hyper-firewall restricts the connection ports

(e.g., SSH port 22). All hyper-firewall policies can be enabled and disabled through the RTSUI. All experiments show the hyper-firewall works well. We tested the network I/O performance with benchmark tool *netperf* [70]. When we only enable outbound policy, the performance results show that our hyper-firewall only introduces (0.096% - 0.064%) performance overhead; when we enable inbound and outbound policies, the hyper-firewall introduces (18.29% - 0.26%) performance overhead. Note that the short packet setting generates more interceptions. Thus its performance is relatively low. Note that the monitoring of NIC does not affect the I/O speed of other derives. The packet filter service only adds $0.9K$ SLOC into Guardian.

7.2.3 Software Runtime Attestation

A rootkit may tamper with the execution of a process. For instance, the rootkit starves the antivirus process or manipulates its execution flow by inserting malicious code or running ROP (Return Oriented Programming) attack. We develop a Guardian-based attestation mechanism for a challenger (in a trusted environment) to check the runtime states of a process over a period of time.

Our goal is *not* to protect a process from being manipulated, but to monitor the runtime states. In our design, Guardian plays the role of a trusted observer securely reporting the runtime measurements to the challenger. It is the latter's task to determine whether the process runs properly and to take actions if necessary.

Described below are the details of the attestation scheme taking place between an attester platform equipped with Guardian and a remote challenger, e.g., a VPN gateway. To check the runtime context of a target software P on the attester, the challenger sends the program's identity to an *untrusted* agent in the attester's kernel level. The agent then obtains the CR3 register value for the corresponding process and passes it to Guardian.

Guardian measure P's execution within a time window consisting of multiple CPU occupations. Out of the performance consideration, Guardian does not per-

form measurements for all occupations. Instead, for each occupation, it tosses an independent random coin (with a pre-configured probability ρ) to decide whether to measure the context. Since the coin is kept secret by Guardian, the guest has no advantage in predicting which occupation is measured.

A runtime context measurement encloses (1) all CPU registers; (2) the user stack; (3) the code page for the current execution with N (e.g., 2) randomly chosen code pages from nearby regions. For every occupation, Guardian also measures the consumed CPU cycles. When the time window expires, Guardian computes c as the sum of consumed CPU cycles, and compiles all measurements of sampled CPU occupation into one data blob $\mathcal{D}$. It computes a signature σ upon $(c, \mathcal{D}, Timestamp)$ using its RSA private key. It also prepares a message file $\mathcal{M}$ which consists of $c, Timestamp$, the snapshots of CPU registers, the stacks and the addresses of all measured code pages. In the end, $(\sigma, \mathcal{M})$ are passed to the attestation agent and forwarded to the challenger. Since the challenger has P's binary code (with the dependent libraries), she recovers $\mathcal{D}$ from $\mathcal{M}$ by using the sampled code page addresses. By enclosing the addresses instead of the actual pages, the size of $\mathcal{M}$ is greatly reduced.

It is *not* straightforward to measure process P' context as it appears. With an illustration in Figure 7.4, we explain our design details below. Guardian intercepts every CR3 update so that it can detect when P is about to occupy the CPU by checking the new CR3 value and when it is about to leave the CPU. At both moments, Guardian gets the clock reading and obtains the occupation duration. However, CR3 switches are *not* the right moments for measurement. When CR3 is about to load P's page tables, P's context is actually not loaded yet. Therefore, a malicious kernel may cheat after CR3 loading. When CR3 is about to offload P's page table, the present code is the kernel's scheduling routine. The malicious kernel can cheat as well. We remark that the kernel can not manipulate the timer interrupts which is a hardware interrupt. Therefore, when the coin toss indicates a measurement, Guardian intercepts the timer interrupts, one of which will trigger the measurement.

A typical CPU time slice for a process is in the range $10ms - 200ms$ [95], while the timer interval is $1ms$ or smaller. Since one occupation comprises of multiple timer interrupts, Guardian chooses a random one for measurement, so that the malicious kernel can not predict the timing.

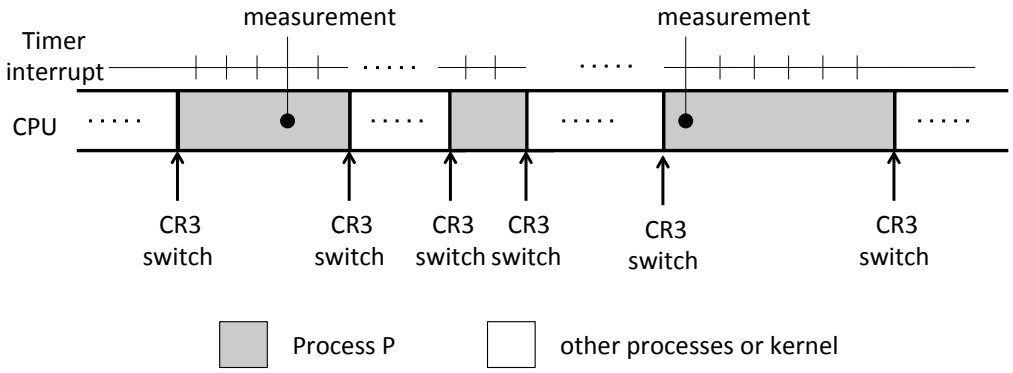


Figure 7.4: An illustration of probabilistic measurement in software runtime attestation.

In summary, the attestation is essentially to randomly sample the runtime context. The whole execution is divided into multiple occupations and one occupation comprises multiple timer interrupts. Guardian uses a probability token to determine which occupation to measure, and then use another probability token to determine interrupt to use. These random tokens can be pre-computed in order to save runtime cost.

Verification of Runtime Attestation

The challenger may perform the basic hash-comparison verification as in the TPM-based attestation scheme, from which the challenger knows which code pages are executed. To have stronger security assurance, the challenger can make uses the call graph (or control flow graph) to do further verification. With the pre-computed call graphs for the target process and its dependent libraries, the challenger locates the corresponding functions for sampled IP values in the temporal order returned by Guardian, and validates whether there exists a valid execution path crossing those functions.

For platforms enabling the Address Space Layout Randomization (ASLR)

mechanism[2], the challenger needs the runtime memory layout (e.g., the layout information in the */proc/[pid]/maps*), which can be collected and reported by a loadable-module in the guest.

Comparing with TPM-based attestation, which usually focuses on the load-time integrity and/or static properties, the hypervisor-based software runtime attestation provides richer semantics with runtime information, from which the challenger could make right and timely response for the identified misbehaviors. As mentioned earlier, our attestation requires a much shorter trust chain than TPM-based attestation.

We acknowledge that runtime attestation may *not* capture all execution misbehaviors due to the performance considerations and the time gap between two snapshots. Therefore, some tricky attacks may be missed if they can clean the traces before the next measurement. To increase the accuracy, the end-user can increase the frequency of the sampling. In the extreme case, the end-user can configure Guardian to measure each single step of the process. We leave it as our future work to devise more efficient attestation scheme with stronger security assurance.

Software Runtime Attestation Evaluation

In our experiment, the system gets the HPET-supported timer interrupt every $1/4$ms, and the target process "firefox" is roughly scheduled per $2 - 3$ms. An untrusted loadable kernel module forwards request and response messages between Guardian and the challenger. Each time the target process occupies the CPU, it has $1/\rho$ chance to be measured. The parameter ρ is configurable and currently chosen as $0x0100$. Each runtime states measurement produces 4280 bytes log data. In our experiment, the monitor time on the target process is about 524.21s, and the CPU-occupation time is about 98.54s. Figure 7.5 illustrates the distribution of the measured code pages and the really-used code pages. During the monitor period, the target process uses 1647 code pages, where 419 pages are measured by Guardian. As illustrated

[2]Note that the current commodity operating system (e.g., Windows and Linux) only randomize dynamic-link libraries.

in Figure 7.5, some pages are not measured in the selected time period. Therefore, some attacks may be successful in our measurement setting. It is an interesting topic to improve the scheme to make sure that it precisely measures the execution of a process with low performance overhead.

Figure 7.5: The benchmark results with and without hyper-firewall.

Note that the hash function used in the attestation is SHA1, and the the length of the private key in the sign operation is 1024 bits. The timestamp is directly read from Real-Time Clock (RTC) in the COMS. The software runtime attestation services only adds 0.5K SLOC into Guardian. Table 7.1 shows the performance results on the following operations: 1) time measurement when the target process on CPU and off CPU, 2) runtime states measurement, and 3) the sign operation.

Operations	**Runtime Cost**
On-CPU Time Measurement	0.03μs
Off-CPU Time Measurement	0.03μs
Runtime States Measurement	0.17ms
Sign	2.64ms

Table 7.1: The runtime cost of each operation in the software runtime attestation service.

7.2.4 User Presence Attestation For Password Authentication

Password authentication is another area which can benefit from using a hypervisor. One example is KGuard [18] which protects the password secrecy against rootkits. Another possible application is that the hypervisor generates the second factor by storing a secret or computing a one-time PIN.

Instead of protecting password secrecy, our interest is to design an authentication system resilient to password theft. The idea of our system is that Guardian attests to the authentication server that the user is present at a particular moment. This can be applied for accessing an organization's critical servers, as it greatly mitigates the damage of password theft either by malware or through social engineering. An outside adversary knowing the password can not login using his own computer due to the absence of Guardian with the certified RSA private key. Neither can the malware residing in the legitimate user's computer, because Guardian does not vouch for it.

The design is as follows. The end-user logins to the server in the normal way, except that she activates the user-presence attestation before entering the password, and deactivates it afterwards, both of which are through the secure hypervisor-user interface in Section 3.2.2. To attest to user-presence, Guardian locks the keyboard and intercepts the keyboard I/O as in running the RTSUI. Namely, it sets the EPT and IOMMU page tables so as to intercept all direct accesses from the guest and to block all DMA accesses (except from the keyboard). This access control is not for protection keystroke secrecy. Instead, it ensures that the only allowed write access to the I/O buffer is the keyboard's DMA operation. When Guardian intercepts the guest interrupt handler's read access to the I/O buffer, it checks whether the present key-code is different from the prior one. (Note that typing a key generates two interrupts with two different key-codes for key-down and key-up, respectively.) If so, Guardian generates an RSA signature on the current time; otherwise, Guardian does not produce any signature. The current time is directly collected from hardware

(i.e., COMS), whose interfaces (i.e., I/O ports) are enforced access control using VMCS by Guardian, so that the guest can only read the time information but can not manipulate it. Guardian's signature is then exported to the guest though a hypercall and forwarded to the server. The server verifies the signature and checks whether the gap between the signed time and the time of authentication is acceptable.

It is well-known that the drawback of using a clock in attestation is that it leaves a short time window for attacks. One remedy is to use a challenge-response scheme, which however incurs one additional round of network flow.

Our scheme can be integrated with authentication schemes without a graphic interface, e.g., a secure terminal. It is compatible with automatic login, since the user can still activate attestation and type the keyboard.

Note that both the runtime attestation and the user presence attestation require that Guardian's RSA public key can be correctly verified by the challenger. For cooperate users, the public keys in use can be certified through a common PKI. We also remark that the proposed user presence authentication is not compatible with automatic password submission.

User Presence Attestation Evaluation

The user presence attestation service is built using a Dell USB-keyboard attached on the EHCI host controller. A loadable module listens to the activation command (e.g., CTRL+ALT+PAGEUP). The sign algorithm is the same as the one in the software runtime attestation. We built a client and a server to simulate the login procedure. Before typing into the user name and password, the user pressed the activation key combination to enable the user-presence-attestation service. Guardian signed the timestamp of the first keystroke. Later the server checked the username, password and the login-timestamp to verify if the user is able to login. The experiments show that the user presence attestation service can be easily integrated with current communication protocol (e.g., SSL/TLS) to enhance the login security. Note that the user presence attestation service adds 0.2K SLOC into Guardian.

Chapter 8

Conclusion and Future Work

In this dissertation, we proposed several systems to harden a system. Specifically, we designed and implemented a lightweight and reliable hypervisor *Guardian* as the system secure foothold, which is the first bare-metal hypervisor with integrity and availability guarantees. Guardian leveraged virtualization technology and a secure boot and shutdown mechanism to protect itself in the whole life cycle. Moreover, we extended Guardian to be a framework of secure foothold, which consisted of summarized common security primitives for facilitating our proposed systems and other security services in the future. Built upon Guardian, AppShield created isolated execution environments for protecting critical applications. In an IEE, the secrecy and integrity of code and data together with the execution integrity are guaranteed. In addition, the application can use the memory and issue system calls in the same fashion as in a normal setting.

As the inputs and outputs of the protected applications are not protected by Appshield, we proposed *DriverGuard* to protect them by building trusted paths between the protected application and hardware input/output devices. DriverGuard is a generic I/O protection system protecting all kinds of I/O flows with combination of cryptographic and virtualization techniques. We also proposed a dedicated system *KGuard* to protect passwords in the increasingly popular online services. KGuard efficiently and securely built a trusted path from the keyboard to the remote web

server, against all system key-loggers, as well as all kinds of network eavesdroppers. Note that the trusted paths built by DriverGuard and KGuard did not leverage any special hardware.

We implemented the prototypes of all the above systems, and evaluated their performance overheads. The experiment results showed that the performance costs on CPU computation and device I/O are insignificant.

8.1 Look into the Future

One of the main reasons why malware is rampant in today's computers is the failure of the operating system's security protection, which in turn is caused by its enormous code size and complexity. A strategic architecture change by taking security into consideration may combat malware more efficiently and effectively. It is not impossible that the future platform architecture encloses the hypervisor as the secure foothold functioning as a trustworthy security delegate between the operating system and hardware. The OS and the hypervisor jointly form a holistic security framework against malicious software. We do not expect the hypervisor to protect the OS from being subverted (which is intractable in our view). Instead, the OS and the hypervisor can jointly provide reliable access control and monitoring services for applications and the enduser. The operating system can define security policies based on the user demands, while the hypervisor enforces them by relying on the hardware. Since policies are essentially data, it is easier to check their integrity. In addition, the hypervisor can also provide a series of security services to the OS and applications in an on-demand fashion, such as a strong isolation, whereby the user has the flexibility to choose between security and performance.

The systems and techniques proposed in this dissertation has laid a solid foundation towards trustworthy systems. In the following, we attempt to propose new security services based on them or extend them to harden new platforms.

- **Whole Platform ROP Defence.** Return-Oriented Programming (ROP) is a

sophisticated exploitation technique that is able to drive target applications/OSes to perform arbitrary unintended operations by constructing a gadget chain reusing existing small code sequences (gadgets). To detect and prevent ROP attacks, many solutions are proposed. However, none of them is the whole system protection scheme, where the proposed scheme protects all user applications and the OS simultaneously. The hypervisor beneath OS is able to protect itself from all guest ROP attacks, and at the same time, it is able to efficiently access both kernel and user spaces in real time. All these make it possible to propose a hypervisor-based scheme to support the whole system against ROP attacks. In the future work, we aim to design and implement a whole platform protection system to efficiently defend against ROP attacks.

- **Application Protection with Availability Guarantee.** All existing in-VM solutions that are able to protect applications through isolated execution environments *cannot* ensure the availability of the protected applications. It means that the compromised OS is able to shutdown or halt the protected applications, preventing them from occupying the CPU to run their services. If we put the scheduler which can explicitly put the protected applications on the CPU into the hypervisor space, the size and the complexity of the hypervisor will increase. Moreover, the frequent hypervisor-guest context switches will dramatically reduce the performance of the whole platform, especially for the protected applications. In the future work, we aim to seek a solution to achieve the availability with low performance overhead and minimum TCB expansion.

- **File Access Control On Untrusted OS.** Traditional file access control is an OS-based mechanism to prevent illicit access from applications. The security of the access control completely relies on the trustworthiness of the OS. Once the OS is compromised, kernel rootkits will freely access/modify/delete any files. To enforce file access control upon an untrusted OS, a hypervisor-based

file access control is needed. To build such mechanism, the main challenges are from the semantic gap and the performance consideration. Specifically, a hypervisor is usually not equipped with disk drivers and file systems due to security considerations (i.e., keeping hypervisor small and simple), which cause that the hypervisor misses the file-level information even if it is able to intercept disk I/O. In addition, the frequent context switches due to the interception on the disk I/O will dramatically reduce the speed of disk I/O and the CPU utilization. In the future work, we aim to seek a solution to efficiently and reliably enforce file access control.

- **Secure Foothold for Mobile.** Mobile phones are integral to our daily lives. A mobile phone usually contains a lot of sensitive personal information and many third-party applications, which attracts the adversary's attention. At the same time, the new mobile phones are designed to efficiently support virtualization and TrustZone technologies simultaneously. All these make it possible to deploy new virtualization-based and/or TrustZone-based secure foothold to harden mobile systems. Based on the new secure foothold, many security services could be built. In the future work, we aim to seek a way to build a new reliable and lightweight mobile secure foothold to facilitate the design and the implementation of mobile security services.

- **Secure Foothold for Cloud Instance (VM).** Our current technology is able to create a reliable secure foothold for personal computers, but it will fail in the cloud computing setting. Specifically, in a cloud platform, a hypervisor has been executed before the guest instance (VM) starts to run. In this way, the secure foothold of the guest instance is not able to get the highest privilege of the whole system that has been occupied by the platform hypervisor. Facing such situations, in the future work, we aim to seek a new technology to create a reliable secure foothold for the guest instance, and make it coexist with the hypervisor of the platform.

Bibliography

[1] AMD. Secure virtual machine architecture reference manual. Technical report, 2005.

[2] T. W. Arnold and L. P. Van Doom. The ibm pcixcc: a new cryptographic coprocessor for the ibm eserver. *IBM J. Res. Dev.*, 48(3-4):475–487, May 2004.

[3] S. Arvind, P. Adrian, v. D. Leendert, and K. Pradeep. Swatt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[4] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. Hima: A hypervisor-based integrity measurement agent. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 461–470, Washington, DC, USA, 2009. IEEE Computer Society.

[5] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[7] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation0. *Commun. ACM*, 27:42–52, January 1984.

[8] K. Borders and A. Prakash. Securing network input via a trusted input proxy. In *Proceedings of the 2nd USENIX workshop on Hot topics in security*, HOTSEC'07, pages 7:1–7:5, Berkeley, CA, USA, 2007. USENIX Association.

[9] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, Oct. 2008.

[10] S. Bugiel, A. Dmitrienko, K. Kostiainen, A.-R. Sadeghi, and M. Winandy. Truwalletm: secure web authentication on mobile platforms. In *Proceedings of the Second international conference on Trusted Systems*, INTRUST'10, pages 219–236, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] K. R. B. Butler, S. McLaughlin, T. Moyer, and P. D. McDaniel. New security architectures based on emerging disk functionality. *IEEE Security and Privacy Magazine*, September 2010.

[12] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[13] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In M. T. Jacob, C. R. Das, and P. Bose, editors, *HPCA*, pages 1–12. IEEE Computer Society, 2010.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In A. Keromytis and V. Shmatikov, editors, *Proceedings of CCS 2010*, pages 559–72. ACM Press, Oct. 2010.

[15] S. Checkoway and H. Shacham. Iago attacks: why the system call api is a bad untrusted rpc interface. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[16] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P. Yew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Technical Report FDUPPITR-2007-0801, Parallel Processing Institute, Fudan University, August 2007.

[17] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *(ASPLOS '08)*, Seattle, WA, USA, Mar. 2008.

[18] Y. Cheng and X. Ding. Virtualization based password protection against malware in untrusted operating systems. In *Proceedings of the 5th International Conference on Trust & Trustworthy Computing*, Vienna, Austria, 2012. Spring.

[19] Y. Cheng, X. Ding, and R. H. Deng. Driverguard: a fine-grained protection on i/o flows. In *Proceedings of the 16th European conference on Research in computer security*, ESORICS'11, pages 227–244, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Y. Cheng, X. Ding, and R. H. Deng. Appshield: Protecting applications against untrusted operating system. Technique Report, 2013.

[21] Y. Cheng, X. Ding, and R. H. Deng. User oriented computer protection with hypervisor as root of trust. submitted to Network and Distributed System Security Symposium (NDSS), 2013.

[22] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 108–119, New York, NY, USA, 2011. ACM.

[23] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor.

[24] I. CORPORATION. Intel trusted execution technology (intel txt) c software development guide. Dec 2009.

[25] S. P. E. Corporation. Spec cint2006. http://www.spec.org/.

[26] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Proceedings of IEEE Symposium on Security and Privacy*, 2006.

[27] CVE-2008-0923. http://cve.mitre.org/cgi-bin/cvename.cgi-?name=cve-2008-0923, 2008.

[28] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS&12, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, UNITED STATES, 02 2012.

[29] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, July 2003.

[30] A. Filyanov, J. M. McCune, A.-R. Sadeghi, and M. Winandy. Uni-directional trusted path: Transaction confirmation on just one device. In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks*, June 2011.

[31] S. Fleming. Accessing pci express configuration registers using intel chipsets. otechnical report, 2008.

[32] S. Gajek, H. Löhr, A.-R. Sadeghi, and M. Winandy. Truwallet: trustworthy and migratable wallet-based web authentication. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, STC '09, pages 19–28, New York, NY, USA, 2009. ACM.

[33] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 168–178, New York, NY, USA, 2008. ACM.

[34] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 193–206, New York, NY, USA, 2003. ACM.

[35] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. Eli: bare-metal performance for i/o virtualization. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, New York, NY, USA, 2012. ACM.

[36] C. Grier, S. Tang, and S. King. Secure web browsing with the OP web browser. In *Proceedings of IEEE Symposium on Security and Privacy*, 2008.

[37] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The nizza secure-system architecture. In *Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on*, pages 10–pp. IEEE, 2005.

[38] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[39] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 346–355, New York, NY, USA, 2006. ACM.

[40] Hewleet-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification. (Revision 3.0b), Oct. 2006.

[41] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM.

[42] IBM. IBM VGA Technical Reference Manual. Website. http://www.mca-mafia.de/pdf/ibm_vgaxga_trm2.pdf.

[43] Intel. Universal host controller interface (uhci) design guide. Mar. 1996.

[44] Intel. Enhanced host controller interface specification for universal serial bus. Mar. 2002.

[45] Intel. Intel i/o controller hub 9 (ich9) family datasheet. 2008.

[46] Intel. Intel 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. Oct. 2011.

[47] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[48] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[49] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[50] I. Z. R. Lab. Security on a stick, Oct. 2008.

[51] H. Langweg. Building a trusted path for applications using cots components. In *In Proceedings of NATO RTO IST Panel Symposium on Adaptive Defence in Unclassified Networks*, 2004.

[52] Y. Li, J. M. McCune, and A. Perrig. Viper: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 3–16, New York, NY, USA, 2011. ACM.

[53] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 178–192, New York, NY, USA, 2003. ACM.

[54] S. C. Limited. CyaSSL Embedded SSL Library. http://www.yassl.com/yaSSL/Products-cyassl.html.

[55] A. Lineberry. Malicious code injection via /dev/mem. In *Black Hat*, Mar. 2009.

[56] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[57] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[58] J. M. McCune, A. Perrig, and M. K. Reiter. Bump in the ether: a framework for securing sensitive user input. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 17–17, Berkeley, CA, USA, 2006. USENIX Association.

[59] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, Feb. 2009.

[60] C. G. Michael, W. Zhi, S. Deepa, L. Jinku, J. Xuxian, L. Zhenkai, and L. Siarhei. Transparent protection of commodity os kernels using hardware virtualization. In *SecureComm*, pages 162–180, 2010.

[61] Microsoft. About the Windows Driver Kit (WDK). Website. http://goo.gl/DfSRi.

[62] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27(6):71–80, Nov. 2002.

[63] D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 151–160, New York, NY, USA, 2008. ACM.

[64] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, Feb. 2005.

[65] D. A. S. d. Oliveira and S. F. Wu. Protecting kernel code and data with a virtualization-aware collaborative operating system. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 451–460, Washington, DC, USA, 2009. IEEE Computer Society.

[66] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 55–64, New York, NY, USA, 2002. ACM.

[67] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society.

[68] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. The perseus system architecture. In *VIS*, pages 1–18, 2001.

[69] W. Rafal, R. Joanna, and T. Alexander. Xen 0wning trilogy. Technical report, 2008.

[70] J. Rick. Network Performance Benchmark Tool - Netpref. http://www.netperf.org/netperf/.

[71] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[72] R. Sahita, U. Warrier, and P. Dewan. Dynamic software application protection. *Intel Corporation, Apr*, 2009.

[73] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[74] R. Santelices, Y. Zhang, S. Jiang, H. Cai, and Y. jie Zhang. Quantitative program slicing: Separating statements by relevance. Technique report, Apr. 2012.

[75] S. Saroiu and A. Wolman. I am a sensor, and i approve this message. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, HotMobile '10, pages 37–42, New York, NY, USA, 2010. ACM.

[76] L. Sawtooth, Consulting. Ctaocrypt embedded cryptography library. http://www.yassh.com/yaSSL/Docs_CTao Crypt_Usage_Reference.html.

[77] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, WiSe '06, pages 85–94, New York, NY, USA, 2006. ACM.

[78] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[79] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 1–16, New York, NY, USA, 2005. ACM.

[80] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In S. De Capitani di Vimercati and P. Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, Oct. 2007.

[81] J. Shapiro. *EROS: A capability system*. PhD thesis, University of Pennsylvania, 1999.

[82] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM.

[83] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[84] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 121–130, New York, NY, USA, 2009. ACM.

[85] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 25–25, Berkeley, CA, USA, 2001. USENIX Association.

[86] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 112–122, New York, NY, USA, 2007. ACM.

[87] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 39–58, Berlin, Heidelberg, 2008. Springer-Verlag.

[88] U. Steinberg and B. Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the European Conference on Computer Systems*, 2010.

[89] R. Strackx and F. Piessens. Fides: selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[90] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM.

[91] K. Sun, J. Wang, F. Zhang, and A. Stavrou. Secureswitch: Bios-assisted isolation and switch between trusted and untrusted commodity oses. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, 2012.

[92] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 279–292, Berkeley, CA, USA, 2006. USENIX Association.

[93] P. Technologies. Trustedcore: Foundation for secure CRTM and BIOS implementation. https://forms.phoenix.com/whitepaperdownload-/docs/trustedcore_wp.pdf, 2006.

[94] The Blue Pill. http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

[95] L. Torrey, J. Coleman, and B. Miller. A comparison of interactivity in the linux 2.6 scheduler and an mlfq scheduler. *Software: Practice and Experience*, 37(4):347–364, 2007.

[96] Trusted Computing Group. TPM main specification. Main Specification Version 1.2 rev. 85, Feb. 2005.

[97] A. Vasudevan, B. Parno, N. Qu, V. Gligor, and A. Perrig. Lockdown: A safe and practical environment for security applications (cmu-cylab-09-011). 2009.

[98] J. Wang, A. Stavrou, and A. Ghosh. Hypercheck: a hardware-assisted integrity monitor. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 158–177, Berlin, Heidelberg, 2010. Springer-Verlag.

[99] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[100] Z. Wang, X. Jiang, W. Cui, and X. Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 21–38, Berlin, Heidelberg, 2008. Springer-Verlag.

[101] M. D. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[102] A. M. White, A. R. Matthews, K. Z. Snow, and F. Monrose. Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 3–18, Washington, DC, USA, 2011. IEEE Computer Society.

[103] WikiPedia. Souce line of code. http://en.wikipedia.org/wiki/Source_lines_of_code.

[104] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized i/o devices. In *Proceedings of USENIX Annual Technical Conference*, 2008.

[105] X. Xiong, T. P. State, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity os from untrusted extensions. *NDSS*, 2011.

[106] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[107] Z. E. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Trans. Inf. Syst. Secur.*, 8(2):153–186, 2005.

[108] M. Zaharia, S. Katti, C. Grier, V. Paxson, S. Shenker, I. Stoica, and D. Song. Hypervisors as a foothold for personal computer security: An agenda for the research community. Technical report, Jan 2012.

[109] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2012.