8-2018

Secure enforcement of isolation policy on multicore platforms with virtualization techniques

Siqi ZHAO
Singapore Management University, siqi.zhao.2013@phdis.smu.edu.sg

Secure Enforcement of Isolation Policy on Multicore Platforms with Virtualization Techniques

SIQI ZHAO

SINGAPORE MANAGEMENT UNIVERSITY
2018

Secure Enforcement of Isolation Policy on Multicore Platforms with Virtualization Techniques

by

Siqi Zhao

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

Dissertation Committee:

Xuhua DING (Supervisor / Chair)
Associate Professor of Information Systems
Singapore Management University

Debin GAO (Co-supervisor)
Associate Professor of Information Systems
Singapore Management University

Robert DENG Huijie
Professor of Information Systems
Singapore Management University

Jianying ZHOU
Professor of Cyber Security
Singapore University of Technology and Design

Singapore Management University
2018

Secure Enforcement of Isolation Policy on Multicore Platforms with Virtualization Techniques

Siqi Zhao

Abstract

A number of virtualization based systems have been proposed in the literature as an effective measure against the adversaries with the kernel privilege. However, under a systematic analysis, such systems exhibit vulnerabilities that can still be exploited by such an attacker with the kernel privilege. The fundamental reason is that there is an inherent incompatibility between the tamper-proof requirement and the complete mediation requirement of the reference monitor model. The incompatibility manifests in the virtualization based systems in the form of a discrepancy between the enforcement capability demanded by the high-level policy and the one achievable through the system design approach mandated by the low-level hardware enforcement mechanism.

The scenario is further complicated by an implicit assumption in existing works, which is that the underlying platform is single-threaded. This assumption is becoming increasingly distant from the real-world computing landscape where multi-core machines have become ubiquitous. With the broken assumption, the adversarial threads running on other cores gain capabilities that are not possible on uni-core platforms and are possible to launch new attacks.

In this work, the existing systems are firstly examined in a systematic manner. The consequences and implications of the aforementioned discrepancy are shown by dissecting and examining existing systems' high-level security goals and design details of leveraging the hardware enforcement mechanism. Meanwhile, the issues caused by concurrent execution on multicore platforms are presented. Two concrete attacks are shown as the examples of the complications of the multicore scenario.

In light of the issues, Fully Isolated Micro-Computing Environment (FIMCE)

is proposed. FIMCE addresses the issues revealed in the analysis by managing involvement of semantics from the kernel. It encloses a complete set of resources needed by a program. FIMCE also features great flexibility and can be tailored to various applications. Built on top of this environment, Immersive Execution Environment (ImEE) is presented. ImEE is designed for efficient introspection through consistent address space mappings. In the ImEE's design, an isolated environment is equipped with tweaked address mappings. It directly reuses the page tables of a target VM and synchronizes the root of the page tables with the target VM. As a result, the target VM cannot present fake address mappings to the introspection tool to mislead the results.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor and committee chair Xuhua Ding for his guidance over the course of the program, for the intellectual challenges posed during the discussion sessions and for the rigorous scrutinization over the entire course of the work. These glimpses of the power of reasoning, and what it might achieve, are the greatest lessons I have learned so far. I am also grateful to the other dissertation committee members, Debin Gao, Robert Deng and Jianying Zhou, for their effort and time on evaluation and guidance during the dissertation defense process.

List of Publications

Conference Papers

S. Zhao and X. Ding. On the Effectiveness of Virtualization Based Memory Isolation on Multicore Platforms. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, 546-560.

S. Zhao, X. Ding, W. Xu and D. Gu. Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed. In *26th USENIX Security Symposium (USENIX Security 17)*, 799-813

Journal Paper

S. Zhao and X Ding. FIMCE: A Fully Isolated Micro-Computing Environment for Multicore Systems. *ACM Transactions on Privacy and Security (TOPS) 21 (3), Article 15 (May 2018)*

Chapter 1

Introduction

1.1 Overview

The Operating System typically acts as the reference monitor [15] that enforces the system-wide security policy on a computer system. For such a purpose, it is granted with the highest privilege. However, complexity in modern OS weakens its own integrity guarantees and leads to frequent security breaches. Once breached, the attackers obtain the same privilege as the kernel, therefore, it is possible for them to launch arbitrary software attacks and compromise the entire system. Such attackers pose serious threats.

The virtualization based approach was proposed to defend against this kind of attackers because it leverages a higher level of privilege than the kernel. A number of such systems have been proposed in the literature. However, these systems exhibit certain issues, drawbacks and even vulnerabilities upon close inspection. In this section, the severity of the adversaries with the kernel privilege is firstly described as the motivation for the rest of the work. A brief summary of the virtualization systems is given next, so as to lay down the context for the research questions pursued in this work.

1.1.1 Adversaries with Kernel Privilege

The Operating System is the entity that manages the resources in a traditional computer system. Conveniently, it also serves as the reference monitor that monitors the accesses to the resources by individual subjects, and enforces any security policies. The reference monitor concept mandates three requirements to be satisfied, namely, tamper-proof, complete mediation and verifiability. In the traditional computer systems, the OS had long been regarded as trusted in the sense that it maintains its own integrity, i.e. it satisfies the requirement of tamper-proof.

However, just as any software, the OS also contains vulnerabilities that can be exploited. Such exploitation is serious since it can lead to violation of any security policies. Once inside the OS, the attackers gain the privilege of the kernel and can manipulate the system-wide policy enforcement. Therefore, arbitrary attacks can be launched. An example class of attack is the so-called *rootkits*. The rootkits are malicious code produced by the attackers and injected into the kernel space, such as the `adore-ng` rootkit [1] which modifies the kernel function pointers. In this case, because the system-wide policy enforcement has been manipulated by the attackers, even detection of these attacks is challenging.

The problem of the adversaries with the kernel privilege is relevant because of two factors. The first factor is that the interface to modern OSes is becoming increasingly complex. Such a complex interface exposes an enormous attack surface. The enormity is exhibited from two aspects. First, the system call interface is known to be large and complex. For example, Linux contains over three hundred system calls and there are more on Windows. Many system calls also contain complex semantics such as `ioctl` whose behavior completely depends on the device that it is interacting with. Second, besides the system calls, the applications can also interact with the kernel via implicit interfaces such as the file systems, e.g. `/proc` and `/sys`. These interfaces also do not have clearly defined semantics. Therefore, it is hard to reason about the behaviors of the applications and enforce effective poli-

cies. In existing works, the complex interface frequently introduces challenges to security designs that interpose on the system call interface such as Janus [45].

The second factor is that a modern OS consists of an enormous amount of code. For example, the Linux kernel reached one million lines of code at version 2.1.63 [5]. Since then, the size of the kernel has undergone a steady increase. Version 4.15.9 has accumulated 20 million lines of code. The large amount of code results in complex semantics that is reachable from the large attack surface. Therefore, the likelihood of vulnerability is high and the modern OSes are routinely broken by attackers.

In practice, the threat from a kernel level adversary is present in many scenarios. One such scenario is when the kernel is owned by a party not necessarily trusted, e.g. the manufacturers. In this case, there is no guarantee that the kernel will enforce any security policy, including the basic isolation policy. In other scenarios, the kernel may be compromised in various ways besides code injection. For example, data-only attacks can leave the kernel data in an invalid state, leading to security policy violations. As shown by the attacks that modify the identity in the `cred` structures, it is possible to obtain root privilege.

Without modifications on the system architecture, certain integrity measurement schemes alleviate this problem by measuring integrity at certain points during the execution. For example, TPM-based integrity measurement [90] can be leveraged to construct a chain of authenticated component loaded. However, this approach has a great limitation because it only guarantees integrity during the load time. The malicious behavior could be injected to the kernel after the measurement had finished. An example is the Trusted Boot [11] which is a boot loader that utilizes the TPM to perform measurement of the loaded OS image. However, integrity of the kernel at runtime is not protected. Furthermore, integrity measurement is usually made over code and static data, however, attacks can also be launched via modification on the dynamic data. An example is the aforementioned `adore-ng` rootkit which modifies dynamic function pointers in the file system layer. Integrity measurement does

not cover the dynamic data, therefore, it cannot prevent this kind of attacks. Other integrity measurement schemes, such as PRIMA [54] and Linux Integrity Measurement Architecture [79], measure applications. Since they assume a trusted kernel, they cannot be applied, neither.

1.1.2 Virtualization-based Systems

The virtualization based approach is an effective means to defend against the adversaries with kernel privilege. This approach is architectural in that it de-privileges the kernel so that it is no longer the entity with the highest privilege. In such systems, the OS is de-privileged to run in a *domain* or virtual machine (VM), which is managed and regulated by a higher-level entity usually called the virtual machine monitor (VMM) or the hypervisor. By the nature of the attacker with the kernel privilege, only the OS is vulnerable; the hypervisor is not affected. Therefore, policy enforcement by the hypervisor is not affected by such powerful attackers.

From a system perspective, the virtualization based systems are reference monitors implemented by hardware virtualization mechanism. Hardware virtualization mechanism allows the reference monitor to interpose on the hardware-software interface through which traditionally the kernel directly interacts with the hardware. The interposition allows the hypervisor to distinguish individual accesses to hardware resources, thereby provides an opportunity to enforce security policies.

In its basic form, such systems enforce a type of policy called domain isolation. In domain isolation, the hypervisor distinguishes accesses to hardware resources by domains. The untrusted OS runs in one of the domains while the trusted OS and applications run in others. Domain isolation guarantees integrity and confidentiality of the trusted domain, which enables the establishment of further security properties of the system. Examples include Terra [46], Lares [73] and HookSafe [96]. The main drawback of this design, however, is that the TCB includes the trusted OS, which is still large.

Further systems push the enforcement granularity level towards the fine-grained direction. They distinguish the accesses by more fine-grained entities such as processes or even code segments. This approach eliminates the guest OS from the TCB, because the logic in the kernel that manages the entities other than the ones concerned is not needed. However, since hardware virtualization mechanism is not aware of the concerned entities which are usually defined by the kernel, policy enforcement by the virtualization based systems needs to rely on the facilities of the untrusted OS that define such concerned entities, such as dynamic memory allocation, file systems and the kernel's page table. The consequence of the reliance is that certain semantics from the untrusted kernel is involved in the policy enforcement process, which creates issues. The issues of the involvement are elaborated in Section 1.3. In the rest of the work, this approach is called the memory isolation because the fine-grained entities are usually memory segments.

A number of existing works have utilized the hardware virtualization mechanism for various security purposes. For the rootkit problem, for example, SecVisor [80] ensures that only whitelisted code can execute with kernel privilege. Lares [73] protects the integrity of hooks placed in the kernel. HookSafe [96] proposes a lightweight approach to protect kernel function pointers by aggregating them.

The virtualization mechanism is also versatile so that it can be used for many other security objectives when facing a malicious kernel, such as isolated execution [46, 68, 51, 26, 29, 93, 86], memory compartmentalization [67], integrity measurement [17], stealthy debugging [41, 37], enforcing execute-only permission [32, 97] and virtual machine introspection [49, 24, 39, 42, 56]. Chapter 2 provides a list of description of these systems.

1.1.3 Issues and Research Objectives

Effectiveness of Policy Enforcement The virtualization based systems are reference monitors implemented by hardware virtualization mechanism. Since hardware

virtualization mechanism mandates a special form of policy, the systems all need to translate the elements of the high-level security policy into the special form, which is expressed with the abstractions both within the virtualization context and observable at the hardware-software interface, e.g. memory pages and domains. For example, TrustVisor enforces that only a sensitive module can access its own code and data. It needs to translate "the sensitive module" into "a registered set of pages".

The main issue is that the translation process involves semantics from the untrusted kernel, because the high-level entities are defined and managed by the untrusted kernel. However, most systems do not explicitly describe this translation process. The extend to which such semantics is involved is not clear, neither the consequences.

Because of involvement of the semantics from the kernel, effectiveness of policy enforcement can be affected by the subtlety in the translation process. The subject and object identity are vulnerable to manipulation and can be inconsistent. For example, when processes are identified by `CR3` values, the processes can be impersonated since the values are managed by the untrusted kernel. Also, the mediation of the operations by the hypervisor can be incomplete. Therefore, it remains an important research question: *how effective can a system based on virtualization techniques enforce the high-level policies?*

Enforcing Isolation In order to enforce the high-level policies, the fundamental guarantee the hypervisor needs to provide is isolation. Isolation concerns about the identity of the entities in the high-level policies. It refers to the requirement that there is no overlap between the low-level representations for any two high-level policy entities. For example, no "sensitive modules" share a memory page. Here, the issues of existing systems are two fold. First, as mentioned above, isolation is enforced with semantics from the untrusted kernel. Second, when the first issue is compounded with concurrent execution, the enforcement become ineffective.

In virtualization based systems, the specific translation that the hypervisor needs to perform is from the high-level entities to the low-level domains defined by the

extra paging structures. Since the kernel firstly defines the high-level entities, the kernel semantics which is the identities information recorded in the kernel page tables is needed to craft the low-level domains. During policy enforcement, because the observed accesses through the hardware-software interface are influenced by the address translation configurations, the kernel can still influence the policy enforcement by manipulating the kernel semantics. Many systems noticed this issue and performed checks on the kernel's page table on various occasions. However, such checks are ad-hoc and not systematic.

Furthermore, the fact that modern hardware platforms are multi-core complicates the situation. Parallel execution on multi-core platforms allows the untrusted domain and the trusted domain to run at the same time, which cannot occur on single-threaded platforms assumed by previous works. The untrusted domain thus can take advantage of any resources available to attack the trusted domain.

Therefore, whether existing approaches still guarantee isolation on a multi-core platform requires a closer investigation. Due to the importance of isolation, we pose the second research question. *How to ensure secure isolation by virtualization techniques on modern multi-core hardware systems?*

Consistent Introspection An isolated trusted domain can serve as a suitable standpoint for extending the trust beyond. One possibility is that it can be used to scan the memory content inside other domains; a technique called Virtual Machine Introspection (VMI). VMI is a prerequisite of many analyses that aim to detect malicious activities, because it provides the raw data for the detection tools to analyze.

The VMI systems face a special form of the policy translation issue mentioned above. Although they do not necessarily enforce any policy over the other domains, they do need to translate the high-level objects typically in memory to raw bytes observed by the hypervisor. More precisely, the VMI systems need to translate the virtual addresses of the objects into the physical addresses so that it can read the content out. Inevitably, the kernel's page tables in the untrusted domain are involved.

The previous VMI tools rely on software logic to walk the kernel's page tables during the translation process. However, the kernel can modify its own page tables in arbitrary ways, which affects the memory layout of the untrusted domain. As a consequence, the VMI tools do not have a consistent view of the virtual address space. Any further analysis, therefore, leads to invalid results. This method is also slow because the page table walks require a series of loads from the memory. Therefore, the agility of introspection cannot match with the untrusted domain's operations on the address mappings. In the face of these issues, the last question is thus: *how to ensure consistency in memory introspection?*

1.2 Threat Model

The adversary in consideration is one with the kernel privilege. For example, the adversary may be the kernel infected by the rootkits. However, the means by which the adversary obtains this privilege is irrelevant. With the kernel privilege, the adversary can launch arbitrary software-based attacks, such as arbitrary memory accesses and execution context manipulations. He is also capable of manipulating external devices that he can access for his purposes. Meanwhile, it is assumed that the BIOS, the firmware and the hardware components in the platform are not compromised by the adversary and behave in compliance with the respective specifications. The hypervisor's code, data and control flow are trusted throughout the lifetime of the platform. Side channel attacks are not considered, nor are denial-of-service attacks.

1.3 Security Policy Enforcement

As stated in Section 1.1.2, the virtualization based systems interpose on the hardware-software interface in order to enforce a set of high-level security policies. Following the reference monitor concept [15], they need to satisfy three basic requirements, namely, tamper-proof, complete mediation and verifiability. Among

these, the tamper-proof requirement is satisfied due to the higher privilege. Meanwhile, the requirement of verifiability typically requires formal verification of the logic which is a separate field of study, therefore, it is out of the scope of this work.

The focus here is the interaction between the tamper-proof requirement and the complete mediation requirement. Ideally, both requirements should be met to guarantee effectiveness of policy enforcement. The tamper-proof requirement is met by adopting the virtualization mechanism; the higher privilege ensures integrity of the hypervisor. However, this design approach introduces inevitable involvement of semantics from the untrusted kernel. The involvement, if not treated with care, can cause failure in meeting the complete mediation requirement.

Involvement of semantics from the untrusted kernel is inevitable due to the point of interposition in the virtualization based systems. At the hardware-software interface, only hardware related states are available, such as the physical addresses of the memory accesses, the current value of the program counter or the root of the current page tables. As a result, there exists a wide gap between this information and the one needed to make useful inference at the level of the high-level policy, leading to a discrepancy between the capability needed to enforce the high-level policy and the one achievable by the naive use of virtualization mechanism. The gap is thereafter referred to as the *inference gap*. Bridging the inference gap requires semantics from the untrusted kernel because the high-level entities are defined by the untrusted kernel. Therefore, the specific strategy adopted by individual systems to leverage such semantics to close the inference gap provides key insights into effectiveness of policy enforcement. Chapter 3 examines involvement of such semantics in the gap-bridging attempts.

Also addressed in Chapter 3 are the issues with respect to concurrency. Possible issues of existing schemes are discussed. In addition, two concrete attacks are presented. The stifling attack allows the guest kernel to preserve stale access permissions for certain memory pages and prevent the hypervisor from revoking the permissions. The VPID attack exploits a performance optimization feature also to

keep stale permissions. Chapter 3 provides detailed descriptions of these attacks.

1.4 Enforcing Isolation Policy on Multicore Platforms

The aforementioned inference gap results in an incomplete isolation boundary when enforcing the isolation policies on multi-core platforms. The reason is that kernel semantics in the guest page tables is involved to translate the subjects and objects, while the guest page tables are subject to manipulation by the guest kernel on a multi-core platform. Therefore, an approach called *full isolation* which specifically takes multi-core platforms into consideration is proposed.

The full isolation approach controls involvement of kernel semantics in the enforcement process of the isolation policy. It applies this principle in its isolation of the memory of the sensitive program, of a physical processor core and of any needed I/O devices. The isolated core, memory and devices form an isolated execution environment which is termed *Fully Isolated Micro-Computing Environment* (FIMCE). Similar to existing works, the FIMCE leverages on the hardware virtualization extension and uses the hypervisor's control on the system resources to achieve isolation.

Due to the full isolation approach, the isolation boundary is clearly defined. The hardware resources are cleanly divided between the trusted and the untrusted, including the core, the memory and any external devices. Because of the complete control over involvement of semantics from the kernel, the inference gap is eliminated. Plus, the concurrency issues are also eliminated by full isolation. The overall benefit is that the malicious guest kernel cannot interfere with management of the identities, neither can it disrupt the execution of the isolated environment.

Another advantage of FIMCE is that the configuration is rather nimble. The hypervisor is free to tweak the hardware configuration of the FIMCE for innovative

use cases. Meanwhile, FIMCE provides a suite of useful libraries for the isolated application, so that the application does not need to be self-contained. The libraries are loaded on demand to minimize the amount of code inside the environment, reducing the risk. To relieve the hypervisor from managing any file system, the library loading process is delegated to the kernel. Thus, the hypervisor's logic is simplified, minimizing its code size. Although the kernel handles library loading, the integrity of the libraries is verified after they are loaded. The linking between the libraries and the application is also verified.

The recently emerged Software Guard Extension (SGX) to the x86 architecture provides hardware based isolation to the applications. Although SGX offers strong memory isolation guarantees, compared to the FIMCE it lacks full isolation and I/O capabilities. A comparison between SGX and the FIMCE and a discussion about possible ways to integrate both for stronger security guarantees are presented.

1.5 Consistent Virtual Machine Introspection

The fully isolated environment can be applied for consistent and efficient Virtual Machine Introspection (VMI). As introduced in Section 1.1.3, in a traditional out-of-VM VMI system, the introspection program faces a special form of the inference gap. The gap is between the memory view at the level of the concerned data objects in the target VM and the low-level memory view provided by virtualization primitives. In a traditional VMI system, the gap is bridged by the replication of the MMU logic of the target VM and the guest page tables. The replication is necessary because the VMI tools need to translate the virtual addresses used in the target VM to the physical addresses. However, the replication causes issues. It is not only slow due to emulation of the hardware logic, but also not consistent due to replicated page table data. It is thus possible for a malicious target VM to present a fake memory view to the introspection tool, while use another for its own execution.

The *Immersive Execution Environment* (ImEE) addresses these issues. The

ImEE is a special execution environment based on the FIMCE with tweaked address space mappings. The ImEE directly reuses the address space mappings of the target VM by setting the `CR3` register to be the same value as the one in the target VM. During execution of the ImEE, the hypervisor also ensures that the `CR3` value is always synchronized with the value in the target VM. Therefore, this design ensures that the target VM cannot manipulate any address translations to hide memory content, neither can it stealthily switch to another set of mappings.

Besides a consistent memory view, utilization of hardware virtualization mechanism provides the ImEE a performance advantage, too. In the evaluations, the ImEE shows a remarkable speedup compared to existing software-based tools. Serving as a memory access engine, the ImEE can be integrated with existing VMI tools, providing immediate benefits.

1.6 Background

The following two pieces of background knowledge are necessary for the understanding of the rest of the content. First, the address translation process in a virtualized platform is necessary, which is self-evident. The first subsection provides a brief description of the process. For the full details, the readers are referred to the Intel's Software Developer Manual. The second subsection describes memory access in a symmetric multiprocessing (SMP) system. This is necessary for the understanding of the multi-core complications.

1.6.1 Address Translation in Virtualization

With hardware-assisted memory virtualization, address translation is divided into two stages for any memory access inside the guest. In the first stage, the MMU translates a virtual address (VA) into a Guest Physical Address (GPA) by walking the guest page tables managed by the kernel. In the second stage, the MMU translates a GPA to a Host Physical Address (HPA) by traversing the EPTs managed by

the hypervisor. The roots of the guest page tables and the EPTs are stored in the `CR3` register and a control structure field called `EPT Pointer`, respectively. During address translation, the MMU raises an exception if the type of a memory access conflicts with the permitted types specified in these page tables. There are multiple sets of the page tables in either stages at runtime. Only one set in each stage is active at any time. However, the MMU does not switch the sets automatically, instead, the system software is designated with such a responsibility. The kernel is responsible for switching the `CR3` for any reason that it deems appropriate, and the hypervisor is responsible for the `EPT Pointer` switching.

To reduce latency of address translation, Translation Lookaside Buffers (TLBs) in each CPU core cache recently used translations and access permissions. The MMU traverses the page tables only when the TLBs do not store a matching entry. However, unlike data and instruction caches, it is the software, instead of the hardware, that maintains consistency between the TLBs and the page tables. The operating system and the hypervisor are expected to invalidate the relevant TLB entries after updating the page tables except a few special occasions such as reloading the `CR3` register.

The TLB entries can be tagged to improve efficiency. In case of a context switch, all TLB entries need to be flushed because the cached entries may not match the address mapping in the new address space. When context switch happens frequently, such as when there are frequent hypercalls, the MMU needs to walk all the page tables again after each switch, which can incur performance penalties. With hardware virtualization, the number of the page tables to walk can be sixteen[1], which means sixteen consecutive loads from memory. To avoid such performance penalties, the TLB entries are tagged with certain address space identifiers such as the *Virtual Processor ID* on later x86 architecture. With a tagged TLB, a TLB entry is considered a hit during translation only if its tag matches with the current address

[1]This assumes a 64-bit guest with 4KB pages, therefore, there are four levels of page table. To reach the guest page table in every next level, EPT needs to be walked because guest page tables store GPAs. Each EPT walk also traverses four levels of page tables, assuming 4KB pages.

space identifier.

1.6.2 Memory Access in SMP Systems

In an SMP setting, multiple cores access the shared physical memory independently as shown in Figure 1.1. They may use different address mappings since address translation is performed independently by each core's MMU. The VA-to-GPA mappings on all cores are controlled by the guest kernel. Depending on the running threads, the cores may or may not use the same set of page tables. The GPA-to-HPA mappings are controlled by the hypervisor. In the most common design, *all* cores of the same VM use the same set of EPTs since the GPA-to-HPA mappings are for the entire virtual machine. In other words, the guest kernel's scheduling, i.e. the guest page table usage, is transparent to the hypervisor.

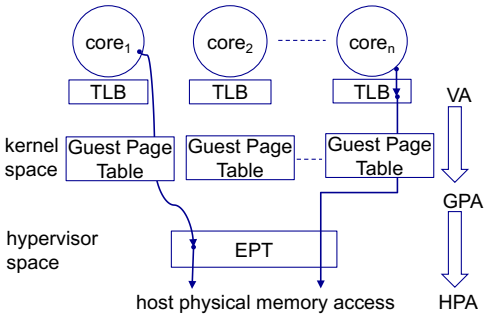


Figure 1.1: The paradigm of memory access in an SMP setting. The first core has TLB misses and accesses the memory via the guest page tables and the EPTs, while the last core has TLB hits and accesses the memory without consulting any page table.

In the SMP setting, it is more complicated to maintain TLB consistency since the threads on any core may modify the shared page tables while the TLBs are local to each core. Typically, the thread that modifies the paging structures initiates a sequence of operation called *TLB shootdown* whereby the thread on the initiating core fires an Interprocessor Interrupt (IPI) to other cores. Upon the arrival of an IPI, a handler is invoked to invalidate those stale TLB entries on the receiving core. On x86 platforms, the Advanced Programmable Interrupt Controller (APIC) is responsible for receiving and sending IPIs. Its proper behavior affects the success of TLB

shootdown, consequently TLB consistency.

1.7 Organization

The rest of this document is organized as follows: Chapter 2 reviews closely related literature. Chapter 3 investigates the issues encountered by existing virtualization based systems, including two attacks on the multi-core platforms. Chapter 4 presents the design of FIMCE. Chapter 5 shows an application in the context of VMI, i.e. the ImEE system and its details. Finally, Chapter 6 concludes this document.

Chapter 2

Related Work

Virtualization techniques were shown to have application in security systems by early works such as Terra [46] and Proxos [88]. These works utilize the virtualization architecture in its crude form that one VM is assumed to be trusted while the others are not. The security-critical software is subsequently run in the trusted VM. Due to the isolation between VMs, the other untrusted part cannot tamper with the code inside the trusted VM. Further works all are based on the isolation between domains defined by the hypervisor, while they are geared towards different security purposes.

2.1 Trusted Execution Environment

The goal of establishing a TEE is to protect confidentiality and integrity of a sensitive task's data and execution against attacks from the untrusted kernel. With the blessing of the EPT, the hypervisor isolates a sensitive task's code and data regions so that the external software (including the kernel) does not have *any* access to them. The rest of the section groups the literature of isolation-based TEE based on the protected task's scope.

Code Segment Isolation TrustVisor [68] proposes to protect a few memory pages called a piece of application logic (PAL) and allows attestation. TrustPath

[104] and MiniBox [62] both build on top of TrustVisor. TrustPath builds a trusted I/O path between a device and an application end point. MiniBox realizes a two-way sandbox by running a Native Client [102] instance as a PAL. XMHF [93] aims for providing a framework for building hypervisor based solutions and is formally verified. Later version of TrustVisor is re-implemented on top of XMHF. XMHF serves as the foundation for two systems that performs I/O isolation [103, 105]. Driverguard [28] is another work on I/O isolation. CAFE [57] also built on XMHF to provide runtime isolation of secret binaries. As a separate line of research, SeCage [67] combines software analysis and hypervisor protection to automatically extract compartments that are protected by the hypervisor at runtime. Fides [86] also protects modules that consist of a few pages. However, the modules are not completely isolated, the code portion is readable while the secret portion is inaccessible. On ARM platform, OSP [30] combines the virtualization extension and TrustZone to offer flexible on-demand protection of a piece of code.

Isolation of a code segment using a separate address space has been demonstrated in previous works such as Gateway [85]. Although the secure driver code is isolated from the guest kernel using shadow page table, it can also be achieved by using EPT.

Application Isolation InkTag [51] isolates the pages of an application and only allows the guest OS to access the ciphertext. A feature of InkTag is that it allows application to express intention to modify the address space so that the hypervisor later can verify the changes performed by the guest OS, a mechanism called *paraverification*. Based on InkTag, Sego [59] improves on verification of the system services. On top of the isolated execution, AppSec [75] provides a trusted human interface for the isolated application. AppShield [29] does not allow guest OS to access the isolated pages, requiring extensive wrapping of system call interfaces. The idea of isolating an entire application has been demonstrated earlier by Overshadow [26] and SP^3 [101], although they are not implemented using the EPT because it was not available.

The difference in the scale of the protected memory results in different design approaches. The systems that protect the entire address space typically need to consider dynamic behavior of the region, i.e. newly mapped or unmapped regions. Therefore, they need to track the changes on the address space. Both Inktag [51] and AppShield [29] lock the page tables of the guest in order to perform such tracking.

The systems that protect only a few pages usually do not consider page swapping by guest kernel. The protected pages are typically allocated by the kernel and are assumed to be always present in memory.

2.2 Kernel Integrity

SecVisor [80] and HUKO [99] remove certain access according to the current code occupying the CPU, in order to protect kernel integrity. While they both try to protect the entire kernel space memory, SecVisor focuses on protecting the kernel space as a whole from user space applications and untrusted extensions. HUKO enforces a fine-grained access control model among kernel space subjects such as kernel module and kernel code.

SecVisor [80] protects the integrity of the kernel code and uses a whitelist to allow any listed kernel extension to be also integrity-protected. On the other hand, HUKO [99] does consider dynamic kernel modules. For this purpose, it needs kernel level semantics so it inserts a module into the protected kernel space in order to track newly added or removed kernel space memory.

As a prerequisite to kernel integrity, SecVisor identifies the kernel by intercepting all traps to kernel. To further harden the kernel, it ensures that the kernel always starts to execute at predefined entry points.

Previous systems that ensure part or the whole of the kernel's integrity before the EPT include HIMA [17], NICKLE [76], HookSafe [96] and Lares [73]. HIMA marks only measured memroy pages to be executable during runtime. NICKLE separates the execute and other types of access to different copies of the kernel code

using page table mappings. HookSafe ensures that the gathered kernel hooks are protected. Lares also protects the hooks placed inside the guest using page table permission bits. All these goals can also be achieved using the EPT.

2.3 Mapping Redirection

Heisenbyte [89] separates read access and instruction fetch to a code page to separate copies prepared for each type of access. The separation is applied to any code uniformly. The EPT is used to distinguish the memory accesses. This type of systems does not consider the kernel as malicious. However, when these systems are deployed to practical use, the consequences of a compromised kernel should be considered. Other measure, such as kernel integrity should be coupled with these systems to ensure the security of the system as a whole.

2.4 Event Trap

The EPT can be utilized as merely a means to intercept certain access to memory. After interception, the access is allowed to continue as per normal. Graffiti [33] tracks the rate of allocation in the address space of an application by tracking modifications on the page table. It uses the EPT to intercept writes to the page table. SPIDER [37] and HyperDBG [41] use the EPT to detect execution of code and data access to memory to implement stealth debugging facilities.

Before the EPT became available, trapping events in the guest had been achieve via using shadow page tables. As demonstrated in Patagonix [65], the execution of any binary code is detected by marking memory pages non-executable. The EPT can also be used for this purpose.

2.5 Auxiliary Uses

The EPT can also be used to enforce access permissions that are not supported by guest page table. For example, execute-only permission is *not* possible with guest page table. Because once the P-bit is set, both read and execute permission are granted, however, there is only the NX-bit that removes the execute permission. No such bit that removes the read permission exists. Readactor [32] and NEAR [97] leverage the EPT to enforce execute-only memory on the code pages of a protected application to mitigate memory disclosure.

2.6 Virtual Machine Introspection

The fundamental problem of VMI is to acquire the kernel's semantic by reconstructing the kernel objects. Significant efforts have been spent on directly recovering the kernel's data structures from the raw bytes. It can be based on expert knowledge (e.g., Memparser [22], GREPEXEC [23], Draugr [38], and others [4, 7, 8, 9, 12, 13, 47, 74]) and automatic tools (e.g, SigGraph [63], KOP [25], and MAS [35]). These studies usually involve a large amount of engineering work and are useful for memory forensic analysis. Since they do not emphasize on live memory introspection, the security and effectiveness of accessing the guest's live state are not their main concerns. In general, they are orthogonal to our study.

A more sophisticated approach is to reuse the existing kernel to interpret and construct the desired kernel objects from the memory of a live guest. Based on whether the introspection uses the guest VM's kernel or not, schemes using this approach can be further divided into in-VM introspection and out-of-VM introspection.

2.6.1 In-VM Introspection

In general, in-VM introspection schemes aim to save the engineering efforts by relying on the guest kernel's capabilities. Process Implanting [49] loads a VMI program such as *strace* and *ltrace* into the guest VM and executes it under the camouflage of an existing process. SYRINGE [24] runs the VMI application in the monitor VM and allows the introspection code to call the guest kernel functions under a guest thread's context. When the guest kernel is not trusted, the security and effectiveness are totally broken, because it is straightforward for a rootkit to evade or tamper with the introspection. Hence, these in-VM introspection schemes are only useful to monitor the user space behavior in the guest VM. SIM [81] is an in-VM monitoring scheme against rootkits. To run the monitoring code inside the untrusted guest, it creates a SIM virtual address space isolated from the guest kernel. Hooks are placed in the guest to intercept events. The address switches between the kernel and the SIM code is guarded by dedicated gates.

2.6.2 Out-of-VM Introspection

The out-of-VM introspection code stays outside of the target guest. Therefore, it is capable of introspecting the guest VM to detect kernel-level malicious activities without directly facing the attack. Virtuoso [39] generates the introspection code by training the monitor application in a trusted VM and reliably extracting the introspection related instructions from the application. The execution trace is replayed in a trusted VM when performing introspection, during which data accesses are redirected to the guest VM's memory. VMST [42] is another out-of-VM introspection technique. It manages to reuse the kernel code by running the introspection application in a monitor VM emulated by QEMU [21]. A taint analysis runs in the monitor VM and relevant data accesses are redirected to the guest's live memory. Hybridbridge [78] is a hybrid approach which combines the strengths of both VMST and Virtuoso. Similarly, the VMI application is running in the trusted monitor VM and

the OS code is reused. The kernel data accesses which are related to the monitoring functionality are identified and redirected to the guest kernel memory when needed. EXTERIOR [44] is another space traveling approach inspired by VMST, which supports not only guest VM introspection but also reconfiguration and recovery of the guest VM.

Process Out-Grafting [84] relocates the monitored process from the guest VM to the monitor VM. The monitor VM always forwards system calls to the guest. The guest kernel handles it and returns the results to the monitored process. This approach requires the implicit assumption that the guest kernel is trusted. VMwatcher [56] is a system that performs rootkit detection from outside of a VM by manually reconstructing the semantic view.

TxIntro [66] is an out-of-VM and non-blocking approach designed for timely introspection. It mainly focuses on retrofitting the hardware transactional memory to avoid reading inconsistent kernel states. In its design, the VMI code runs on an implanted core and can also access the guest memory at a native speed. Nevertheless, it lacks sufficient security concerns and does not provide the introspection code a consistent memory view to that of the guest's.

2.7 Isolation With Other Techniques

Flicker [69] makes use of trusted computing techniques to set up a secure execution environment at runtime. It explores AMD's late launch technology which incorporates the TPM-based DRTM. The late launch technique sets up a secure and measured environment to protect a piece of code and data. The drawback is its high latency due to the slow speed of the TPM chip. Moreover, the protected code cannot interact with the rest of the platform.

The recently announced Intel Software Guard Extensions (SGX) [53] offers a set of instructions for an application to set up an *enclave* to protect its sensitive code and data. The hardware isolates the memory region and ensures that data in

the region can only be accessed by the code within. All other accesses are rejected by the hardware. Nonetheless, it is not able to support secure I/O operations, e.g., taking a password input from the keyboard.

As shown in TZ-RKP [18], a security monitor that resides in the secure world established by ARM TrustZone can protect the OS kernel in the normal world at runtime. Virtual Ghost [34] uses a language-level virtual machine to prevent an untrusted OS from accessing an application's sensitive memory regions. It requires compiler support and source code instrumentation on the kernel code in order to ensure control-flow integrity at runtime. PixelVault [92] creates an isolated execution environment on Graphics Processing Units (GPUs). Being an isolated device from the CPU with its own memory, GPU provides a natural ground for building an isolated execution environment. In the past, programming on GPU was difficult because of its highly specialized hardware. However, modern GPUs are becoming increasingly programmable so that executing code on GPU is easier. Nonetheless, this approach still requires significant development effort because there is little support from current systems. SICE [19] isolates a program that ranges from an instrumented application to a complete VM from the guest OS using System Management Mode (SMM). Compared to the micro-hypervisor approach, it features a smaller TCB since the TCB only consists of the hardware, BIOS and the SMM code. However, compared to virtualization, SMM is less standardized, which makes it hard to apply SICE's approach on certain platforms. For example, SICE's multiple processor support relies on hardware features only available on AMD processors.

Chapter 3

An Analysis of Effectiveness of the Existing Virtualization-based Schemes

As introduced in Chapter 1, to meet the tamper-proof requirement and the complete mediation requirement, virtualization based systems resort to the higher privilege level given by the hardware virtualization mechanism. Although this approach guarantees integrity thus satisfies tamper-proof, the systems are confronted with the inference gap which is caused by the system design approach dictated by the hardware virtualization mechanism. Interposing over the hardware-software interface, such systems are limited by the semantics available inside its trust boundary, i.e. the hypervisor space. The limitation makes involvement of semantics from the untrusted kernel inevitable. In order to enforce the high-level policy, these systems need to construct the view at the level of the high-level policy, however, they typically include kernel-defined entities. The construction occurs in a process called policy translation, during which the high-level policy is converted to a form understood by the hardware virtualization mechanism.

In this chapter, the policy translation process in various previous systems is reviewed. And the involvement of semantics from untrusted kernel in the gap-

bridging process is analyzed. The purpose is to show how the effectiveness of the enforcement is influenced. In doing so, firstly, a conceptual model that serves as the foundation of the rest of the discussion in this chapter is described. Next, the details of the policy enforcement process are examined. The virtualization based systems are referred to as enforcement systems hereafter, because their purpose is to enforce a set of high-level policy.

3.1 A Model of the Enforcement Systems

In this section, a model of the enforcement systems is described. The model provides a conceptual foundation for subsequent discussion of this chapter. It shows the policy formulation process in the enforcement system. It emphasizes the trust boundary and the involvement of semantics outside of the boundary during the policy formulation process.

The enforcement systems under consideration are designed to fulfill certain high-level security goal by exercising the reference monitor model. The high-level security goal is expressed in the form of a set of security policy, of which the enforcement realizes the security goal. For the actual enforcement, the enforcement systems rely on certain low-level mechanism, such as the MMU, assumed to be trusted. An illustration is shown in Figure 3.1. An example is a virtualization based system for the purpose of isolation, which enforces a set of isolation policies. The enforcement is achieved by configuring the MMU which is the low-level mechanism.

The enforcement system resides in an environment in which there are two privilege levels, low privilege level denoted P_L, and high privilege level denoted P_H. It is assumed that all entities in P_H are trusted, while all entities in P_L are not. In practice, there usually exists more than one privilege level in a contemporary computer system. All the privilege levels from low to high constitute an ordered set $L = L_1, L_2, ..., L_n$. Our discussion considers the partition of the set L at various

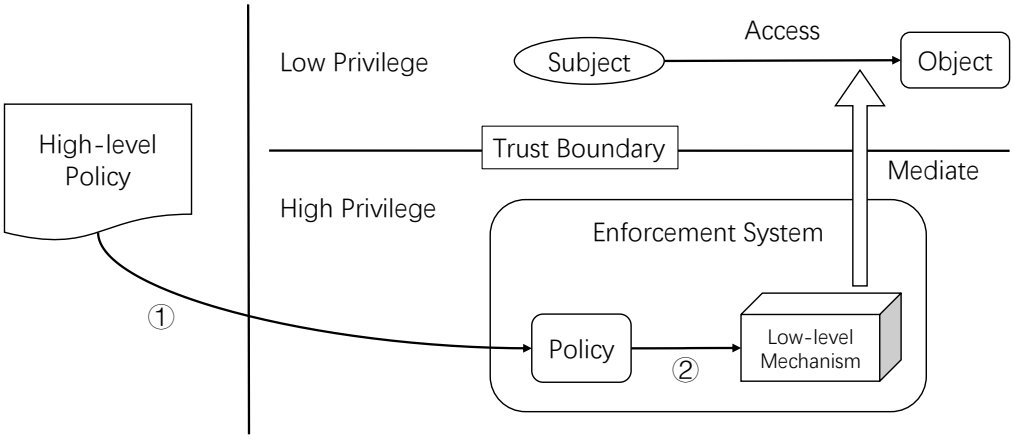


Figure 3.1: The Enforcement System Model

positions. All the privilege levels in the left set is denoted P_L and only the lowest privilege levels in the right set is denoted P_H. The boundary between P_L and P_H is call the *trust boundary*.

The model also shows the policy formulation process commonly observed in the enforcement systems. Note the fact that the high-level policy is usually described in natural language and involves terms that describe the user-facing entities of a computer system. For example, a policy may be concerned about files, programs or network connections. Furthermore, the fact that the enforcement systems are programs themselves implies that the system designer needs to carefully convert the policy to a form that can be understood by the systems, since programs do not understand the language in which the high-level policies are specified. This form understandable to programs is called *effective policy* thereafter. Lastly, the reliance on the lower-level mechanism usually dictates a second conversion from the effective policy to a form required by the low-level mechanism. This form is called *binary policy*. The two stages of conversion are also illustrated as step 1 and step 2 in Figure 3.1.

The point of mediation by the low-level mechanism on the accesses to objects depend on the nature of the mechanism itself. Typically, the mediation occurs on certain execution path that each access must undergo, so as to satisfy the complete mediation requirement for the low-level mechanism itself. From an architectural

point of view, the mediation usually concentrates at the interfaces between layers, because all accesses can be easily intercepted when they cross an interface. In the case of virtualization, the hypervisor interposes over the hardware-software interface which is traditionally used by the kernel to interact with the hardware. In past systems, the mediation typically happens at the system call interface such as the Janus [45] system. A common drawback of this type of mediation is that the information available at the interface is limited compared to that which is needed for certain high-level policy. For example, the hypervisors are only able to obtain information about physical pages, domains and program counters etc., all of which cannot be directly related to the high-level entities.

3.1.1 Conflict Between Tamper-Proof and Complete Mediation

There are two observations that can be made from the model. First, in order to satisfy the tamper-proof requirement, the enforcement system is isolated from the untrusted for its own integrity via the privilege separation mechanism. Second, the isolated states do not provide necessary semantics for decision-making at the level of the high-level policy.

Due to the limited scope, the enforcement system is obliged to associate the isolated states to the high-level entities. Note that the association usually involves semantic information, not necessarily data, beyond the trusted boundary. Such semantic information is defined by the software outside of the trust boundary. However, the enforcement system cannot directly execute untrusted code. Rather, it needs to replicate the semantics-defining logic inside its trust boundary and then acquire the needed semantics using the replicated logic. The consequence of the extra acquisition step is that, the semantics acquired by they enforcement systems is not always accurate. Such inaccurate semantic information relied upon by the enforcement systems presents an approximated view of the rest of the system, which affects the completeness of mediation.

In essence, the enforcement systems face a dilemma. For its own tamper-proof, only isolated states are used which provide limited semantics. However, for complete mediation, all semantics that are relevant should be used. It is, therefore, possible that the scope required by tamper-proof mismatch with the scope required by complete mediation. This situation leads to compromises in design that result in sacrifice in either requirement. For example, the Chrome browser adopts a process-based isolation design, and each browser tab is isolated using one process. The security monitor of Chrome checks the origins of the accesses made by the scripts on a web page to enforce the Same Origin Policy. To perform the checks, rich semantics of the web page and scripts is required. The security monitor resides in the same process as the scripts so that acquisition of the semantics is straightforward. However, this design gives up on the tamper-proof, leading to vulnerabilities that bypasses the checks [55].

On the other hand, many systems choose to ensure tamper-proof first. Consequently, they are limited in the scope of semantics and must approximate by acquiring the semantics outside of the trust boundary. In other words, they need to cross the inference gap. It is shown in the subsequent sections that when this approximation is inaccurate, the complete mediation requirement is compromised.

CAVEAT The discussion here is different from the semantic gap problem discussed in previous works such as VMST [42] and Virtuoso [39], although they appear to both be related to semantics. The semantic gap refers to the challenges in interpreting the meaning of a chunk of bytes. The problem here is about the consequence of the involvement of the semantics outside of the trust boundary in the policy enforcement process.

3.1.2 The Inference Gap

The inference gap refers to the gap between the semantics provided within the trust boundary and necessary semantics needed to enforce the high-level policy. The

gap exists when the trust boundary does not encompass all the semantics needed to construct the semantic universe defined by the high-level policy. Because of the fact that the enforcement systems all interpose on the hardware-software interface, the semantics in the trust boundary is limited. Therefore, the gap exists and semantics outside of the trust boundary is involved in policy enforcement. Consequently, the enforcement is not accurate because of the untrusted party's manipulation of the semantics. This is a challenging problem and exposes unexpected attack vectors to the untrusted party.

In the enforcement systems, both stages of the policy formulation process cross the inference gap. First, in order to formulate the effective policy, the enforcement systems choose a set of trusted states inside the trust boundary to represent instances of the entities in the high-level policy. The inference gap is crossed in the sense that the trusted states are associated with the high-level entities by using the identity-defining semantics beyond the trust boundary. Secondly, to formulate the binary policy, low-level states and events observable by the low-level enforcement mechanism are associated with the entities and operations defined in the high-level policy.

For example, a hypervisor can securely utilize the architectural states and the physical states of the guest VMs, i.e. register states and physical memory. These states are considered to be inside its trust boundary. Now consider a case where a policy concerns about user input data is desired. To enforce this policy using a hypervisor, the hypervisor needs to deduce the physical address of the input data from the guest's address mappings. However, the guest's address mappings are the semantics outside of the trust boundary of the hypervisor. The hypervisor crosses the inference gap in the sense that it uses this mapping to associate physical addresses to the high-level entity which is the input buffer.

3.1.3 The Approximation Function

Crossing the inference gap needs to define relations to bind the states inside the trust boundary to the intended entity at the high-level. A set of such relation is called an *approximation function*. The approximation function is where the fundamental conflict between tamper-proof and complete mediation manifests. Therefore, it is given an abstract treatment first and discuss the concrete conflicts afterwards.

Informally, an approximation function $F_{approx} : S \rightarrow E$ maps from the states in the trust boundary S to the intended entities E. For example, an enforcement system based on the kernel contains the states that correspond to the OS abstractions, such as files and processes etc., inside the trust boundary.

The construction of the approximation function is crucial to ensure that the view at the level of the high-level policy is accurate. The basic requirement is that the relations need to be *stable* in the sense that they cannot be manipulated illegally. However, when the semantics inside the trust boundary is limited due to ensuring tamper-proof, the involvement of the semantics outside of the trust boundary is unavoidable. Such semantics leads to instability in the relations, presenting the enforcement system an inaccurate view. As a result, the policy enforcement may be carried out against the incorrect subjects or objects, or is not invoked when intended. In other words, the complete mediation requirement is undermined.

Involvement of Semantics Beyond the Trust Boundary

The approximation function needs to be defined during both the policy formulation stages, because the inference gap is crossed in both stages. In the first stage, the relations about subjects and objects are defined. For example, the virtual addresses of the objects may be recorded. In the second stage, depending on the low-level enforcement mechanism, more relations may be needed and existing relations may be updated. For example, the virtual addresses are replaced by their physical addresses. More importantly, the point of mediation is also defined at this stage. In

other words, the low-level events that correspond to the operations in the high-level policy need to be identified and intercepted. Conceptually, the approximation function also includes relations that map low-level events to high-level operations.

In many cases, there may be no apparent relation between the low-level event and high-level operations, because the events in the low-level mechanism are concerned about operations that are more primitive than the ones in the high-level policy. In this case, the events that high-level operations occur need to be synthesized. The synthesis may be extremely challenging due to the inherent ambiguity of the high-level events. For example, 'reading a file' is inherently ambiguous and extremely difficult to define accurately using low-level primitives. There are at least three ways. It may be defined to be the execution of relevant code that handles files, or accessing in-memory file content by the processor, or issuing I/O commands to the external storage devices. Nevertheless, certain semantics outside of the trust boundary is needed to synthesize these events.

Implications

The consequence of using semantics beyond the boundary in the approximation function depends on the type of relation that uses the semantics. If the relations are defined in the first conversion stage, such semantics results in invalid identity of either the subject or object, leading to confusion of the identities. If the relations are defined in the second stage, besides the identity issue, certain events may be missed. Both consequences tamper the complete mediation requirement.

In the input buffer example in Section 3.1.2, the approximation function involves the mappings inside the guest page table, which is controlled by the untrusted kernel. Since the kernel still controls the guest page tables, the mediation by the enforcement systems may be incomplete. A more in-depth analysis of this scenario is given in Section 3.2.2.

In many systems, attempts are made to either verify the semantics each time it is used or to control the source of it. However, whether such hardening does

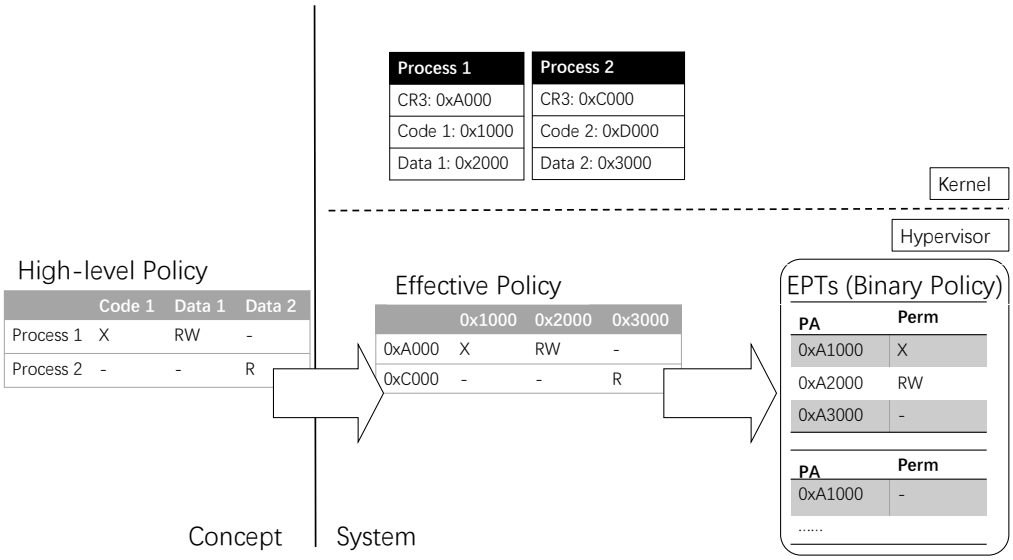


Figure 3.2: An Example of the Enforcement System

provide the required stability guarantee has not been systematically discussed. The effectiveness usually depending on specific implementation details.

3.1.4 Example Use of Semantics Beyond the Trust Boundary

Let us focus on the conversion stages and use a more comprehensive example to illustrate the details. The place of involvement of semantics beyond the trust boundary is highlighted while the specific issues caused is delayed to later part of the chapter.

The example is shown in Figure 3.2. The high-level policy is concerned about restricting the access by two processes to three objects. The policy is an isolation policy and the access matrix shows the authorization of each process's access to each object. Suppose this policy is to be enforced by a hypervisor based enforcement system. The policy needs to undergo two conversions. The first conversion converts it to the effective policy and the second to the binary policy. In the effective policy, the rows and columns in the original matrix are replaced by the states in the trust boundary of the hypervisor. Specifically, the processes are replaced by their respective `CR3` values. The objects are replaced by their virtual addresses (VAs) passed to the hypervisor in general purpose registers. The corresponding permis-

sions are kept intact. The second conversion breaks each row of the effective policy to a set of EPT as required by the MMU. The VAs are converted to the corresponding physical addresses (PAs). The EPTs are the binary policy here.

The hypervisor's trust boundary stops short at the boundary between kernel and hypervisor, since the kernel is assumed to be untrusted. However, both conversion stages require certain semantics from the kernel. In the first stage, the example system chooses the `CR3` register value to represent the process, since normally each process is designated with its own address space. Therefore, the subjects in the rows are replaced by their respective `CR3` values. Note that the relation between a particular `CR3` value and a process is kernel semantics. Directly using the `CR3` value essentially involves semantics beyond the trust boundary and can be potentially problematic. Similarly, the objects are replaced by the VAs. The relation between the VAs and the objects is also semantics under the control of the kernel.

In the second stage, the VA of each object are converted to PA to fill the first column in the EPTs. Because the memory allocation is done by the kernel, this conversion requires kernel's page table. Besides, because in this example, one set of EPT is only applicable for one process, the hypervisor needs to switch the active EPTs whenever the process on the CPU switches. In order to track the process switch, the hypervisor needs to interpose on the context switch events in the kernel, which is also kernel semantics. In this example, the permission in the EPT is sufficient to support the operations in the high-level policy, therefore, no event synthesis is needed.

3.2 Policy Formulation

In this section, the issues caused by the involvement of semantics beyond the trust boundary in the first conversion stage are presented. As discussed in Section 3.1.1, the enforcement systems ensure tamper-proof thus face limited scope of semantics. This limitation manifests in this stage in the sense that the semantics about

the identity of the subjects and objects is missing within the trust boundary, thus must come from the outside of the trust boundary. Subsequently, the main issue is the confusion of identity of either the subjects or the objects, which causes the enforcement systems to not mediate when they should. Therefore, the complete mediation requirement of the reference monitor is undermined. Three cases are presented depending on the type of subjects and objects, namely, process subjects, memory ranges and privilege levels.

3.2.1 Process Subjects

Some virtualization based systems proposed in the literature such as InkTag [51], Fides [86] and AppShield [29] isolate an entire process from an untrusted guest OS. The high-level policy in their concern is about individual processes. However, the virtualization based systems do not manage processes themselves, in order to keep the hypervisor logic simple and the TCB small. The typical approach is to piggy-back on the process management done by the guest OS and to override the guest OS's access to the resources allocated for the processes. Unfortunately, the hypervisor needs the identifying semantics during the piggy-backing. The identifying semantics reused by most virtualization based systems is the one-to-one relation between address space and process, which is defined by the guest OS. The address spaces can be identified via values in the `CR3` register which the hypervisor can easily monitor, therefore, many schemes, e.g. [51, 29] use the `CR3` register content to identify the process subject. It follows that in the effective policy, typically the value of the `CR3` register is chosen as the approximation of the processes. Thus, the approximation function contains mappings from the `CR3` values to the processes. This scenario is the example shown in Figure 3.2.

In the example in Figure 3.2, the `CR3` used for process 1 stores $0xA000$ when there is no attack. It follows that the hypervisor replicates a copy of the value $0xA000$ and defines a relation in the approximation function from this value to

process 1.

Since the relation between address space and process is semantics defined by the untrusted guest OS, binding `CR3` content to process is not secure. In the above example, because the kernel manages the processes, it can load $0xA000$ to `CR3` for process 2. The page at $0xA000$ can also be modified so that the imposter process can run its own code. When process 2 runs, the hypervisor mistaken it to be process 1, therefore, access to Code 1 and Data 1 is granted whereas according to the policy, it should not.

A variation is the case when the subjects are threads. Similarly, thread identifiers cause the same issue. Since threads of the same process share the same `CR3`, thread identification in the literature (e.g., AppShield [29]) relies on both `CR3` content and the kernel stack location as the identifier. Similar to the `CR3` falsifying, the kernel can also swap kernel stack locations and contents between the threads.

3.2.2 Memory Ranges

The second case concerns about memory ranges. The memory range is a form both the subjects and objects can take. For example, the policy subject are modules in a program such as TrustVisor [68] and SeCage [67]. Also, almost any objects in the high-level policy are memory ranges. An effective policy after conversion is shown in Figure 3.3

	0x1000 – 0x2000	0x3000	0x4000
0xA000 – 0x1A000	X	RW	-
0xC0000 – 0xCE000	-	-	R

Figure 3.3: The Effective Policy of Memory Ranges

Without loss of generality, let us suppose that subject 1 is between $0xA000$ and $0x1A000$ in its address space. Subject 2 resides in $0xC0000$ and $0xCE000$. The permission assignment is as shown in the figure. This arrangement is in accordance

with existing systems (e.g., TrustVisor [68], SeCage [67] and Fides [86]) that use the instruction pointer stored in `EIP` to identify a memory range subject X. `CR3` is also used jointly for identification when X is in userland. When a memory range is an object, the virtual address that is used to access the object is used to identify the object.

The issue is that the mappings in the guest page table are included in the approximation function. Since the boundaries of the memory range subjects and objects are delineated using virtual addresses, the approximation function now contains relation that maps certain virtual addresses to the high-level subjects and objects. However, the virtual addresses are designated by the untrusted kernel. So it is free to map the same object or subject at any virtual address range. As a result, the stability of the approximation function cannot be guaranteed. For example, suppose the kernel swaps the mappings of page $0x3000$ and $0x4000$. Subject 1's access to $0x3000$ will be allowed. However, the high-level object behind this address should not be accessed by subject 1.

Some systems [68, 86, 67] are aware of this issue and implement certain runtime measures such as checking the page table or providing its own set of guest page table. The runtime measures are not satisfactory to maintain the approximation function. First, it is costly for the hypervisor to traverse the guest page table, while the concerned thread is hanging. Second, guest page table traversing is subject to race condition attacks from the kernel, since the adversary may run on another CPU core. More details about the race condition are discussed in Section 3.4.1. Although the hypervisor can quiesce other cores as described in XMHF [93], it incurs a non-negligible performance toll.

3.2.3 Issues in SP^3

Given the approaches to represent the above two kinds of subjects, it is appropriate to review a system that depends on both methods to control access to memory,

which is the SP^3 [101] system. This design causes identity issues. Note that SP^3 is implemented on a para-virtualized platform, thus it involves some design that requires hardware modification if implemented on a hardware-assisted virtualization platform. Still, the discussion here focuses on the access control design, independent from the implementation.

SP^3 is a virtualization based system that aims to protect confidentiality of memory of application processes. Each process is assigned a SP^3 domain with an identifier sid. Each virtual memory page of the process is assigned an identifier that corresponds to a key. The hypervisor maintains an access matrix whose rows are the domains and the columns are the key identifiers. If a domain s is assigned access to a page p, the matrix records the authorization at (s, K_p) where K_p is the key identifier attached to the page p. The relevant design is shown in Figure 3.4.

Figure 3.4: The Access Control Design of SP^3

The original SP^3 design is not explicitly aimed at enforcing a higher-level security policy. Rather, it provides the mechanisms needed for potential security applications.

Suppose the mechanisms are used to enforce policy at the process level. In this scenario, there are issues with both subject identity and object identity in this design. The subjects are tracked by the hypervisor. Each domain is associated with a secured context saved on one of the kernel stacks which contains the domain identifier. During context switches, the hypervisor obtains the domain identifier of the next domain from the exception frame saved during the previous exception on

the kernel stack for that process. This approach is similar to the thread identification above using kernel stack. Therefore, the kernel can supply another kernel stack frame during context switch in order to fake to the domain identity.

The objects are essentially memory ranges grouped by key identifiers attached to the Page Table Entries (PTEs). It is not clear from the original design that whether the key identifiers in the PTEs are directly writable to the kernel, but still, they are readable. If they are writable, the kernel is free to modify the key identifier in the PTE of a virtual page to the key identifier of another process, so that the hypervisor decrypts the memory content when accessing any page accessible via this PTE. If they are not, the kernel can still modify the higher-level paging structure to replace the entire mapping in that range with another domain's leaf page table. Since page table sharing is allowed, it is difficult for the hypervisor to verify the intention of this modification. Still, the hypervisor will decrypt the pages because the key identifiers in the leaf page table matches the the ones in the access matrix.

3.2.4 Privilege Level Based Subjects

A special case in tracking the subjects is to track kernel / user switches. Most modern OS places the kernel in the same address space as the user applications and leverages on the Current Privilege Level (CPL) feature of the hardware to isolate them. Therefore, tracking kernel / user switches is equivalent to tracking CPL switches. This type of subjects is usually seen in the systems that enforces kernel integrity, such as SecVisor [80]. An example effective policy is shown in Figure 3.5.

	0x1000 – 0x2000	0x3000	0x4000
CPL = 0	X	RW	-
CPL = 3	-	-	R

Figure 3.5: The Effective Policy of Privilege Levels

In this case, the approximation function maps `CPL 0` to kernel and `CPL 3`

to others. The mapping itself is secure, because CPL is a hardware feature and cannot be forged by software. However, the effectiveness of this kind of system does not solely depend on CPL. The reason is that using CPL implies that any code, regardless of its origin, running with `CPL 0` is regarded as the kernel. This implication leads to two consequences. First, to prevent return-to-user attacks, the enforcement system needs to distinguish kernel memory and user memory in the same address space, in order to prevent execution of code in user memory while CPL is 0. This reduces the issue to the above memory range case and the same issue applies. Second, to prevent illegal injection of code into kernel memory via open interfaces such as kernel modules, such software level semantics needs to be included as well. In Section 3.3.4, it is shown that software level semantics also poses a challenge.

The recent speculative execution attacks, e.g. Meltdown [64] and Spectre [58], drive Linux to turn to separate address spaces for kernel and user spaces [3]. In this design, the kernel / user switch is equivalent to the address space switch above in Section 3.2.1.

3.3 Utilization by Low-Level Mechanism

The enforcement systems rely on certain low-level mechanism which is the MMU Therefore, in the second policy conversion stage described in Section 3.1, the effective policy is converted to a form necessitated by the low-level enforcement mechanism, called binary policy. The format of binary policy is dictated by the low-level mechanism. Our discussion in this section mainly uses the virtualization based systems as examples.

3.3.1 A General Approach

The low-level mechanism typically provide less than required number of subjects, and can only enforce a subset of the high-level policy at any time. For example,

the MMU essentially provides only one subject which is the hardware thread on the CPU and the active policy enforced is the current address space mappings in the page table. For enforcement of the full set of the policy, the high-level subjects are dynamically bound to the low-level subjects. Accompanying the subject switches, the active policy is dynamically switched as well. In the MMU case, the hardware thread acts on behalf of one high-level subject at any time. And when the subject switches, the page table also needs to be switched.

The implication is that it is required to divide up the access permissions assigned for the various high-level subjects into sets. Each set is essentially a bag of tickets for accesses to the objects allowed for a particular subject. In the MMU case, any virtual address is a ticket, the corresponding physical address and the permission assigned in the PTE control which object a subject can access with that ticket and with what permission.

The general approach is, therefore, as follows. Ideally, the binary policy should consist of sets with exactly the tickets allowed for a subject. By switching the active bag of tickets along with the change of dynamic binding between the high-level subjects and low-level subjects, the high-level policy is enforced and the complete mediation requirement is satisfied.

The challenges posed by this approach are as following. Firstly, the division of the access permissions requires semantics defined by the untrusted software, because the set of objects accessible to a subject is defined by the untrusted software. At least, the subject needs to access itself as an object. Therefore, correct division of permissions is a challenge. Note that because of this involvement, the binary policy is also an approximation of the high-level policy.

Secondly, tracking the high-level subject switches also involves semantics outside of the trust boundary. Such tracking is not always straightforward because there may be no low-level event that corresponds to the high-level semantics of subject switch.

Thirdly, in order to mediate all the operations defined in the high-level policy,

the low-level events that correspond to the operations need to be identified and intercepted. In other words, the high-level operations need to be synthesized. Because not all low-level events are relevant, policy mediation requires semantics of the untrusted software to filter the irrelevant ones. As a result, mediating the high-level events at the low-level is a challenge.

The impact of these challenges is presented in the rest of the section.

3.3.2 Division into Binary Policy Sets

In virtualization based systems, the binary policy is the paging structures needed by the MMU. In order to divide up the effective policy to sets of paging structures, the hypervisor needs to establish a number of relations between the virtual addresses in the effective policy and the physical addresses that points to the objects. Once established, the objects accessible to a subject are added to its bag.

In hardware-assisted virtualization, there are two stages in MMU's address translation. The VA to guest physical address (GPA) translation is the first, and the GPA to PA translation is the second. During the policy division, an important task that the hypervisor needs to perform is to find the GPAs so that the correct EPT entry can be located to fill the permissions. In order to acquire the GPAs, the guest page tables need to be traversed. However, the guest page tables are outside of the trust boundary, therefore, semantics outside of the trust boundary is used.

In fact, the division of the effective policy into bags of tickets can be implemented at either stage. In other words, the inclusion of the objects accessible into a subject's binary policy set can be achieved by configuring either the guest page table or the EPT. This option drives two different types of design. In the first type, the subjects are still distinguished and restricted by the guest page table. It implies that the EPT is shared among all the subjects and contains a union of all the objects and permissions assigned to all the subjects. In this design, one bag corresponds to one set of guest page table. InkTag [51], SeCage [67] and Fides [86] are example

systems that adopt this design. The second design is that the subjects are restricted using individual EPT sets. The guest page tables are typically directly reused. In this design, one bag corresponds to one set of EPT. TrustVisor [68] and AppShield [29] are both systems that adopt this design.

Nevertheless, both types involve the semantics outside the trust boundary which is the mappings in the guest page table, although the specific consequence depends on the way that such semantics is utilized. The consequences are below.

Shared EPT The systems that follow this design can be further divided into two subtypes. The first subtype directly reuses the guest page tables, as shown by SeCage [67]. The second subtype constructs a separate set of guest page table that contains replicated and verified mappings from the set of guest page table provided by the untrusted kernel. This type is represented by InkTag [51] and Fides [86].

The first subtype is not secure because of the race condition attacks described in Section 3.4.1. The consequence is that, the attempted to check the mappings in the guest page table can be defeated. The guest OS can inject or modify the mapping by modifying the guest page tables. Such modification alters the mapping of a virtual address to another physical address, thereby destroy the stability of the approximation function.

The second subtype replicates the mappings in the guest page table and verifies them, before writing the mappings into the guest page tables constructed by the hypervisor. The latter guest page tables are always used when the intended subject is executing. They also remain unchanged unless requested by the protected subject. Therefore, the above issues do not affect them. However, existing schemes are still vulnerable to the attacks related to permission revocation discussed in Section 3.4.2. The consequence is that the guest page tables provided by the hypervisor can still be modified.

Exclusive EPT This approach ensures that there exists only active mapping for one subject in each EPT set. However, there is still the issue of directly reusing the guest page table. The similar issues to the first subtype above apply.

Attacking Inaccurate Conversion

It should be noted that during the second conversion stage, the EPT arrangement needs to exactly follow the intended policy. If the subjects are intended to be separated from each other, the EPT sets are not allowed to contain mappings to the same physical memory. Unfortunately, SeCage [67] places EPT mappings to the shared data segment of the application in the EPT for the secret compartments and the untrusted compartment. In other words, the application semantics outside of the trust domain requires access to the data segment and this need is accommodated by the EPT. The design is shown in Figure 3.6.

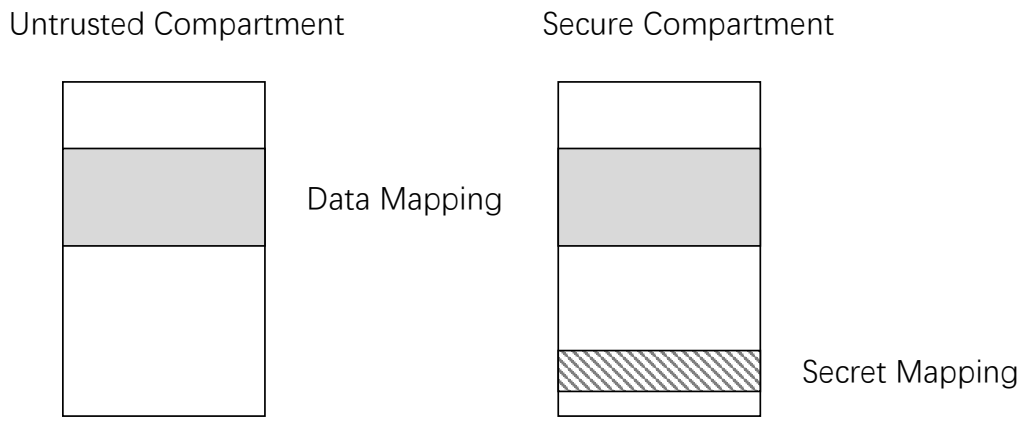


Figure 3.6: The EPT Arrangement in SeCage

This shared region between the secret compartment and untrusted compartment become a potential ground for leaking secrets in the secret compartment. The attacker can attempt to induce the code in the secret compartment to read the secret and write to the shared memory which is readable by the untrusted compartment.

3.3.3 Detecting Subject Switches

The second challenge in Section 3.3.1 concerns switching active policy along with the subject switches on the CPU. The subject switches are also semantics beyond the trust boundary. The enforcement systems need to monitor every switch, and follow up with policy switches, which requires the enforcement systems to obtain

semantics outside of the trust boundary. If certain switch is missed, the applied policy may be wrong, thereby undermining complete mediation.

Depending on the type of subjects that needs to be tracked, the amount of semantics involves also differs. For process subjects, it is straightforward to track the switches, because it is a straightforward for a hypervisor to track `CR3` switches. However, for other types of subjects, more semantics is involved and there are potentially more issues.

Subject Switch within an Address Space

Certain virtualization based systems enforce policies that concern about finer-grained entities than an entire address space. Assume the entity of concern is a consecutive range R in the address space. Different policies are applied to R and other parts of the address space. Therefore, it is necessary to detect when the execution switches between R and the rest. Take the policy in Section 3.2.2 as an example, R may be between $0xA000$ and $0x1A000$.

It is challenging to detect such switches because they do not necessarily incur any events defined in the hypervisor's trust boundary. In essence, this type of switches are control flow transfers which are not within the original design considerations of the hypervisor. Nevertheless, there are two workarounds proposed in the literature.

In the first method, the hypervisor disables the execution permission of all memory pages, except those for R. As a result, whenever the execution leaves R, an access violation is reported by the MMU and the control then traps to the hypervisor. However, this method have a hidden issue. Before the hypervisor applies the binary policy sets, the hypervisor has to know exactly the boundary of R, which requires the semantics in the guest page tables. Therefore, this method depends on the correct construction of the binary policy sets which is discussed in Section 3.3.2.

The second method is to arrange the subjects to actively signal the switches. As used in SeCage [67], the entry and exit of R are instrumented with code that invokes

the `VMFUNC` instruction to switch the underlying EPT.

Attacking Un-mediated Subject Switch However, this method is not secure because the subject switches completely evade the mediation of the hypervisor. As a result, there is no guarantee that the hypervisor enforces complete isolation among the subjects. In other words, certain state switches are left to the protected application itself. Amongst the states, the secure stack used by the secret compartment is of particular concern. In SeCage's design, the *trampoline* and *springboard* are code fragments used to perform the state switches between the secure compartments and the untrusted compartments. The fragments contain instructions that replace the stack according to the direction of the switch. A malicious kernel can intercept the instructions and emulate them, e.g. by using break points. During the emulation, a wrong stack pointer is moved to `ESP`, so that the secure compartment uses the kernel supplied stack. The kernel then skips the emulated instruction on return. The execution continues to the secure compartment and the secure compartment uses the insecure stack.

3.3.4 Event Synthesis

Since the binary policy needs to faithfully enforce the high-level policy, all the operations performed on the objects need to be intercepted. As introduced in Section 3.3.1, the approach is to capture the low-level events that could be an indication of the high-level operations and filter the irrelevant ones. In essence, the enforcement system needs to construct a set S_E which consists of pairs (e, v), in which e refers to a particular low-level event, while v is a combination of the trusted states. The element in S_E are chosen so that when a low-level event e occurs, if the trusted states match the value v, a high-level event is uniquely identified. S_E is included in the approximation function.

The main issue is that e may not even be triggered when certain high-level operation occurs. Plus, in order to specify v, the enforcement systems need to reuse

the semantics outside of the trust boundary to interpret the states also outside of the trust boundary. The interpretation may be misled because the untrusted software manipulated the states.

High-Level Software Events

The most prominent example is the software level event: dynamic allocation of memory. This type of events are of interest, because they can imply policy updates when the newly created object is of security concern. The typical approach to acquire the software level semantics is to insert an agent into the software. The agent hooks the concerned code that "handles" the event to intercept the invocation. Then it explicitly calls the enforcement system to inform the captured event. For integrity, the memory page where the hooks reside are set as read-only.

With this approach, much software semantics outside of the trust boundary is included in the approximation function. Examples include but not limited to the virtual memory layout, the one-to-one relation from the invocation of certain code to the concerned event, the parameters and the stack layout etc. Any manipulation in these leads to incorrect approximation of the software events, thus undermines the complete mediation requirement.

Hook Bypassing Without eliciting too much software level details, here presents an issue with the hooking technique. Although the hooks are set as read-only, there is still no guarantee that all invocation of the hooked function is intercepted. The reason is that the invocation is made via virtual address but the read-only permission must be set on GPA. A malicious kernel can simply copy the page that contain the hooks, and maps the virtual address to the copy. The hooks are never triggered while the hooked application runs normally.

To defend against this kind of attack, the guest page tables need to be monitored and tracked. However, as discussed in Section 3.4.1, checking the guest page table faces the race condition issue and is not always secure.

The above approach is adopted by systems such as HUKO [99] which enforce

a mandatory access control model. In order to label all objects, it is necessary for these systems to track all allocations and deallocations in the kernel space. HUKO inserts a protected and trusted component into the kernel itself to intercept the allocation and deallocation events and notify the hypervisor. Other systems that performs stealthy debugging [37, 41] also potentially face this issue when the kernel is malicious, although in their original design, they do not explicitly consider the kernel to be malicious.

3.4 The Impact of Concurrency

3.4.1 Race Conditions

Concurrent access by multiple cores could lead to race conditions. They pose a threat because the semantics could be manipulated unexpectedly. In a multicore setting, walking the guest page table at runtime is not secure because it is subjects to race condition attacks. On a single core system, the guest OS is paused when the CPU traps to the hypervisor so the problem does not exist. However, on a multicore system, when one core traps to the hypervisor or execute the isolated code, the guest OS on other cores are still running. So there is ample opportunity for the guest OS to manipulate the guest page table when the hypervisor is walking it or when isolated code is using it for translation.

Specifically, the effective policies enforced by the hypervisor use virtual addresses (VA), because user space programs in general are not aware of any physical addresses. In the second policy conversion stage, the hypervisor faces the challenging problem which is to find out the corresponding GPA so that the right EPT entry can be updated accordingly. In order to do so, the hypervisor needs to traverse the guest page tables of the virtual addresses at runtime in order to find the GPA. Software page table traversing cannot be done instantaneously. During its traversal, the kernel can modify the page table entries and feed the hypervisor with a wrong

mapping.

Consider that the hypervisor attempts to lock the guest page table as an example. Walking page table is a rather lengthy operation because it involves a number of memory loads and stores. Thus, the guest OS running on another core has a non-negligible time window to change one of the leaf page tables after it is verified. In order to do so, the guest simply needs to write a few bytes into the higher-level page. The result is that the hypervisor ends up protecting the wrong physical page, while another page is actually used to perform VA-to-GPA mapping.

3.4.2 Permission Revocation

Even if the binary policy are correctly constructed, there is still a need to perform revocation once the active policies are switched or when policy update happens. The revocation needs to be complete in that any component that could hold a copy of the permission needs to be considered, including any buffers system-wide.

In virtualization based systems, the address translation is part of the binary policy. Meanwhile, recently used address translations together with the permissions are cached in modern CPU's Translation Lookaside Buffer (TLB). The caching results in implicit replication of the binary policy. Such replication causes undesired consequences when a multicore platform is considered.

Unlike data cache and instruction cache, the consistency between the TLB and the page tables in the main memory is maintained by the software, instead of the hardware. Therefore, when an address mapping is updated, the software needs to explicitly *invalidate* corresponding TLB entry.

Moreover, the hardware does not enforce coherence among the TLBs on different cores. All such operations need to be explicitly carried out by software as well. When more than one core access an address space, the core that changes the mapping is supposed to perform *TLB shootdown* to invalidate any existing entries on other cores. Typically, it is achieved by using the Interprocessor Interrupts (IPIs).

Specifically, the initiating core fires an IPI to each core that needs to invalidate its TLBs. On modern x86 platforms, the Advanced Programmable Interrupt Controller (APIC) interfaces with the bus for receiving and sending IPIs. The IPI is received by the other cores and treated exactly the same way as an external interrupt. A handler is invoked and the specified TLB entry is invalidated. In this way, the consistent view of the address space is maintained across all CPU cores.

3.4.3 TLB-Related Attacks

There are two attacks that can be launched on a multicore platform by exploiting the TLB. The first is the stifling attack that actively abuses the TLB and IPI across CPUs. The second exploits a hardware performance feature called Virtual Processor ID (VPID) to keep stale entries. The attacks are performed on two open-source micro-hypervisors, BitVisor [82] and XMHF [93], running on a PC with multiple cores.

In the attacks, consider a typical isolation scenario where the hypervisor receives the request from a security sensitive application at runtime, and then sets the read-only permission in the EPT entry for the application's code page. The objective of the attacks is for the malicious OS to successfully modify the protected page without the write permission on the EPT entry. The general idea behind the attacks is to use a stale TLB entry so that the core continue to use the write permission granted before the EPT update.

The Stifling Attack

The *stifling attack* prevents the CPU core controlled by the malicious thread from responding to the hypervisor's TLB shootdown, so that its stale TLB entry is not invalidated. Note that trapping to the hypervisor cannot be denied by setting the interrupt masking bit (namely `EFLAGS.IF`), because the hardware ignores it whenever the External-interrupt Exiting bit in the Virtual Machine Control Structure (VMCS)

is set.

The attack exploits a hardware design feature to block all maskable external interrupts, including the IPI used for TLB shootdown. According to the hardware specification, the IPI handler is expected to perform a write to the `End Of Interrupt (EOI)` register in the local APIC before executing an `iret` instruction. The EOI write operation signals the end of the current interrupt handling and allows the local APIC to deliver the next IPI message (if any) to the core. If no such write is performed, the local APIC withholds subsequent IPIs and never delivers them. The guest can access the physical APIC because in hypervisors for memory isolation, the APIC is typically not virtualized in order to minimize the TCB.

As depicted in Figure 3.7, suppose that the victim application occupies core_v while two malicious kernel threads occupy core_1 and core_2. The attack steps are described below.

1. At core_v: The victim application starts to run and writes data into a memory buffer.

2. At core_1: The malicious kernel maps the guest physical address of the buffer into its own address space by changing its guest page table. It reads the buffer so that the corresponding EPT entry is loaded in the TLB of core_1. It also disables interrupt and preemption so that it is not scheduled off from core_1 in order to avoid any TLB invalidation due to events within the guest.

3. At core_2: Another thread of the malicious kernel sends an IPI to core_1 by using a legitimate IPI vector for OS synchronization.

4. At core_1: The malicious IPI handler returns without writing to the EOI register of the local APIC. As a result, subsequent IPIs are never accepted by core_1.

5. At core_v: The victim issues a hypercall for memory protection. The hypervisor updates the EPT for all other cores to disallow accesses. It broadcasts an

IPI to trigger VM exit on other cores.

6. At $core_1$: The IPI from $core_v$ is not delivered to $core_1$. The kernel thread can continue to read/write the isolated data buffer without trigger any EPT violation, because the core's MMU uses the EPT entry in the TLB which has the stale permissions assigned prior to the hypercall.

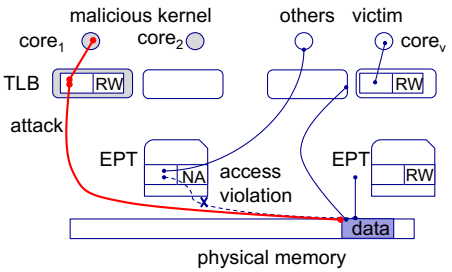


Figure 3.7: Illustration of the stifling attack bypassing the EPT's access control over the victim's data. The attacker controls $core_1$ and $core_2$.

The attack is implemented on BitVisor [82] with necessary changes including new code to change EPT permission bits for isolation and an interrupt handler for TLB shootdown. The experiment shows that the kernel successfully writes to the protected buffer, even though the access permission in the corresponding EPT entry has been changed into read-only.

One possible countermeasure to the stifling attack is to virtualize local APICs so that the hypervisor intercepts the external interrupts and enforces `EOI` writes. However, this approach not only increases the hypervisor's code size and complexity, but also has performance tolls as it is recommended to remove the hypervisor from the code path handling interrupts for better efficiency [48, 91].

An alternative is to resort to non-maskable interrupts (NMIs) instead of IPIs. NMIs are delivered immediately by the local APIC to the CPU core as they are usually sent by hardware such as watchdogs to indicate critical hardware failure which needs immediate attention. However, it is strongly discouraged to use software to generate NMIs because of its complex handling. Moreover, it requires a high level of expertise to implement a proper NMI handler [77] because it needs to

deal with recursive execution. Briefly speaking, once an NMI is delivered to a core, subsequent NMIs are blocked until the core executes `iret`. If the NMI handler causes any exception, the exception handler's `iret` immediately allows the next NMI to be delivered while the present one is still in processing. From the system perspective, it is risky to use the hypervisor to issue and handle NMIs.

Virtual Processor ID (VPID) Attack

XMHF [93] is an open-source micro-hypervisor on x86 platforms that explicitly takes the multicore setting into its design consideration. To deal with concurrency in the hypervisor space, XMHF enforces a *single threaded* execution model for the hypervisor. When one core is trapped to the hypervisor space, it "quiesces all other cores" by broadcasting a Non-maskable Interrupt (NMI) which triggers a VM exit and effectively pauses the execution of all other threads across the system. Therefore, it is not subject to the stifling attack. Nevertheless, it still has another TLB-related vulnerability.

Recent generations of x86 processors introduce a feature called Virtual Processor ID (VPID) to avoid unnecessary TLB invalidation induced by VM exit events. Identifiers are assigned to address spaces of each virtual CPU and of the hypervisor and tagged to their TLB entries. When a TLB entry is used during translation, it is considered a hit only when its VPID tag matches the VPID of the present address space. With this extra checking, the hardware does not need to invalidate *all* TLB entries during VM exit.

Although improving the performance, this technology has an unexpected security side effect. Since not all TLB entries are evicted by the hardware during a VM exit, the stale entries of the guest must be explicitly invalidated by the hypervisor. However, the XMHF hypervisor neglects this issue. It assigns VPID 0 to the hypervisor and VPID 1 to the guest. Unfortunately, there is no explicit invalidation of TLB entries tagged with VPID 1 when handling the quiesce-NMI. With this loophole, the following attack can be launched by the guest OS to write the page set as

read-only by the EPTs. The system setting is the same as the stifling attack shown in Figure 3.7.

1. At core_v: The victim application starts execution. It allocates a page and requests memory isolation.

2. At core_1: The malicious kernel running on core_1 maps the buffer into its own space, reads it once so that a TLB entry is loaded by the MMU. It disables interrupt and preemption so that the TLB entry is not evicted by events in the guest.

3. At core_v: The victim application performs a hypercall to the XMHF hypervisor. The hypervisor issues an NMI to trap other cores and sets the read-only permission bit in the relevant EPT entry after CPU quiesce.

4. At core_v: The execution returns to the victim application.

5. At core_1: The guest OS resumes its execution. Due to incomplete TLB invalidation, the stale entry is not removed. The guest OS continues to read and write the page, regardless of the permission in the current EPT.

The implementation involves a hypervisor application, or hypapp in XMHF's terminology, based on XMHF APIs. The hypapp takes an address of a physical page as input and sets its access permission in EPTs as read-only. The hypapp is invoked via a hypercall from an application bound to a core. The kernel runs a malicious thread on another core to continuously access the page. It is observed that the malicious thread keeps a stale TLB entry and successfully writes the target page without triggering EPT violation.

3.4.4 Implications

The revocation issue impacts all systems that reuse memory pages previously used by the untrusted guest. If the memory pages contain sensitive data itself such as

[68, 67, 29, 86], the guest can directly access the data with stale permission. If the memory pages with data are encrypted as in InkTag [51], the guest kernel can still manipulate the guest page tables used by isolated execution, i.e. the hypervisor page table in InkTag's term. These page tables are allocated by the kernel and handed over to the hypervisor, therefore, permission revocation is needed. Even though InkTag verifies the guest page tables and do not directly reuse them, its security is still undermined.

3.5 Discussions

In this section, a few related issues about the mismatch in scope of semantics and policy enforcement are discussed.

3.5.1 Memory Monitors

The discussion in previous sections can be applied to monitor systems and virtual machine introspection (VMI) systems. Copilot [71], KI-Mon [60] and Vigilare [70] are example systems that utilize a separate hardware component for monitoring purposes. Such hardware-based monitors are proposed to monitor kernel code integrity, static data integrity and kernel objects. The VMI systems [84, 81, 24, 49, 42, 50] leverage the hypervisor's support to introspect the memory states of virtual machines.

These systems do not necessarily enforce any policy regarding the subjects such as processes. However, they are concerned about the objects. Also, all the systems are isolated for the tamper-proof requirement. Therefore, the possible mismatch between the semantics within the trust boundary and the semantics needed to monitor the objects still exists. Thus, the discussion about involvement of semantics outside of the trust boundary in the approximation function, and the effect on object identity and events is still applicable.

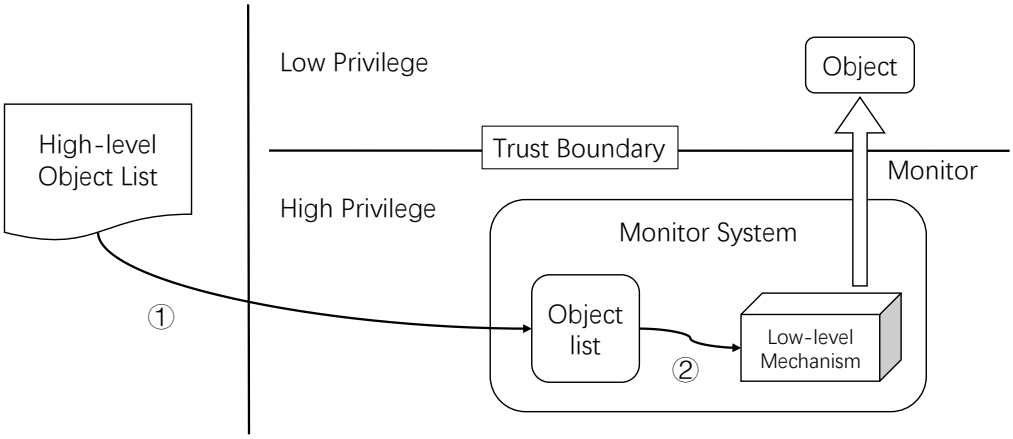


Figure 3.8: The enforcement system in the monitor scenario

Figure 3.8 presents a conceptual illustration of the monitor systems. The high-level object list undergoes two stages of conversion, marked as arrow 1 and arrow 2 in the figure. In the first stage, the high-level objects are converted to their corresponding addresses at runtime, i.e. virtual addresses. In the second stage, the virtual addresses are further converted to physical addresses so that the low-level mechanism can directly monitor. The low-level mechanism differs depending on the specific system. The hardware-based monitors use Direct Memory Access (DMA) or bus sniffing, while the VMI systems rely on the MMU.

There are two major issues in these systems. First, the semantics in the kernel page table plays an important role. Second, the isolation in these systems removes the possibility to interpose on important events in the CPU. Therefore, these systems may not be able to respond to state changes and miss the monitor target.

Kernel Page Table Since the kernel under monitor still controls the full set of page table, the monitors may be presented a completely false view of the entities that it is monitoring. The existing approach is to assume certain virtual memory layout that is normally followed by the kernel under monitor. This approach in essence does not attempt to verify the use of the semantics beyond the trust boundary. Worse, the page table can even be modified by legitimate means. For example, the `kfork()` system call is a legitimate way provided by Linux kernel to load a new kernel image. Afterwards, the system soft-reboots to run using the new image,

while leaving the old kernel image intact in physical memory. Once executed, the monitors, which still read the old image, will be unaware of the new kernel.

CPU States The memory monitors are not designed with the capability of tracking activities on the CPU. Especially so are the hardware monitors, because they are separated from the CPU. A malicious kernel may modify the CPU states to violate fundamental assumptions and mislead the monitoring, e.g. by modifying the paging mode or even turning on virtualization.

3.5.2 Runtime Updates and Policy Coherence

Since multiple binary policy sets may contain access permissions to the same object, the enforcement systems need to ensure that the binary policy sets are coherent, i.e. in accordance with the high-level policy and do not conflict with each other. For example, if the policy states that only subject X has access to a certain page P, only the EPT for X should allow access to page P; all other EPTs should deny such access. Otherwise, there may be contradicting rules in different policies which confuse the hypervisor during enforcement.

However, since the access permissions are scattered in the binary policy sets, the enforcement systems usually face a daunting task of enumerate all of the sets to ensure coherence. Beside the obvious performance implications, the ambiguity in the policy may also hamper effective verification.

3.5.3 Functionality

An ideal enforcement system maintains full compatibility with existing functionalities, which requires them to understand the rich semantics in the untrusted software. One of the functionality often left unaddressed is swapping. An important aspect of virtual memory is that the backing page for a virtual page can be freely swapped out and in by the guest OS. Therefore, the guest OS do have legitimately reasons to access the data. Except InkTag [51], existing systems do not describe explicitly

how their policy enforcement can cooperate with swapping.

Swapping is an interesting issue because it provides the guest OS a chance to freely modify both the content in a swapped page and the mappings defined in the guest page table when a page is swapped in. Integrity protection on the page content is not enough, and the guest OS must also be checked so that it places a page at its original location. The paraverification-style of verification does not apply here because there is no application to specify the intention; swapping is purely initiated by the kernel. It suggests certain deep interaction is needed between the hypervisor and the guest OS and this remains an open question.

3.5.4 Forced Serialization of Concurrent Accesses

The compatibility with swapping introduces another unexpected effect. The feature of InkTag's approach is that it allows read access to all the physical page via both EPTs. In other words, instead of denying access via the untrusted EPT, it separates the physical memory into two mutually exclusive sets which are both readable: a trusted set and an untrusted set. The content of a page is encrypted when it is mapped in the untrusted set, and decrypted in the trusted set. The hypervisor encrypts and decrypts the page content when switching the set that a page is mapped in. The switch is triggered by guest access. When the page is not mapped in the set that the guest attempts to access, the EPT page fault is handled by the hypervisor and the page's set is switched.

This approach is effective on unicore platforms, however, it cannot be scaled to multicore platforms. Because the sets are exclusive, there cannot be concurrent accesses from more than one core via different EPTs. The hypervisor needs to serialize such accesses and the cores need to wait until the other cores finish. Therefore, this approach introduces significant overhead and is not scalable. The hypervisor can optimize and keep two read-only copies of the same page so that read access can happen concurrently. However, writes from the trusted view still

have to be intercepted and synchronized by the hypervisor, which requires the hypervisor to decrypt the page first, perform the synchronization and encrypt the page again. These operations are still not efficient.

3.6 Possible Solutions

The root cause of the aforementioned issues is the mismatch between the semantic information available within the trust boundary and beyond. The solutions should address this root cause. The following discusses a few possible directions.

3.6.1 Expanding the Trust Boundary

The traditional reference monitor inside the OS kernel such as SELinux [83] does not suffer from the problem because the trust boundary, i.e. the user/kernel boundary, encompasses all needed semantics. Therefore, there is no mismatch in semantic scope. However, expanding the trust boundary to the user/kernel boundary faces the drawback that the kernel contains a huge TCB.

The TCB issue can be potentially remedied by privilege separation. The extreme form of privilege separation is to resort to a microkernel. However, this approach likely encounters practical barriers of converting existing infrastructure to another. Nested Kernel [36] attempts to strike a balance without drastically modifying the architecture, however, its design only includes page tables inside the trust boundary. Other kernel semantics is not covered.

3.6.2 Self-Supplied Semantics

It is also possible for the enforcement systems to resort to alternative designs. The pursuit for small TCB drives the design in many systems towards the direction that still leaves all resource management to the untrusted kernel. The enforcement systems only try to deprive the kernel access to resources that previously belong to it.

Therefore, the enforcement systems all need to understand the internal working of the untrusted kernel, thus the semantics causes issue.

In an alternative design, the enforcement systems can perform some light-weight resource management. The untrusted guest software is only utilized as a library and only handles tokens to the real resources. The semantics is still controlled by the enforcement systems, with the benefits of rich functionality from existing software.

SecPod [98] hints on this approach in that the hypervisor manages the entire paging structures including the guest page tables. The kernel's computation about address mapping is recorded in shadow page tables never used for policy enforcement. Therefore, the VA to GPA mapping is controlled. However, the control does not extend beyond the mappings in SecPod's design. The design presented in the next Chapter, FIMCE, also falls into this category.

3.6.3 Hardware Assistance

Another option is to seek assistance from hardware. If the scope of semantics in the hardware is aligned with the high-level policy, the burden on the enforcement system is minimized. Thus, the issues incurred by the misalignment can be greatly alleviated.

An example can be seen in the design of SGX. Although the address space layout is designated to the untrusted kernel, the hardware records VA to PA mappings when the enclaves are constructed. On every subsequent memory access, the hardware compares the VA and the PA used by the access and the recorded version. The access is only allowed if there is a match. SGX hardware understands the semantics about the virtual address space, therefore, it can perform the checks in hardware. The drawback of this approach is also obvious, though. Since hardware modification is needed, legacy platform cannot benefit from the new design.

3.6.4 Restricting Untrusted Software

Lastly, it is an option to impose restrictions on the untrusted software so that the semantics outside of the trust boundary can only behave in expected ways.

The paraverification idea in the design of InkTag offers certain insight into a software-only approach to restricting the untrusted software. The kernel's modification on the address space of the isolated application must be coupled by a token supplied by the initiating application. The hypervisor verifies the intention represented by the token and the action performed by the kernel. If they match, the modification is allowed. However, this design needs to be specially adapted for individual types of semantics. For example, special design may be needed for operations on files. It remains an open challenge to design a generic and practical verification mechanism in software.

Chapter 4

Enforcing Isolation with Fully Isolated Micro-Computing Environment (FIMCE)

In this chapter, a new isolation paradigm is proposed to enclose an entire computing environment including the CPU core, the memory region, and (optionally) the needed peripheral device(s). The new system is named as *fully isolated micro-computing environment* or FIMCE. It avoids all aforementioned security pitfalls of virtualization-based memory isolation, constructs a secure isolated execution environment with less than 1% of overhead. The principle of FIMCE is to control the involvement of the kernel semantics in the isolation policy enforcement. Meanwhile, FIMCE performs minimum resource management so that necessary semantics is provided by itself.

The Software Guard Extension (SGX) to the x86 architecture is a hardware based isolated environment and offers strong isolation guarantees. SGX allows applications to create so-called *enclaves* as an isolated environment within its address space. Once created, only code inside an enclave is able to access memory assigned to the enclave. All other accesses are not allowed, including SMM. While the isolation guarantee of SGX is strong, certain capability is still not included such as the

ability to perform I/O. In this chapter, a comparison of FIMCE and SGX is given and potential ways to integrate both for strong security guarantees are discussed.

4.1 FIMCE Architecture

In a nutshell, a FIMCE is an isolated computing environment dynamically set up by the hypervisor to protect a security task. The hypervisor enforces the isolation between the guest and a FIMCE by using virtualization techniques. A FIMCE consists of the minimal hardware and software satisfying the task's execution needs. Its default hardware consists of a vCPU core, a segment of main memory and a Trusted Platform Module (TPM) chip. If requested, peripheral devices (e.g., a keyboard) can be assigned to a FIMCE as well. These hardware resources are exclusively used by the task when it is running inside its FIMCE.

The code running inside a FIMCE consists of a piece of housekeeping code called the *FIMCE manager* and the software components that comprise a set of *pillars*, as well as the security task itself. A pillar is in essence a self-contained library the security task's execution depends on. For instance, a TPM pillar provides the TPM support to the task. A FIMCE hosts a *single* thread of execution starting from the entry of the FIMCE manager. All code in a FIMCE runs in Ring 0 and calls each other via function calls. Hence, there is no context switches within a FIMCE.

Core Isolation The vCPU core used by the protected task is isolated from the untrusted OS for two reasons. Firstly, it prevents the hypervisor from understanding the kernel's scheduling semantics which causes a series of issues as discussed in the previous chapter. Secondly, it prevents untrusted OS from interfering with the FIMCE by using the inter-core communication mechanisms such as INIT signals. In modern systems, such signals can be triggered via programming the APIC. Note that core isolation does *not* mean that a physical CPU core is permanently dedicated to a protected task. In fact, the task can migrate from one core to another. However, while it is running, it exclusively occupies the vCPU and does not share it with other

threads, until it voluntarily yields the control or is terminated by the hypervisor.

In addition, the hypervisor sets up the virtual core of the isolated environment such that external interrupts, NMI, INIT signal and SIPI are all trapped to the hypervisor. By default, INIT and SIPI automatically trigger VM exit. To intercept NMI, the hypervisor sets the NMI exiting bit in the pin-based VM-Execution control bitmap of the VMCS structure. To handle external interrupts including possible IPIs, empty interrupt handlers are installed by the hypervisor inside the FIMCE, while the security task may choose to replace them with its own handlers to manage peripheral devices.

Memory Isolation Memory isolation is still indispensable in FIMCE design. FIMCE guarantees that the entire address translation process is out of the guest kernel's reach. All data structures used in the translation process such as the guest page table and the Global Descriptor Table (GDT) are separated from the kernel. Moreover, the physical memory pages used by a FIMCE are allocated from a pool of pages priorly reserved by the hypervisor. Using reserved memory pages controls the involvement of kernel semantics about the relation between identity and memory pages. It deprives the kernel of the chance to influence the mappings used inside the FIMCE. Therefore, the issues about the involvement of kernel semantics about object identities are avoided.

Different from conventional virtual machines, it is not necessary to turn on memory virtualization for the FIMCE. Without any EPT, the FIMCE's MMU uses the page table to translate a virtual address directly to a physical address. The main benefits of the setting are to save the hypervisor's workload of managing EPTs and to speed up FIMCE's memory access. Figure 4.1 explains the memory setting for a FIMCE.

When launching a FIMCE, the hypervisor sets up the page table according to the parameters that describe the virtual address space. To prevent the code in the FIMCE from accidentally or maliciously accessing pages outside of the isolated region, the hypervisor does not allow it to update the page table. In other words, the

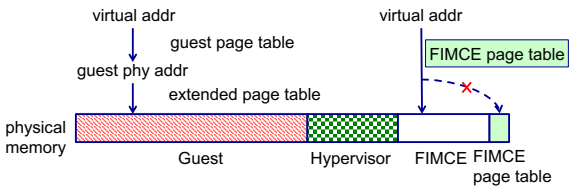


Figure 4.1: Memory isolation for FIMCE without EPT

address mapping is fixed. For this purpose, the page table does not have mappings to its own physical pages and updates to the `CR3` register and other control registers are trapped to the hypervisor. (Note that different paging modes make different interpretation of the same paging structure, which may introduce loopholes that allow the FIMCE code to breach memory isolation.) Lastly, the hypervisor configures Input-Output Memory Management Unit (IOMMU) page tables to prevent illegal DMA accesses.

I/O Device Isolation FIMCE utilizes DMA remapping and interrupt remapping supported by hardware based I/O virtualization, together with VMCS configuration and EPTs to ensure that the FIMCE has exclusive accesses to peripheral devices assigned to it. Firstly, any I/O command issued from the guest to the FIMCE device should be blocked. For port I/O devices, the hypervisor sets the corresponding bits in the guest's I/O bitmap. For Memory Mapped I/O (MMIO) devices, the hypervisor configures the guest's EPTs to intercept accesses to the MMIO region of the device.

Secondly, interrupts and data produced by a FIMCE device are only bound to the FIMCE core. For this purpose, the hypervisor configures the translation tables used by DMA and interrupt remapping. The former redirects DMA accesses from the device to the memory region inside the FIMCE and the latter ensures that interrupts from the device are delivered to the FIMCE core rather than other cores of the guest.

4.2 The Lifecycle of FIMCE

When the platform is powered on, its DRTM is invoked to load and measure the hypervisor which in turn launches the guest OS. A FIMCE is launched only when the

hypervisor receives a hypercall to protect a security task. After the task finishes its job, it issues another hypercall within the FIMCE to request FIMCE shutdown. The following describes the main procedures the hypervisor performs during FIMCE launch, runtime and termination.

4.2.1 FIMCE Bootup

The hypervisor's main tasks here are to allocate the needed hardware resources and to set up the environment for the security task.

Hardware resource allocation The default FIMCE hardware resources comprise a CPU core, a set of physical memory pages, and the TPM chip. To make a graceful core ownership handover from the guest to the FIMCE, the guest OS's CPU hot-plug facility is utilized to remove a physical core from the guest. To unplug a core, the kernel migrates all processes, kernel threads and IRQ handlers to other cores and only leaves on the core the idle task which puts the CPU to suspension. The kernel also configures the unplugged core's local APIC so that local interrupts are masked. Note that the guest OS cannot prevent the hypervisor from gaining control of a core after a hypercall. After removal, the (benign) kernel will not attempt to use the unplugged core any longer. The hypervisor then allocates a new VMCS for the logical core of the FIMCE and initializes the VMCS control bits such that core isolation takes effect once the FIMCE starts execution.

The physical memory used by the FIMCE is allocated by the hypervisor from a reserved memory frame pool so that they are not accessible from the guest. Note that the dynamic memory allocation approach adopted by existing systems is not secure due to the stifling attack. Suppose one page is allocated inside the guest and later isolated for use by FIMCE, in order to isolate the page, any existing TLB entry for that page needs to be invalidated. However, due to the stifling attack, such invalidation is not always possible.

To setup for the FIMCE's access to the TPM chip, the hypervisor blocks other

cores' access to the TPM's MMIO registers by configuring the EPT. The FIMCE may also contain peripheral devices. If the security task requires accesses to a peripheral device, the hypervisor leverages the IOMMU's capability to redirect DMA accesses and interrupts to ensure the FIMCE's exclusive ownership. The hypervisor also configures the I/O bitmap in the guest's VMCS to intercept any guest access to the device through Port I/O, and configures the EPT for accesses to the MMIO regions. The hard disk is not considered in the design because disk data can easily be protected by cryptographic means.

Environment Initialization After all hardware resources are allocated, the hypervisor sets up the FIMCE environment for the task's code to run inside. Firstly, the hypervisor initializes a minimum set of data structures required by the hardware architecture. These include a Global Descriptor Table (GDT) with a code segment descriptor, a data segment descriptor and a task-state segment (TSS) descriptor. The design uses simple flat descriptors for both code and data segments. The descriptors are configured with a Descriptor Privilege Level (DPL) of 0, which means that all FIMCE code runs with Ring 0 privilege. There is no adverse consequence to elevate the FIMCE code's privilege, because it cannot access the guest or the hypervisor space due to full isolation. It cannot attack other FIMCE instances neither, because each FIMCE instance is independent of each other and is launched with a clean state.

An Interrupt Descriptor Table (IDT) is also initialized with entries pointing to empty interrupt handlers. With the IDT, proper interrupt handlers can be installed if the security task requests I/O support. In addition, the hypervisor sets up the page table for the FIMCE. The hypervisor also flushes the FIMCE core's TLB to invalidate all existing entries. Lastly, it properly fills in the VMCS fields to complete environment initialization.

Code Loading The hypervisor first loads the FIMCE manager code into the FIMCE memory. The FIMCE manager is the hypervisor's delegate for housekeeping purposes, e.g., setting up the software infrastructure. The hypervisor then loads

the security task. Based on the parameters in the FIMCE starting hypercall, the hypervisor may also load pillars. More details about the pillars and the FIMCE manager are presented in Section 4.4.

At the end of the boot-up, the hypervisor prepares the security task's execution. If there are input parameters, it marshals them onto the FIMCE's stack and sets up the new secure stack pointer. Finally, the hypervisor passes the control to the FIMCE manager which starts execution within the isolated environment.

4.2.2 Runtime

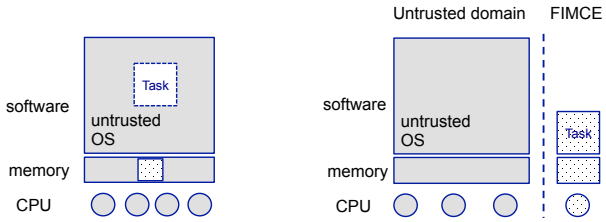
Once the FIMCE manager takes control, the FIMCE is in the running state. It can only be interrupted by hardware events, software exceptions and non-maskable interrupts. All software exceptions and critical hardware interrupts are considered as system failure and trigger the hypervisor to terminate the FIMCE. Other interrupts are handled by the empty handlers by default which simply return, unless their corresponding handlers are installed during FIMCE launching (as part of a pillar).

4.2.3 Termination

A FIMCE can be shutdown due to the security task's termination hypercall or due to the system failure interrupts. To turn off a FIMCE, the hypervisor zeros its registers, memory partition and invalidates the TLBs. It then switches the current VMCS to the one used for the guest OS, re-enables the EPT on the current core. It undoes any device assignment and returns them to the OS. Lastly it notifies the OS that the core is returned.

4.2.4 Comparisons to Memory Isolation Primitive

Figure 4.2 depicts the architectural difference between the memory isolation primitive used in existing schemes and the full isolation of FIMCE. The main distinction is the boundary between the trusted and the untrusted.



(a) Memory isolation primtive

(b) Full isolation of FIMCE

Figure 4.2: The comparison between the memory isolation primitive and FIMCE. The gray regions denote resources controlled by the adversary and the dotted regions denote isolated resources.

As compared with domain isolation [46] and memory isolation, FIMCE possesses their virtues without their drawbacks. It has the well-defined isolation boundary between the trusted and the untrusted, as in domain isolation. Nonetheless, it does not have a large Trusted Computing Base (TCB) as in Terra [46]. FIMCE also achieves page-level granularity as in a memory isolation scheme, e.g., TrustVisor [68]. It reuses the existing OS facility to load the security task, i.e. reading the binary from disk and constructing the virtual address space. However, once the security task is loaded, the guest OS is completely deprived of the capability to interfere with its execution. In contrast, existing memory isolation schemes still allow the guest OS to do so. For example, the guest OS can extract sensitive information via page swapping, as demonstrated in [100]. Furthermore, on a multicore system, the guest OS's task scheduling makes it difficult for the hypervisor to track and enforce access control policies as shown in Chapter 3. The design of FIMCE also avoids the address space layout verification used in TrustVisor [68] and InkTag [51] which is vulnerable to race condition attacks.

4.3 FIMCE and SGX

Although virtualization-based memory isolation systems suffer from the aforementioned security pitfalls, hardware-based techniques do not. Intel SGX provides a hardware-based isolation environment for user-space programs. However, it re-

mains as a challenging problem to use memory isolation techniques alone to protect sensitive I/O tasks, e.g., reading a password from the keyboard. Existing systems like Haven [20] rely on cryptographic techniques with a high performance toll. In the following, SGX and FIMCE are compared from various aspects and then explore potential integration.

4.3.1 Comparisons

FIMCE and SGX are designed with different goals. SGX offers the memory isolation service for applications to protect their sensitive data. Nonetheless, it still requires the OS to manage platform resources, including the enclaves and the associated Enclave Page Cache (EPC) pages. FIMCE is geared to isolate a complete computing environment and therefore covers all hardware and software resources needed by the protected task. The wider coverage allows FIMCE to offer unique advantages over SGX in several aspects.

Memory Isolation SGX and FIMCE provide different strengths of memory isolation. SGX's isolation is tightly integrated with hardware. The TCB only consists of the underlying hardware which is the SGX-enabled CPU and the firmware. The system software is considered as untrusted in SGX's model. The EPC-related memory access check is performed after address translation and before physical memory access [31]. When the processor is not in enclave mode, no system software, including the System Management Mode (SMM) code, can access the content inside an enclave. Data is also encrypted by the hardware when a cache line is evicted to EPC pages. It is hence secure against bus snooping attacks. FIMCE isolation leverages hardware-assisted virtualization techniques. Besides the underlying hardware and firmware, the TCB of FIMCE also encloses the micro-hypervisor which has several thousands SLOC. No encryption is applied when the isolated code writes to the FIMCE memory. Therefore, FIMCE is strictly weaker than SGX in terms of blocking illicit memory accesses.

When the code inside the SGX enclave accesses non-EPC pages, SGX provides *no* security assurance at all. The malicious OS may proactively present a faked memory view to the enclave by using manipulated page tables. In contrast, the code inside the FIMCE accesses all memory pages without obstruction from the OS as the paging structures in use are entirely beyond the OS's control. FIMCE offers stronger security from this perspective.

Autonomous Execution SGX and FIMCE give different treatments of CPU scheduling of the isolated program. SGX leaves thread scheduling inside the enclave to the OS. This design decision allows for the page fault attack [100] wherein the guest OS interrupts the enclave execution and exfiltrates sensitive data.

FIMCE isolates a physical core and exposes a smaller attack surface to the OS. The guest OS cannot influence the execution of the FIMCE thread. All exceptions, including page fault, inside the FIMCE are handled internally. It is hence a highly autonomous system with no runtime dependency on the OS.

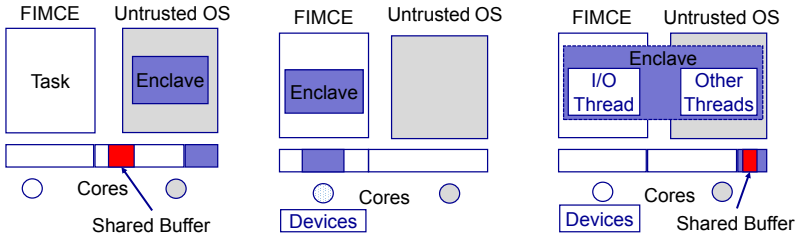
I/O Capability SGX does not support I/O operations. Therefore, secure data exchange between an enclave and the outside world becomes a challenging task. As shown by Haven [20] and SCONE [16], a middle layer between the isolated application and the OS is introduced to ensure data confidentiality and authenticity via cryptographic means. This approach entails significant performance overhead, due to the costly context switches to/from the enclave as well as the cryptographic operations. According to SCONE [16], the I/O intensive benchmarks such as Apache and Redis suffer 21% and 31% performance loss, respectively. Similar results are also reported in Haven.

FIMCE provides inherent I/O support for the isolated tasks. I/O device isolation guarantees exclusive accesses to peripheral devices. Although whitelisted device drivers are still needed, the OS layer abstraction such as sockets or filesystems could be simplified. Furthermore, since the device is isolated and accessed exclusively, cryptographic protection is no longer needed. Note that context switch costs are also saved as no context switch is required inside the FIMCE.

Verifiable Launch SGX and FIMCE also differ in controlling the launch of the isolated environment. Although SGX supports attestation, it does not enforce policies for enclave launch. The permission to launch an enclave is decided by the untrusted OS. In FIMCE's design, the trusted hypervisor can verify the environment to be launched against the security policies (if any). For instance, the platform administrator may supply a signed whitelist to the hypervisor to specify permissible tasks.

4.3.2 Integration with SGX

SGX and FIMCE are not mutually exclusive to each other. It is possible to integrate both so that one's strength complements the other's weakness. In the following, possible ways to combine them are presented.



(a) FIMCE and SGX enclave side-by-side

(b) SGX enclave inside FIMCE

(c) Multithreaded SGX enclave, one thread inside I/O capable FIMCE

Figure 4.3: FIMCE based isolated I/O for SGX enclaves

Share Memory Between Enclave and FIMCE A naive integration is to run the SGX enclave and the FIMCE side-by-side (as in Figure 4.3(a)) with a shared memory buffer used for intercommunication. Unfortunately, it is hard to secure FIMCE-enclave data exchange in this fashion.

The shared buffer has to be on a non-EPC page. Otherwise, the task isolated by the FIMCE cannot access it even with the hypervisor's assistance. Since the shared buffer is outside of the enclave, SGX provides no assurance on the security of accessing them. Although the hypervisor is trusted under the FIMCE model,

protecting the shared buffer using virtualization faces the same issues elaborated in Chapter 3. Hence, it is not a promising approach to integrate SGX and FIMCE as two separated environments in parallel.

Enclave Inside FIMCE Since loading a FIMCE within an enclave is infeasible, the next plausible approach is to launch an enclave inside a FIMCE (as in Figure 4.3(b)). The security benefits of the combination are twofold. As compared to SGX isolation alone, the adversary has no more control over the FIMCE/enclave core. Hence, the composite isolation eliminates those SGX side-channel attacks that require interleaved executions on the core. As compared to FIMCE isolation, the composite isolation is not subject to bus snooping attacks by virtue of SGX protection.

To support SGX enclaves inside the FIMCE, the hypervisor priorly reserves a pool of EPC pages for FIMCE usage. This can be achieved by using normal EPT mappings during boot up, as already implemented in KVM [1]. Running with Ring 0 privilege, the FIMCE manager takes up the kernel's responsibility to set up and manage the enclave.

Specifically, the FIMCE environment is created and launched as described before, except that the protected task and pillars are loaded with Ring 3 privilege. To create the enclave, the FIMCE manager executes SGX special instructions such as `EINIT`. It then assigns EPC pages from the reserved pool to the enclave, and issues `EADD` to add those needed FIMCE pages. After setting up the enclave, the manager passes the control to the protected task which then issues `EENTER` to enter into the enclave.

Multithreaded I/O A direct benefit of the design in Figure 4.3(b) is that the code in the enclave can securely interact with I/O devices via the FIMCE environment. The design can be further extended to support multithreaded I/O operations as depicted in Figure 4.3(c).

The idea is to leverage SGX multithread support. The main thread protected

[1] KVM patch note at https://www.spinics.net/lists/kvm/msg149534.html

by SGX spawns one special thread dedicated for I/O operations. After the enclave is created, the hypervisor migrates the I/O thread into the FIMCE following the design in Figure 4.3(b). The FIMCE manager runs the `EENTER` instruction with the supplied Thread Control Structure (TCS) that uniquely identifies the I/O thread. Hence, the thread running on the FIMCE core and the main thread on the original SGX core belong to the same enclave. The threads intercommunicate through a shared buffer on the allocated EPC pages. The hardware ensures that only those threads belong to the same enclave can access the buffer.

When the main thread needs to access the device, it places a request in the shared buffer. The request is served by the I/O thread inside the FIMCE whereby the FIMCE's device pillars which then perform the desired I/O operations. Similarly, incoming I/O data can be securely forwarded back to the main thread.

The OS may create a fake FIMCE environment when the I/O thread's enclave is created. To detect the attack, the main thread inside the enclave can request the I/O thread to perform an attestation about the underlying environment. Note that after the I/O thread is migrated into the FIMCE, it is no longer under the OS scheduling.

4.4 Modularized Software Infrastructure

It is widely recognized that existing schemes require the protected task to be self-contained, such as in TrustVisor [68] . In contrast, FIMCE has the inborn support for dynamically setting up the software infrastructure e.g., libraries, drivers and interrupt handlers, to cater to the task's needs. It's better to use a structured way to construct the FIMCE software infrastructure. Based on their functionalities, a set of software modules called *pillars* are stored in the disk in the form of Executable and Linkable Format (ELF) files. A pillar is a self-contained shared library for a particular purpose. For instance, a TPM pillar consists of all functions needed to operate the TPM chip. Based on the protected task's demand, the guest OS loads the needed pillar files from the disk to the memory. Then, the hypervisor relocates

them into the FIMCE after integrity checking. The main challenges here are how to ensure the integrity of pillar and how to check correctness of linking without significantly increasing the hypervisor's code size.

4.4.1 Pillars

Pillars provide services that are otherwise not easily available to the security task due to the absence of OS in the FIMCE. Each pillar is assigned with a globally unique 32-bit *pillar identifier* (PLID). Each function that a pillar exports is assigned with a locally unique 32-bit *interface identifier* (IID). Therefore, a (PLID, IID) pair uniquely identify a function in the whole system.

Pillars resemble legacy shared libraries to a large extent. They are compiled as position-independent code because the actual position of a pillar in memory is not determined until it is loaded. They reside in disks as files in the ELF format (for Linux). A pillar differs from a shared library in two aspects. Firstly, a pillar must be self-contained. The code of a pillar function does not depend on any code outside of the pillar. Secondly, the ELF format of a pillar has a new section called pillar descriptor (`.p_desc`) which contains the pillar's PLID, the description of exposed interfaces, and a signature from a Trusted Third Party (TTP).

Pillar Signing The signature included in a pillars' binary image can be provided by a third party who provides the signing service. The service provider computes the signature for binaries submitted to it for signing and embed the signature in the aforementioned section in the returned binaries. The service provider also makes available any public keys that are needed to verify the signatures. The system administrators configure the system so that the keys are loaded into the hypervisor, which can be done by directly embedding the key into the binary image of the hypervisor. This way allows the boot time integrity of the keys to be verified by TPM as part of the hypervisor image. Therefore, at runtime the hypervisor can verify the signatures in the pillars.

Pillar Loading To reduce the hypervisor's workload, the application hosting the security task requests the guest kernel to load the needed pillars into the guest memory as regular shared libraries. When the application issues a hypercall to request FIMCE protection for its security task, the parameters passed to the hypervisor include the needed pillars' PLIDs and their memory layouts. Given the PLID, the hypervisor locates the corresponding pillar in the guest memory, copies it into the FIMCE memory and maps the pillar pages at the same virtual addresses as in the guest. By relying on the guest to manage the pillars in the memory, the hypervisor does not need to support filesystems or disk operations.

4.4.2 Pillar Verification and Linking

Once the needed pillars and the security task are loaded into the FIMCE, the hypervisor passes the control of the core to the FIMCE manager code. If the manager code verifies pillar integrity successfully, it links the security task to the pillars and passes the control to the entry point of the security task.

Pillar verification

Integrity verification is not as straightforward as it seems because it is infeasible to verify the pillar as one whole chunk. The shared library loading procedure may zero some sections and the kernel also performs dynamic linking and running of the library initialization code as well. These operations result in discrepancies between the pillar's memory image and its file in the disk.

Nonetheless, shared libraries are compiled into position independent code that is expected to remain unaltered throughout the loading process. Therefore, the authentication tag of a pillar is a TTP signature protecting the following invariant sections: code sections, data sections, library initialization sections, finish sections, the pillar descriptor section and the section header table of the pillar ELF file. If the verification fails, the manager issues a hypercall to terminate the FIMCE and an error is

returned to the application.

Global symbol references within a pillar are also subject to attacks by the guest kernel. Such references are completed with the assistance of the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) inside the module itself. The entries are typically filled by the loader in the guest. In order to thwart possible attacks, the dynamic symbol table, relocation entries and the string table of the pillar are also signed by the TTP. During loading, the dynamic loader are explicitly requested to resolve all symbols, so that all GOT entries are filled. After the pillar is loaded into the FIMCE, these values are then verified by the FIMCE manager against the corresponding relocation entry to ensure that they refer to the correct locations.

Dynamic Linking

Although the kernel has the capability of linking pillars with the security task, FIMCE can hardly benefit from the kernel's assistance because of potential attacks. A function call to another object file is normally compiled to a call to an entry in the PLT in the originating object file. Because the hypervisor lacks sufficient semantics to determine which entries in the PLT are genuine ones used by the security task, it is costly for the hypervisor to bridge the gap.

FIMCE devise a novel lightweight scheme to link the security task with pillar functions at runtime within the FIMCE. The idea is to introduce a *resolver* function and a *jump table* as part of the FIMCE manager. Both are placed at fixed locations in the FIMCE address space by the hypervisor during FIMCE setup. After verifying all pillars loaded in the FIMCE, the manager parses their descriptor sections and fills the jump table with entries corresponding to each available interface. A jump table is in essence a sorted array of (PLID, IID, entry-address).

A function call from the security task to a pillar function is replaced by a call via a function pointer which takes the original input parameters as well as a pair of (PLID,IID) with a parameter counter. At runtime, the function pointer is assigned with the resolver function's address. After being called, the resolver rearranges the

stack according to the parameter counter, looks up the jump table to find the entry address of the callee function, restores the stack frame used before the call and uses an unconditional jump to redirect execution to the entry. The callee function executes as if it were called directly by the security task, and returns to the security task after execution.

4.5 Applications of FIMCE

Besides isolation, FIMCE can be applied to other types of applications. In the following, two new types of applications are presented. The first application taps into FIMCE malleability to protect a program's long term secret. The second one establishes a runtime trust anchor by exploring the parallelism between a FIMCE and the guest kernel.

4.5.1 Malleability

The FIMCE environment can be configured in a non-standard fashion because its hardware setting is prepared by the hypervisor for the isolated task's exclusively use. For instance, the hypervisor can twist the CPU registers and even the TPM configuration.

To demonstrate the benefit of malleability, let us consider the challenge of ensuring that an application P's long term secret k can only be accessed in its isolated environment. Suppose that k has been initially encrypted with the binding to the isolated environment. The difficulty lies in how to authenticate the thread that requests to enter into the isolated environment. Note TPM's sealed storage *alone* cannot directly solve this problem. Sealed storage is a mechanism to bind a secret to a set of Platform Configuration Registers (PCRs) on the TPM chip. Since most PCRs can be extended by software, the PCR values are dependent on the software that extends them. Therefore, without a proper access control on the PCRs, PCR values do not truly reflect the environment states. Under the adversary model, the

default locality-based access control is not adequate.

Since the application cannot hide any secret in the unprotected memory against the OS, both have the equal knowledge and capability in terms of presenting the authentication information to the hardware. One may suggest leveraging the hypervisor to perform authentication as shown in [68]. However, as analyzed in Chapter 3, it is challenging for the hypervisor to securely authenticate the application without sufficient knowledge about the kernel level semantics.

With a malleable environment, FIMCE offers an elegant solution. The hypervisor uses the TPM Locality 2 and assigns the OS with Locality 0 and the code inside a FIMCE with Locality 1. During boot up, the DRTM extends `PCR17` and `PCR18` with the hypervisor and other loaded modules. When a FIMCE is launched, the hypervisor resets `PCR20` and extends `PCR20` with all code and data loaded in the FIMCE. The protected code in turn extends it with all relevant data, and seals the secret k with `PCR17, PCR18` and `PCR20`. Once the seal operation is done, it extends `PCR20` with an arbitrary binary string to obliterate `PCR20` content and relinquishes its Locality-1 access so that the OS is free to use the TPM. The same steps are performed in order to unseal k.

Note that `PCR17` and `PCR18` are in Locality 4 and 3 respectively. The hardware ensures that they cannot be reset by any software. During the boot up, the DRTM extends these two registers with the loaded modules. Their correct content implies the loading time integrity of the hypervisor. Since the OS is in Locality 0, it does not have the privilege to extend or reset `PCR20`, even though it can prepare the same input used by the hypervisor and application P. Other (malicious) applications in their own FIMCEs cannot impersonate P either. `PCR20` stores the birthmark of a FIMCE instance because the code in a FIMCE cannot reset `PCR20`. Therefore, other applications cannot remove their own birthmarks to produce the same digest as P does.

The advantage of the method is that the hypervisor does not hold any secret and is oblivious to the application's logic and semantics. Besides the stronger security

bolstered by the hardware, it has a small TCB and supports process migration.

4.5.2 Runtime Trust Anchor

Another noticeable strength of FIMCE is its ability to provide an isolated environment that securely runs in parallel with the untrusted OS yet without suffering from the semantic gap. The environment can host a trust anchor to tackle runtime security issues such as monitoring and policy enforcement. To show the benefit of a runtime trust anchor, two systems are sketched below.

The first system is to prevent sensitive disk files from being modified or deleted by the untrusted OS. This problem has been studied by Lockdown [94] and Guardian [27]. Lockdown suffers from performance loss as every disk I/O operation entails a VM exit (if not optimized) while the approach used in Guardian cannot be applied for arbitrary files chosen by applications. In the FIMCE approach, a FIMCE is used as the disk I/O checkpoint. The hypervisor isolates the disk to the FIMCE. A disk I/O filter is loaded in the FIMCE. It continuously reads from a share buffer the disk-related Direct Memory Access (DMA) requests placed by the OS. If the request is compliant with the security policy, the filter forwards it to the disk controller. Otherwise, it drops the request. All disk interrupts are channeled to the OS so that the filter is not necessarily involved in handling them. In this design, the filter is isolated from the guest and the DMA requests are always checked without the cost for VM exit and entry.

The second system is about the runtime attestation of the OS behavior. Most existing remote attestation schemes [79, 54, 68] focus on loading time integrity check. It is challenging to realize runtime attestation because it requires the attestation agent to run securely inside the attesting platform managed by an untrusted OS. With FIMCE protection, the agent runs like an isolated kernel thread side-by-side with the OS. The attestation agent can read the kernel objects without facing the challenging semantic gap problem [87, 40, 52, 43]. To support kernel memory

read, the entire kernel page table is copied into the FIMCE. The hypervisor properly configures the EPTs such that only the agent code pages are executable in order to prevent untrusted kernel code from executing inside the FIMCE.

It is difficult, if not impossible, to solve the attestation problem using the existing memory isolation primitive. If the untrusted guest OS still manages the CPU cores, it can schedule off the attestation agent from the CPU before its attacks and resumes it afterwards. Hence, the attestation does not reflect the genuine state of the kernel.

4.6 Evaluations

4.6.1 Security Analysis

It remains as an open problem to formally prove the security of a system design (not implementation). Therefore, the security analysis below is informal. First, the multicore complications plaguing memory isolation systems are not applicable to the FIMCE design. Then, FIMCE's security is evaluated based on its attack surface and TCB size.

Complication Free Recall that Chapter 3 has enumerated a number of security complications on the multicore platforms. Since the FIMCE is a fully isolated environment, its design does not face these complications. The semantics from the untrusted kernel is controlled and does not affect the enforcement of the isolation policy.

- The EPT management of FIMCE is rather simple. The EPTs used for the OS (and the applications) are not affected by FIMCE. Since the trusted and untrusted execution flows do not interleave with each other on any CPU core, the hypervisor does not need to trace the executions in order to switch EPTs. In addition, the attacks in Section 3.4.3 that exploit stale TLB entries are infeasible. The physical memory of the FIMCE is never accessed by threads outside of the environment. Moreover, when a FIMCE is terminated, the TLB

entries in the core are all flushed out. Hence, there is no stale TLB entry in the system.

- FIMCE does not suffer from the issue of guest page table checking. The execution inside the FIMCE has no dependence on data controlled by the guest OS including the page tables, which makes Iago-like attacks impossible. It is also clear that the full isolation is not subject to the race condition attack described in Section 3.4.

- Existing schemes need to bind subjects and objects to certain states to choose the proper EPT setting. This challenging problem does not exist in the FIMCE scheme. The isolated task is bound to the FIMCE created for it through its whole lifetime. It exclusively accesses the memory. The task may continue the execution without being preempted by other threads under the OS's control. In case that it relinquishes the CPU, its FIMCE hibernates without changing ownership. In other words, all memory states and the CPU context are saved. The CPU states are cleaned up before the OS takes control. When needed, the FIMCE is re-activated from the saved state. Therefore, subject identification is not needed.

- The issue of translating events do not exist in FIMCE. Because the operation of FIMCE is independent from the guest OS. The only events that the hypervisor needs to intervene are the hypercalls from the protected application. These events are well-defined. During the operation of the FIMCE, the hypervisor also does not intervene.

- The enforcement granularity issue does not exist in FIMCE. Although FIMCE still relies on page tables, the memory pages used by a FIMCE are not shared with the kernel. Therefore, there is no page that contains data that needs to be accessed by different privileges.

Reduced Attack Surface The malicious kernel in memory isolation systems

enjoys a large attack surface, as it has the full control over the CPU cores and the VA-to-GPA mapping, which leads to various attacks and design complications described in Chapter 3. In contrast, the attack surface exposed by FIMCE is reduced.

Owing to the full isolation approach, the FIMCE's hardware and software are beyond the kernel's access, interference and manipulation. The kernel cannot access the FIMCE core's registers, L1 and L2 caches. L3 cache is not effectively accessible either because the FIMCE's host physical address region is never mapped to the guest. Although the kernel may use IPI or NMI to interrupt the FIMCE, the worst consequence is equivalent to a DoS attack. Since the isolated code handles the interrupts by itself, an IPI or NMI only results in a detour of the control flow. Note that there is no context switch inside the FIMCE.

Another attack vector widely considered in the literature is the interaction between the hypervisor and the kernel. The FIMCE hypervisor only exports two hypercalls, for setting up and terminating the FIMCE respectively. Moreover, the hypervisor does not interpose on either the guest execution or the FIMCE execution.

The FIMCE may exchange data with threads in the outside environment. In that case, the malicious kernel may poison the input data to the isolated task. It is acknowledged that the protected code is subject to such attacks if no proper input checking is in place. However, it is out of scope of the FIMCE work to sanitize the inputs.

Strength of Full Isolation The FIMCE isolation is enforced by the hardware. The memory used by the FIMCE is from a reserved physical memory region which is never mapped to the guest by the EPT and the IOMMU page table. Both segmentation and paging of the FIMCE memory are constructed by the hypervisor. It is thus obvious that the attacks in Section 3.4.3 do not work against the FIMCE. The guest kernel cannot make modifications to either segmentation- or paging-related data structures to tamper with the address space.

Security of Foreign Code in FIMCE In addition to the security task, the FIMCE hosts the needed pillars and the FIMCE manager. It is crucial to ensure

the integrity of their code and the linking process. The FIMCE manager's code is *copied* from the hypervisor space. Therefore, its integrity is ensured as long as the hypervisor is trusted.

During FIMCE launching, the integrity of the loaded pillars' code images (including the relocation sections) are verified by the FIMCE manager in order to prevent the kernel from poisoning them. To ensure linking integrity, the jump table is constructed using the verified pillar description sections. Addresses in the GOT entries used for symbol references within a pillar are also checked by the FIMCE manager to match the corresponding addresses computed from the pillar's relocation-related sections.

Small TCB Size The hypervisor is the *only* trusted code in the system. With a simple logic, the FIMCE hypervisor has a tiny code base with around six thousand lines of source code for runtime execution. It is easy to manage concurrency in the hypervisor space. Because only the setup and teardown code possibly execute concurrently on different cores, they can be synchronized with simple spinlocks. Note that each FIMCE instance does not have overlapping regions, which also help simplify concurrency handling.

4.6.2 Implementation

A prototype of FIMCE has been implemented on a desktop computer with an Intel Core i7 2600 quad-core processor running at 3.4 GHz, Q67 chipset, 4GB of DDR3 RAM and a TPM chip. The platform runs an Ubuntu 12.04 guest with the stock kernel version 3.2.0-84-generic. The FIMCE hypervisor consists of around 6000 source line of code (SLOC). It exports two hypercalls, i.e., `FIMCE_start()` and `FIMCE_term()`, for starting and terminating a FIMCE, respectively. The TCB of FIMCE only consists of the TXT bootloader, the hypervisor and any hardware and firmware required by DRTM. Intel's open source TXT bootloader *tboot*[2] is slightly

[2]http://sourceforge.net/projects/tboot/

modified to load the FIMCE hypervisor. The modification is to make the bootloader jump to the FIMCE hypervisor's entry at the end of the boot up sequence. During hypervisor initialization, a set of EPT entries are initialized such that a chunk of physical memory is reserved for exclusive use by the hypervisor. During the OS kernel initialization, all cores are set to use the same set of EPTs, ensuring a uniform view of the memory.

To showcase the applications of FIMCE, three pillars are also developed: a 7KB serial port driver pillar that supports keyboard I/O, a comprehensive crypto pillar of 451KB size based on the *mbed* TLS library [3], a TPM driver pillar of 20KB size. The implementation also encloses a FIMCE management code of 413 SLOC which verifies and links pillars in use.

Programming Interface As shown in Table 4.1, the user space FIMCE library provides two interfaces and three macros. The two interface functions are used to start and stop the FIMCE, respectively. They are in essence wrappers of the hypercalls to pass parameters to the hypervisor. The first two macros are for the application developer to specify the function to be protected, as well as the ELF file names of pillars to be loaded. The third macro converts a normal C function call into a pillar function call. (Note that the dynamic linking mechanism in a FIMCE is different from in the OS.)

Table 4.1: User Space Library Interfaces and Macros

Interface	**Description**
start_FIMCE ()	Call to start FIMCE
stop_FIMCE ()	Call to stop FIMCE
Macros	
__FIMCE_SECURITY_TASK ()	Specify function that is the security task
__FIMCE_LOAD_PILLAR ()	Specify the file name of the pillar
__CALL_PIL ()	Call a pillar function in a security task

Internals of `start_FIMCE()` This user space function runs in two steps in the guest domain. Firstly, it helps to load all data and code needed by the hypervisor into the main memory. It finds the address of `FIMCE_PAYLOAD` from the symbol table and the locations of needed pillars, including the base addresses and lengths

[3] https://tls.mbed.org/

of all segments with assistance of the dynamic loader. To ensure that the contents are indeed loaded into memory, it reads all pages of the pillars.

The second step is to gracefully take one CPU core offline from the guest so that the (honest) kernel does not attempt to use the core dedicated for the FIMCE. For this purpose, the function reads the file `/proc/cpuinfo` and gets the physical APIC ID of a randomly chosen CPU core. It then writes a '0' to the corresponding control file named as *online*. In the end, it issues a hypercall to pass to the hypervisor the physical APIC ID, all offsets and lengths of the security task and its pillars.

The issuance of the hypercall traps the *present* core running `start_FIMCE` to the hypervisor, not the offline core chosen for the FIMCE. Therefore, the hypervisor initializes necessary data structures for the FIMCE and needs to take control of the FIMCE core. To achieve this, it sends an INIT signal to the offline core identified by the physical APIC ID and returns from the hypercall so that the present core is returned to the guest OS. The INIT signal is intercepted by the hypervisor on the FIMCE core. The hypervisor then loads the prepared VMCS and performs a VM entry to start the FIMCE.

4.6.3 Benchmarks

Since a FIMCE occupies a CPU core exclusively, the OS has less computation power at its disposal while the FIMCE is running. In order to understand the overall impact of a running FIMCE on the platform, a suite of benchmarks including multithreaded *SPECint_rate 2006* and *kernel-build* as well as single threaded *lmbench*, *postmark* and *netperf* are run. They are first run on top of the OS without any FIMCE running and then repeated with an infinite loop as the security task in the FIMCE.

For the multithreaded *SPECint_rate 2006*, the concurrency level is set to four. Figure 4.4 shows that it has 15% percent performance drop on average due to the presence of FIMCE. In the kernel-build experiment, the Linux kernel v2.6 is com-

piled using the default configuration with four-level concurrency. The results are reported in Table 4.2. The two sets of experiments indicate that the relative performance loss grows with the degree of concurrency, mainly due to more frequent context switches. Nonetheless, the loss is bounded by the inverse of the number of physical cores in the platform (namely 25% in the setting).

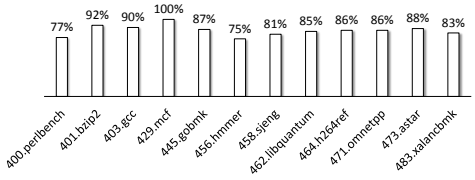


Figure 4.4: SPECint_rate 2006 results. The numbers are the percentage of the score with FIMCE to the score without FIMCE.

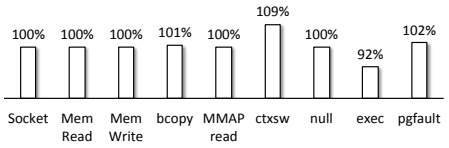


Figure 4.5: Lmbench results. The numbers are the percentage of the score with FIMCE to the score without FIMCE.

To verify the estimation that FIMCE does not incur much performance cost for single-threaded applications, Lmbench, Netperf and Postmark are run with and without FIMCE. Figure 4.5 shows that most tasks of *Lmbench* are not affected by FIMCE, except that one task has 8% performance drop. Similar results are also found for Netperf as in Table 4.3 and Postmark as in Table 4.4.

4.6.4 Component Costs

The major overhead of FIMCE is in its launching phase. The security task's execution inside the FIMCE does not involve the hypervisor and thus incurs no cost as compared to its normal execution. The launching cost consists of three parts: a hypercall (a VM exit and a VM entry), FIMCE setup including resource allocation and environment setup, and code loading.

Table 4.2: Kernel Build Time (in seconds)

Concurrency level	**4**	**6**	**8**	**12**
W/O FIMCE	783	708	640	643
With FIMCE	900	828	797	803
Performance Loss (%)	15	17	24	24

Table 4.3: Netperf Bandwidth With And Without FIMCE Running (in Mbps)

	TCP_Stream	**UDP_Stream**
W/O FIMCE	93.92	95.99
With FIMCE	93.95	95.95
Performance Loss (%)	0.03	0

Table 4.4: Single-threaded Postmark Performance with and without FIMCE Running (in seconds)

W/O FIMCE	327
With FIMCE	330
Performance Loss (%)	1

Table 4.5: Loading Time for Pillars with Various Sizes

Size (KB)	7	11	15	19	23	27	31	35
Time (μs)	56	58	61	63	63	65	66	68

On average, a null hypercall on the experiment platform takes 0.31 millisecond. FIMCE setup takes about 47.33 milliseconds which is the interval between the `FIMCE_start` hypercall to the `INIT` signal prior to the start of FIMCE execution. The code loading time depends on the total binary size of the loaded pillars and the security task. Table 4.5 shows the time needed to copy a chunk of bytes from the guest to the FIMCE, including preparing the mapping and memory read/write. On the experiment platform, every 4KB memory read and write cost about 2μs.

Pillar loading also involves integrity verification. The measurement shows that it takes about 40.3 microseconds to verify one RSA signature inside the FIMCE. Therefore, the total cost of launching the FIMCE (mostly depending on the number of public key signatures to verify) is in the range of 100 milliseconds to a few seconds. There are several ways to save this one-time cost. For instance, a pillar's integrity can be protected by using HMAC whose verification is several orders of magnitude faster than signature verification. Another is for the hypervisor to cache some frequently used pillars which are used without integrity check during FIMCE launching.

4.6.5 Application Evaluation

Four use cases are implemented to demonstrate the power of FIMCE. The use cases include password based decryption, an Apache server performing online RSA decryption, long term secret protection and runtime kernel state attestation.

Password based decryption It is challenging to protect tasks with I/O operations using memory isolation, mainly because I/O operations are normally in the kernel which has a large and dispersed code base and are interactive with devices.

Driverguard [28] relies on manual driver code instrumentation, which is tedious and error-prone. TrustPath [104] relocates the entire driver into the isolated user space, which not only requires significant changes in the user space code, but also burdens the hypervisor with complex functions. As a result, there are a lot of hypercalls when issuing I/O commands and handling interrupts, which incurs a heavy performance loss because of frequent expensive VM exits.

FIMCE offers a more efficient solution. The code running inside the FIMCE is in Ring 0 and is capable of handling interrupts. Furthermore, with hardware virtualization, the hypervisor can channel the peripheral device interrupts to the FIMCE core for the isolated task to process. Therefore, a device's I/O can be conveniently supported as long as its driver pillar is loaded into the FIMCE.

A program is implemented that reads the user's password inputs from the keyboard, converts it to the decryption key and then performs an AES decryption. When the FIMCE is launched to protect this program, the hypervisor isolates the keyboard by intercepting the guest's port I/O accesses. A serial port pillar and the crypto pillar are loaded into the FIMCE. The program is run with FIMCE protection for 100 times. In average, it takes 0.94 milliseconds to decrypt the ciphertext of 1kilobytes, which is only 5.2% slower than in the guest.

Apache Server In this case study, FIMCE is utilized to harden an SSL web server by isolating its RSA decryption of Secure Sockets Layer (SSL) handshakes. As noted previously, existing systems [68, 26] on Apache protection is not secure under the multicore setting. Moreover, since the isolated code runs in the *same* thread as its caller, it incurs frequent VM exits and VM entries as the control flow enters and leaves the isolated environment. FIMCE does not incur context switches at runtime because the isolated task in a FIMCE runs as a separated thread in parallel with others.

In the experiment, the Apache source code is customized such that its SSL handshake decryption function is protected by a FIMCE. Apache runs in *prefork* mode with eight worker processes. Each worker process forwards incoming requests to

the decryption function inside the FIMCE and subsequently fetches the decrypted master secrets.

The server is connected to a LAN and ApacheBench is run with different concurrency levels. The Apache server hosts an HTML page of 500KB. This setting is compared with the same experiment without using FIMCE protection whereby all worker processes are able to perform the decryption concurrently. The results are shown in Table 4.6.

Table 4.6: Modified Apache Performance, # of SSL Handshakes per Second

Concurrency Level	1	2	4	32	128	256
W/O FIMCE	7.39	13.96	20.21	26.95	27.88	29.69
With FIMCE	7.31	14.04	20.09	20.21	21.09	22.23
Overhead (%)	1	0	0.5	25.0	24.4	25

It is evident that at low concurrency level of up to four, the FIMCE-enabled Apache server performs almost equally well as the native multithreaded Apache. It outperforms existing schemes listed in Table 4.7 due to the fact that FIMCE does not involve costly context switches. However, its performance drops as the concurrency level increases, but is bounded by 25%. This is because the single-threaded FIMCE cannot match the performance of a multithreaded Apache which can use all four cores to perform concurrent decryption. The performances of TrustVisor [68], InkTag [51] and Overshadow [26] are not affected by concurrency, albeit they are *not* secure in a multicore system.

Table 4.7: Overhead Of Other Protection Schemes (numbers are excerpted from respective paper)

Schemes	**Overhead**
TrustVisor [68]	9.7% to 11.9% depending on concurrent transaction
InkTag [51]	2% in throughput, 100 concurrent request
Overshadow [26]	20% to 50% on a 1Gbps link, 50 concurrent request

However, the design of FIMCE can certainly be extended to support concurrent FIMCE instances, at the expense of more cores dedicated for security. Also, in real world web transactions, the time spent for RSA decryption accounts for a much smaller portion of the entire transactions as compared to in the benchmark testing, because of (1) longer network delays in the Internet; (2) more SSL sessions using

the same master key decrypted from one SSL handshake; (3) more time needed to generate or locate the need web pages. Therefore, the performance loss of using FIMCE for a real web server does not appear as discouraging as in the experiments.

Long Term Secret Protection The malleability of FIMCE architecture is demonstrated via the long term secret protection application. A program is implemented to bind a long term secret to the FIMCE instance. The program and the TPM pillar are loaded into the FIMCE. During FIMCE launching, the hypervisor extends `PCR 20` which bear the birthmark of this FIMCE instance. The program seals and unseals a long term secret to `PCR17, PCR18` and `PCR20`. the time taken by the TPM seal and unseal operations inside the FIMCE is measured and compared with the baseline experiment wherein the TPM pillar runs as a kernel module. The results of protecting a 20-byte long secret are shown Table 4.8.

Table 4.8: TPM Performance (in seconds)

	TPM Seal	**TPM Unseal**
Guest	0.54	0.96
FIMCE	0.41	0.94

FIMCE shows slight speed up compared to the performance inside the guest. One of the contributing factors is that there is more kernel code involved when running the TPM operations inside the guest. CPU scheduling is another possible factor affecting the performance because the entire operation is rather lengthy. In contrast, due to the simple structure, FIMCE does not have such overhead.

Compared with existing approaches that virtualize the TPM using software such as [68], the FIMCE approach places the trust directly on the hardware TPM chip. In contrast, virtualizing TPM requires the code that virtualizes the TPM to be included in the trust chain. The architecture of FIMCE allows us to multiplex accesses to the TPM chip with a smaller attack surface and the smaller TCB.

Runtime Kernel Introspection with Attestation A program for kernel introspection is implemented in this case. Running as a kernel thread isolated in the FIMCE, the program reads the `mm_struct` member of the `init_task` structure used by the guest kernel. It takes about 3.04 μs to read a kernel object which is

comparable with the time (around 3.11 μs) needed by the kernel itself. This introspection system is more efficient than existing schemes like [43] because it runs natively on the hardware in the same fashion as running inside the kernel. According to the experiment, the speed of native instruction execution with MMU translating a virtual address is about 300 times faster than using software to walk the page table. This prototype is further extended to be the ImEE system in Chapter 5.

The introspection results can be attested by the FIMCE system to a remote verifier. As there is a chain of trust established during FIMCE launching, it is convenient to use the code inside the FIMCE to do runtime attestation. The root of the trust chain is Intel's TXT facility. When the hypervisor is loaded, the hardware measures its integrity before launching. The hypervisor then measures all code during FIMCE launching. At runtime, the code inside the FIMCE measures the kernel's states. The measurements are stored in various PCRs depending on the assigned localities. Note that one of the challenges of existing TPM-based attestation schemes is to have a reliable attestation agent which (ideally) is immune from attacks of the attested objects, and at the same time, nimble enough to dynamically perform measurements whenever needed. FIMCE exactly offers such a solution.

In this implementation, the introspection code inside the FIMCE uses the crypto pillar to sign the introspection results with a TPM quote for `PCR 17`, `18` and `20` which vouches for the FIMCE environment. The entire process runs in parallel with the guest OS. It takes 3.47 seconds in average to perform the entire procedure, including the time for TPM quote operation.

4.7 Discussion and Future Directions

With the core number steadily increasing, the seemingly significant 25imposed by the FIMCE architecture is likely to wane. For example, some of the latest generation desktop CPU at the time of writing has already been equipped with eight cores, thereby reducing the performance drop to be bounded by 12.5to continue to grow

in the future, which makes the FIMCE architecture increasingly relevant.

To minimize the performance impact, the FIMCE environment can safely sleep when not needed. The isolation on the physical core can be temporarily torn down while the one on the memory and devices is still in effect. The core is then returned to the guest OS for its own purpose. This kind of temporary switch of the identity binding between the hardware thread and the high-level entities does not undermine security because the isolation policy about the resources, i.e. time slices, memory and devices, is clearcut due to the full isolation approach.

Since the switching between a non-FIMCE environment and FIMCE on a physical core still imposes some overhead, it remains open to evaluate such overhead and design a secure and efficient on-demand activation scheme of the FIMCE environment. It is likely that such a scheme would employ certain batch processing technique to amortize the cost over a number of service requests, while minimizing the observed delay by the applications that request for the service. The scheme should also take into account the fact that in the on-demand design, the scheduling policy is left to the untrusted guest kernel to enforce, therefore, it may simply deny FIMCE's execution by permanently holding on to the cores.

Lastly, it remains open to evaluate the applicability of FIMCE on other architectures such as ARM platforms equipped with virtualization features. ARM processors are also equipped with more and more cores, which makes them a suitable ground for the FIMCE approach.

Chapter 5

Consistent Virtual Machine Introspection

The Immersive Execution Environment (ImEE) is presented in this chapter. The ImEE builds on top of the fully isolated environment, while incorporating special configurations for a consistent memory view with the target VM. ImEE directly reuses the kernel semantics in the kernel page tables for consistent introspection of any high-level objects. In this chapter, firstly, the inadequacy of existing software-based approach is discussed to motivate the design. Next, the details of ImEE are present in the following sections.

5.1 The Inference Gap in Software-based Guest Access

It is a common practice in the VMI literature to use software to emulate virtual address translation before accessing a target guest VM. This approach to bridge the inference gap results in an inconsistent memory between the introspection tool and the guest VM.

In the software-based approach, the target memory is mapped to the monitor VM as a set of read-only pages. In the most general case, given a virtual address X,

the introspection code walks through all levels of the paging structures, including the EPTs in the memory to find out the corresponding HPA. It then maps the HPA to its own virtual address space before finally reads it. Obviously, such a procedure incurs a much longer latency than the native access to X in the guest.

To assess how slow the software-based guest access is relative to the native access, a "cat-and-mouse" experiment is run. The introspection program based-on LibVMI [72], a cross-platform VMI framework, keeps reading a guest process's `task->cred` pointer, while a guest kernel thread periodically modifies the pointer and waits for 20,000 CPU cycles before restoring its value. The page-level data cache of LibVMI is disabled to ensure the freshness of every read whereas the translation caches are on since no address mapping is modified. The experiment was run for eight times, each lasting 10 seconds. In average, the modification was only spotted after being repeated 60 rounds. In one of the eight times, no modification was caught. The experiment result demonstrates that introspection at low speed cannot catch up with the fast-running attacker. It is ill-suited for scenarios demanding quick responses such as live forensics and real-time I/O monitoring.

The slow speed also affects the mapping consistency as the guest malware in the kernel may make transient changes to the page tables, rather than the data. Since walking the paging structures appears instant to the malware using the MMU, but not to the introspection software, the malware's attack on the page tables causes the VMI tool to use inconsistent information obtained from the paging structures.

Caching techniques have been used in order to reduce the latency of guest accesses. For instance, LibVMI introduces three types of caches: the page-level data cache, the VA-to-HPA translation cache and the `pid` to `CR3` cache. While promoting the performance, using the caches is detrimental to effective introspection. Since the guest continuously runs during the introspection, any cached mapping or data is not guaranteed to be consistent with the one in the memory. Moreover, it is difficult for the software-based approach with caches to catch up with the pace of `CR3` updates in the guest. Since the guest kernel is untrusted, the introspection

cannot presume that all guest threads share the same kernel address space. `CR3` synchronization with the guest may lead to cache thrashing which backfires on the introspection performance.

Besides the security related limitations described above, the software method has performance-related drawbacks. It usually has a bulky code base since it has to fully emulate the MMU's behavior, such as supporting 32-bit and 64-bit paging structures as well as different modes and page sizes. Its operation leaves a large memory footprint because of the intensive reliance on data and translation caches. It also suffers from slow-start due to the complex setup. For instance, the LibVMI initialization costs 100 milliseconds according to the measurement. To change the introspection target from one VM to another requires a new setup. With these performance pitfalls, the software-based method is not the best choice for introspection in data centers where the VMI tools may need to scan a large crowd of virtual machines.

5.2 Overview

5.2.1 Basic Idea

The basic idea behind the special computing environment called *Immersive Execution Environment* (ImEE) is to construct a twisted address mapping setting (as in Figure 5.1). The ImEE's `CR3` is synchronized with the target VM's active `CR3` so that its MMU directly uses the target's VA-to-GPA mappings. Its GPA-to-HPA mappings are split into two. The GPAs for the intended introspection are translated with the same mappings as in the target VM; the GPAs for the local usage (indicated by the dotted box in Figure 5.1) are mapped to the local physical pages via separated GPA-to-HPA mappings. With this setting, memory accesses are automatically directed by the MMU into the target and the local memory regions according to the paging structures.

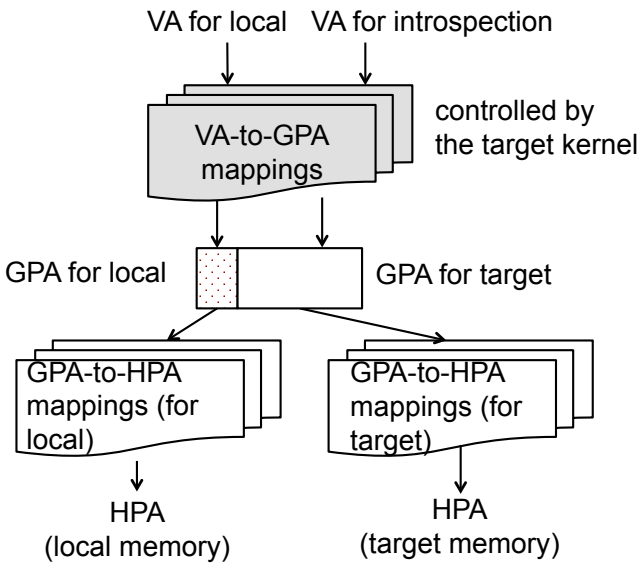


Figure 5.1: Illustration of the idea of direct usage of the target VM's VA-to-GPA mappings and splitting in GPA-to-HPA mappings. Note that the shadow box is fully controlled by the target (i.e., the adversary).

The paging structure setup in the ImEE ensures mapping consistency with the target VM. Firstly, the ImEE's VA-to-GPA mappings remain the same as the target's, because its `CR3` and the target `CR3` always point to the same location. Any mapping modification in the target also takes effect in the ImEE simultaneously. Secondly, the hypervisor ensures that the ImEE GPAs intended for introspection are mapped in the same way as within the target. Hence, any VA for introspection is translated consistently with the target. Note that the VA is accessed at native speed because the MMU performs the address translation.

5.2.2 Challenges

Suppose that the ImEE has been set up following the idea above with an introspection agent running inside and accessing the target memory. The following design challenges need to be addressed in order to achieve a successful introspection.

Functionality Challenge The ImEE agent's virtual address space comprises the executable code, data buffers to read and write, and the target kernel's address space. Since the agent code and data are logically different from the target kernel,

there needs to be a way to properly *split* the GPA domain so that VAs for the local uses are not mapped to the target and VAs for introspection are not mapped to the agent memory.

This challenge to divide the GPA domain is further complicated by two issues. Firstly, the virtual address space layout of the target is not priorly known, because it is entirely dependent on the current thread in the target. Therefore, it is a challenge to device a universal mechanism to load the ImEE agent regardless the target's address space layout. Secondly, read/write operations on the local memory and on the target memory are not distinguishable to the hardware. Therefore, it is difficult to separate access to local pages and target pages. For example, it is difficult to detect whether a VA for introspection is wrongly mapped to the local data (which could be induced by the target kernel inadvertently or willfully) because it does not violate the access permissions on the page table.

Security Challenge The ImEE is not fully isolated from the adversary. The target VM's kernel has the full control of the VA-to-GPA mappings which affect the resulting HPA. Hence, the adversary can manipulate the ImEE agent's control flow and data flow by modifying the mappings at runtime. Although access permissions can be enforced via the GPA-to-HPA translation, the adversary can still redirect the memory reference at one page to another with the same permissions.

A more subtle, yet important issue, is the introspection *blind spot*, namely the set of virtual addresses in the target which are not reachable by the ImEE agent. As shown in Figure 5.2, a VA for introspection is in the blind spot if and only if it is mapped to the GPA for local use. This is because the full address translation ends up with a local page, instead of the target VM's page. The malicious target can turn its pages into the blind spot by manipulating its guest page table. The blind spot issue has two implications. First, detecting its existence efficiently is challenging. Note that it is time-consuming to find out all VAs in the blind spot, because the guest page tables have to be traversed to obtain the GPA corresponding to a suspicious VA. Second, the attacker can manipulate VA to GPA mappings in an

attempt to disrupt the execution of the ImEE agent. By manipulating the mappings, the attacker tries to cause invalid code to be executed inside the environment, or cause the introspection to read arbitrary data.

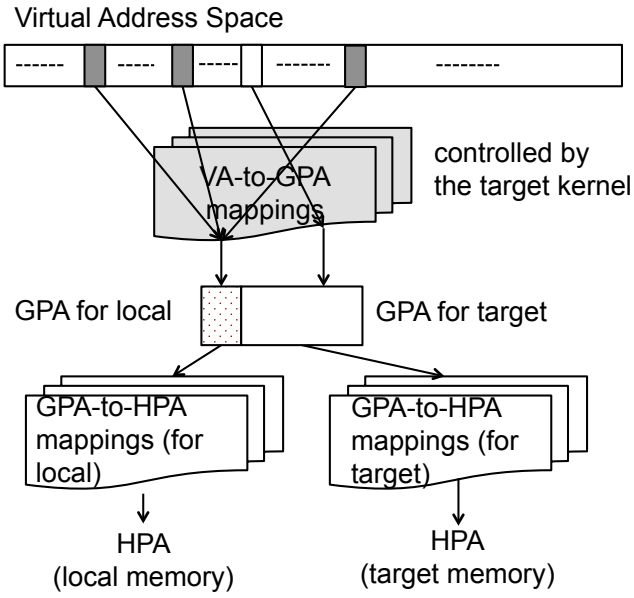


Figure 5.2: Illustration of the blind spot comprising three virtual pages (in the dark color). Target kernel objects in those pages cannot be introspected since they are mapped to the local memory.

Performance Challenge Although the ImEE agent accesses the target memory at native speed, the goal is to minimize the time for setting it up in order to maximize its capability of quickly responding to real-time events and/or adapting to a new introspection target (e.g., another thread in the target VM or even another target VM). The challenge is how to load the agent into the virtual address space currently defined by the target thread and to prepare the corresponding GPA-to-HPA mappings. Searching in the virtual address space is not an option since it is time-consuming to walk the target VM's paging structures. In addition, it is also desirable to minimize the hypervisor's runtime involvement, because the incurred VM exit and VM entry events cost non-negligible CPU time.

Besides the above three major challenges, there are other minor issues related to the runtime event handling, such as page faults and the target VM's EPT updates. The requirement of Out-of-VM introspection is to minimize intrusive effects on the

target. For example, the hypervisor is refrained from modifying the target VM's guest page tables because it leads to execution exceptions in the target. Therefore, the minor issues also need careful treatment.

5.2.3 System Overview

The ImEE is in essence a special virtual machine which is created and terminated by the hypervisor based on the VMI application's request. Like a normal VM, the ImEE hardware consists of a vCPU core and a segment of physical memory, both (de)allocated by the hypervisor when needed. No I/O device is attached to the ImEE. The ImEE does not have an OS and the only software running in it is the ImEE agent which reads the target memory. Figure 5.3 depicts an overview of the whole system.

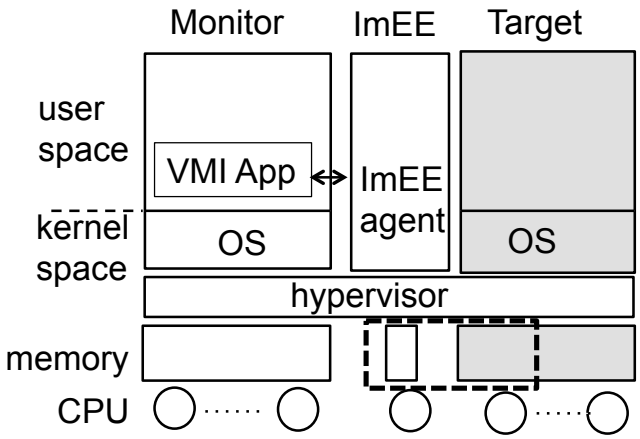


Figure 5.3: Overview of ImEE-based introspection. The box with dashed lines illustrates the mixture of physical memory. The shadowed regions belong to the target and are not trusted.

The VMI application can launch the ImEE, put it into sleep, and terminate it. Like a regular VM, the ImEE can also migrate from one core to another. While the ImEE is active, it runs in *sessions* which is defined as the tenure of its `CR3` content. To kick off a session, the hypervisor either induces a VM exit or intercepting `CR3` changes in the target.

5.3 The Design Details

In this section, first, the internals of the ImEE is presented with the focus on the paging structures, and then the ImEE agent is explained. The design choices for performance are shown where appropriate. Lastly, the lifecycle of ImEE is described, focusing on the runtime issues such as transitions between sessions.

The approach is to carefully concert system design, e.g., setting the ImEE's EPTs and software design (i.e. crafting the agent) so that the ImEE agent execution straddles between two virtual address spaces: one for the local usage and the other for accessing the target VM.

5.3.1 ImEE Internals

The ImEE requires a vCPU core which can be migrated from one core to another. It also comprises one executable code frame and one read/writable data frame. The former stores the agent code while the latter stores the agent's input and output data. To differentiate them from the target VM's physical memory, they are referred to as the *ImEE frames*.

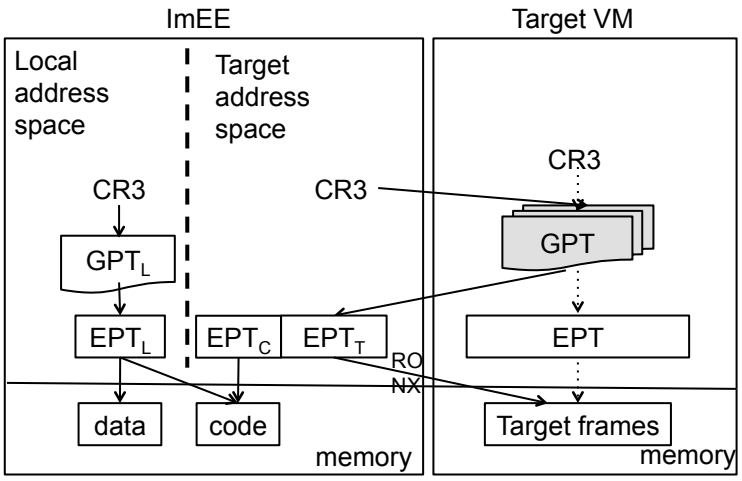


Figure 5.4: The solid arrows describe the translation for a VA within the ImEE, while the dotted arrows describe the translation inside the target. All target frames accessible to the ImEE agent are set as read-only and non-executable in EPT$_T$.

According to the `CR3` content, the agent runs either in the *local address space*

or the *target address space*, as depicted in Figure 5.4. When in the local address space, the agent interacts with the VMI application. While it runs in the target address space, it reads the target memory. The code frame is mapped into both spaces while the data frame is mapped in the local address space only.

Local Address Space The paging structures used in the local address space comprise GPT_L and EPT_L, which map the entire space to the ImEE frames. GPT_L only consists of two pages as shown in Figure 5.5. The global flag on the GPT_L is set so that the local address space mappings in the TLB are not flushed out during `CR3` update. Specifically, only one virtual page is mapped to the data frame while all others are mapped to the code frame. With this setup, the agent code can execute from all but one page. Moreover, the GPAs of the ImEE frames are *not* within the GPA range the target VM uses, which avoids conflict mappings used in the target address space.

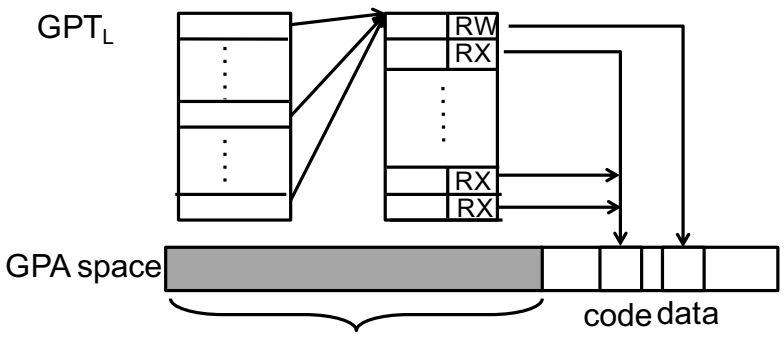


Figure 5.5: The Illustration of GPT_L. All entries in the page table directory point to the same page table page which has one PTE points to the data frame and all other to the code frame.

Target Address Space The target address space implements the idea in Figure 5.1. To run the agent in this space, the ImEE `CR3` register is synchronized with the target `CR3`, so that they use the same guest page tables. The GPA-to-HPA mappings used in this space are governed by EPT_T and EPT_C.

All GPAs are mapped to the target frames by EPT_T, except that one page is redirected by EPT_C to the ImEE code frame. Specifically, EPT_T is populated with the GPA-to-HPA mappings from the target VM's EPT, except that all target frames

are guarded by read-only and non-executable permissions. This stops the agent from modifying the target memory for the sake of non-intrusiveness. It also prevents the adversary from injecting code, because the adversary can place arbitrary binaries to those frames. The permission of the mapping defined by EPT_C is set as *executable-only*. Namely, it cannot be read or written from the target address space.

Note that the ImEE data frame is not mapped in the target address space for two reasons. Firstly, it minimizes the number of GPA pages redirected from the target to the ImEE, and therefore reduces the potential blind spot. Secondly, *all* memory read accesses performed in the target address space are bounded to the target. Therefore, it is feasible to configure the hardware to regulate memory accesses so that any manipulation on the target GPT that attempts to redirect the introspection access to the ImEE memory is caught by a page fault exception.

CAVEAT. Address switches inside the ImEE does not cause any changes on the EPT level. The GPA-to-HPA mappings used in one address space are cached in the ImEE TLBs and are not automatically invalidated during switches. Note that EPT_L, EPT_C and EPT_T do not have conflict mappings because they map different GPA ranges. The two address spaces are assigned with different Process-Context Identifier (PCID) to avoid undesired TLB invalidation on address space switch.

5.3.2 ImEE Agent

The ImEE agent is the only piece of code running inside the ImEE, without the OS or other programs. It is granted with Ring 0 privilege so that it has the privilege to read the target kernel memory and to manage its own system settings, such as updating the `CR3` register. It is self-contained without external dependency and does not incur address space layout changes at runtime in the sense that all the needed memory resources are priorly defined and allocated.

The description below involves many addresses. Table 5.1 defines the notations.

Overview The main logic of the agent is as follows. Initially, the agent runs in

	VA	GPA
ImEE data	P_d	GP_d
ImEE code (local addr. space)	P_c	GP_c
ImEE code (target addr. space)	P_c	GP'_c
Target page	P_t	GP_t

Table 5.1: Address notations. For instance, GP_c is the guest physical address of the ImEE code page in the local address space.

the local address space and reads an introspection request from the data frame. Then it switches to the target address space and reads the targeted memory data from the target memory into the registers. Finally, it switches back to the local address space, dumps the fetched data to the data page and fetches the next request.

The Agent Figure 5.6 presents the pseudo code of the agent. The agent has only one code page and one data page. Since the data frame is out of the target address space, all needed introspection parameters (e.g., the destination VA and the number of bytes to read) are loaded into the general-purpose registers (Line 6). For the same reason, the agent loads the target memory data into the ImEE floating-point registers as a cache (Line 12), before switching to the local address space to write to the data frame (Line 17).

The agent is loaded at P_c in the local address space by the hypervisor. P_c is chosen by the hypervisor such that it is an executable page according to the target's guest page table. Because GPT_L maps the entire VA range (except one page) to the code frame, there is an overwhelming probability that P_c is also an executable page in the local address space[1]. Therefore, the agent can execute in the two address spaces back and forth which explain Line 12 and 17 can run successfully without relocation.

Impact of TLB No matter whether there is an attack or not, TLB retention has no adverse effect on the introspection. Suppose that the mappings in the local address space are cached in the TLB. When the agent runs in the target address space, the only VAs involved are for the instructions (P_c) and the target addresses (P_t). For

[1]In case P_c is not executable under GPT_L, the hypervisor only needs to adjust the corresponding PTE.

```
1:  while TRUE do
2:     /* local address space: Read the request */
3:     repeat
4:        poll the interface lock;
5:     until the lock is off
6:     Read the request from the data frame to general-purpose registers;
7:
8:     /* switch to target address space */
9:     Load the target CR3 provided by the hypervisor;
10:
11:    /* target access */
12:    Move n bytes from the target address x to floating-point registers;
13:
14:    /*switch to local address space */
15:    Load CR3 with GPT_L;
16:    /* output to data frame */
17:    Move data from the floating-point registers to the ImEE data page;
18:    if requested service not completed then
19:       goto Line 9;
20:    end if
21:    Set interface lock;
22: end while
```

Figure 5.6: The sketch of the ImEE agent's pseudo code

VAs in P_c, the cached mapping remains valid because the address mappings are not changed. There are two exclusive cases for P_t. If $P_t \neq P_d$, the translation does not hit any TLB entry because it is never used in the local address space. Otherwise, the TLB entry for P_d is still considered as a miss because of different PCIDs. The same reasoning also applies to the cached mappings in the target address space.

Note that the adversary gains no advantage from a TLB hit on a cached local address space translation. Since EPT_L is available in the target address space, the adversary can manipulate its own page tables to achieve the same outcome as a TLB hit. It can use arbitrary GPA in its page tables.

5.3.3 Defeating Attacks via the Blind Spot

The introspection security demands the agent execution to have both control flow integrity and data flow integrity. Data confidentiality is also required since the leakage of the introspection targets can help the adversary evade introspection. The EPT settings of the ImEE and of the target ensure that the adversary can only launch side-channel attacks, which is beyond the scope of the study.

The only attack vectors exposed by the ImEE to the adversary are the shared GPT and the target physical memory which are fully controlled by the adversary. The adversary can manipulate the VA-to-GPA mappings for P_c and P_t. Depending on the specific manipulation, either such attempts can be detected by the EPT violation triggered, or the attack does not adversely affect the introspection.

Detecting Blind Spot The attacks on P_c is defeated by the fact that the code frame is the only executable frame inside the ImEE. Hence, the attack on P_c's mapping, i.e. mapping P_c to a page in GP_t, is doomed to trigger an EPT violation exception. Similarly, mapping P_t to GP'_c also triggers EPT violations because the read is on an execute-only page.

Defeating Mapping Attacks The attack attempts that manipulate the mappings of P_t do not adversely affect the introspection. Specifically, there are three cases for the GP_t whose virtual page P_t is mapped to by the adversary.

- $GP_t = GP'_c$. Nonetheless, the EPT_C maps the agent code frame non-readable. Therefore, an EPT violation exception is thrown. The hypervisor can find out the faulting VA and reports to the VMI tool. The hypervisor can also reload the agent into a new executable page to introspect the faulting page. This is the same case as in detecting blind spot described above.

- $GP_t \neq GP'_c$, and GP_t is within the pre-assigned GPA range for the target VM. In this case, the ImEE's MMU walks the target VM's GPT and fetches the data in the same way as in the target VM. In other words, the mapping consistency between the ImEE and the target VM is still guaranteed. Although the agent

may read invalid data, its execution is not affected by such mappings. The attack has no harm to the execution as it is equivalent to feeding poisonous contents to the VMI application, in the hope to exploit a programming vulnerability. This is the inevitable risk faced by any memory introspection and can be coped with software security countermeasures.

- GP_t is mapped out of the pre-assigned GPA range for the target. If $GP_t = GP_d$ or $GP_t = GP_c$, the attack causes the agent to read from the ImEE frames; otherwise it causes an EPT page fault as the needed mapping is absent. This case is not considered as a blind-spot problem, because the target VM's EPT does not have the mapping for GP_t. Hence, the target VM's kernel, including the adversary, is not able to access this page. This attack does not give the adversary any advantage over mapping P_t to an in-range GPA whose physical frame stores the same contents prepared by the adversary. (Note that ImEE do not assume or rely on the secrecy of the introspection code.)

5.3.4 Operations of ImEE

Initialization To start the introspection, the hypervisor loads the needed agent code and data into the memory. It initializes EPT_T as a copy of the entire EPT used for the target, and allocates a vCPU core for the ImEE. The ImEE `CR3` is initially loaded with the address of GPT_L. target's page table directory address.

In case the target's EPT occupies too many pages, the hypervisor copies them in an on-demand fashion. In other words, when the agent's target memory access encounters a missing GPA-to-HPA mapping, the hypervisor then copies the EPT page from the target's EPT. Note that it does not weaken security or effectiveness, because the EPTs are managed by the hypervisor only.

Activation Based on the VMI application's request, the hypervisor launches the ImEE wherein the agent runs in the local address space with an arbitrarily chosen virtual address. The start of a session is marked by the target VM's `CR3` capture.

If it is the first session, the hypervisor may send out an Inter-Processor Interrupt (IPI) to the target VM, or induce an EPT violation to the target, or passively wait for a natural VM-exit (which is more stealthy). After trapping the core, the hypervisor configures the target's Virtual Machine Control Structure (VMCS) to intercept `CR3` updates on it. Namely, the execution of `CR3` loading instruction(s) on the captured vCPU triggers a VM exit. Note that the target's other vCPUs (if any) are not affected.

Agent Reloading Once the target `CR3` value is switched, the hypervisor sends an IPI to the ImEE CPU to cause it to trap to the hypervisor. The hypervisor then reloads the agent. If the agent is currently running in the target address space, its `CR3` in the VMCS is immediately replaced. The hypervisor then extracts the page frame number from the target's Instruction Pointer (IP). It replaces the page frame number in the ImEE IP with the one in the target IP without changing the offset. Since the agent code lies within one page, preserving the offset allows it to smoothly continue the interrupted execution.

If the agent is in the local address space, the `CR3` for the new target address space is saved in a register. The crux of the session transition is to *minimize the hypervisor execution time* as it hinders the ImEE's performance by holding the core.

ImEE use a *lazy-allocation* method to find GP'_c for the purpose of setting up EPT_C. When the agent resumes execution, an EPT violation is triggered because the corresponding physical page is mapped as read-only in EPT_T. From the exception, the hypervisor reads the faulting GPA, changes the corresponding EPT permissions, and restores the previous one to read-only. The newly modified EPT_T entry becomes the new EPT_C. Since the lazy method uses the MMU to find GP'_c, it saves the CPU time for walking the page table.

Page Fault Handling Although it is rare for kernel introspection, it is possible to encounter a page fault due to absent pages in the target VM. One possible reason is that the malware inside the target attempts to evade introspection by swapping out page content to disk. In this case, since the mapping inside ImEE is consistent

with the one in the target VM, introspection on the swapped-out page results in a page fault inside ImEE. This behavior is the expected consequence of maintaining mapping consistency between ImEE and the target. The effectiveness of ImEE's introspection is not undermined because once the swapped-out page is swapped in, it is visible to ImEE immediately.

For the sake of resilience, ImEE installs a page fault handler inside the ImEE. Since the agent resides in Ring 0, the exceptions do not cause any context switch. Out of the consideration of transparency and stealthiness, the ImEE's page fault handler does not attempt to resolve the cause. Instead, it simply runs dozens of `NOP` instructions and retries the read. If the rounds of failure exceed the predefined threshold, it aborts the execution.

5.4 Implementation

In this section, the details of ImEE prototype implementation is presented. The prototype is based on KVM. The introspection tools implemented on top of the prototype are also described.

5.4.1 ImEE on KVM

A prototype of the ImEE and its agent are implemented on Ubuntu 12.04 with Linux kernel 3.2.79. The implementation adds around 1400 SLOC to the Linux KVM module. The main changes on the KVM module include two new `ioctl` call handlers as the interface for the VMI application to request the ImEE setup and execution. The new handlers leverage existing KVM utility in the kernel to setup the ImEE as a special VM.

The KVM's handling of VM-exit events is customized in order to achieve better performance. Those events intended for the ImEE introspection are redirected to the new handler dedicated for the ImEE. Therefore, the long execution path of the KVM's event handling routines is bypassed.

5.4.2 Specialized Agent

According to the commonly seen memory reading patterns, three types of ImEE agents are implemented as listed in Table 5.2. The Type-1 agent performs a block read, i.e., to read a contiguous memory block at the base address. The Type-2 agent performs a traversal read, i.e., to read the specified member(s) of a list of structured objects chained together through a pointer defined in the structure. The Type-3 agent reads the memory in the same way as the Type-2, except that the extracted member is a pointer and a dereference is performed to read another structure. Note that the Type-2 and 3 agents are particularly useful for traversing the kernel objects.

Type	Mode of read	# of Instructions
1	Block-read	38
2	Traversal-read	22
3	Traversal-read-dereference	40

Table 5.2: Three ImEE agents. The Type-3 agent uses 2 pointer deferences while the Type-2 agent uses one.

The interface between the VMI application and the ImEE agent are two fixed-size buffers residing on the agent's data frame and being mapped into the VMI application's space. One buffer is for the request to the agent and the other stores the reply from the agent. Both buffers are guarded by one spin-lock to resolve the read-write conflict from both sides. When the ImEE session starts, the agent polls the buffer and serves the request. The VMI application ensures that the reply buffer is not overflowed. The polling based approach is faster than using interrupts as it does not induce any VM-exit/entry.

5.4.3 Usability

The simple interface of ImEE allows easy development of introspection tools. For common introspection tasks that focus on kernel data structures, the development requires a selection of the agent type, and a set of memory reading parameters including the starting virtual address, the number of bytes to read, and the offset(s)

used for traversal. Based on this method, four user space VMI programs are developed. They collect different critical kernel objects and have distinct memory reading behaviors. The objectives and logics of the four programs are explained below.

- syscalldmp It dumps totally 351 entries of the guest's system call table pointed to by `sys_call_table`. A continuous block of 1404 bytes from the guest is returned to the program.

- pidlist It lists all process identifiers in the guest. It traverses the `task_struct` list pointed to by the kernel symbol `init_task`, and records the `PID` value of every visited structure in the list. In total, 4 bytes are returned while 8 bytes are read from the guest for each task.

- pslist It lists all tasks' identifiers and task names stored in `task_struct`. A task's name is stored in the member `comm` with a fixed size of 16 bytes. Hence, 24 bytes are returned for each task node.

- credlist It lists all tasks' credential structures referenced by the `task_struct`'s `cred` pointer. In total, 116 bytes including the credential structure to the application for each task node are read. Hence, it takes more time than pidlist and pslist.

Because of their different memory access patterns, they run with different types of agents. The syscalldmp tool runs with Type-1 agent to perform block-reads. The pidlist and pslist programs work with Type-2 agent and the credlist program works with Type-3 agent. These tools are linked with a small wrapper code to interact with the ImEE-enabled KVM module via the customized `ioctl` handler.

5.5 Evaluation

The prototype is evaluated from four aspects with LibVMI as the baseline. LibVMI [72] is a cross-platform introspection library which a variety of tools depend on. LibVMI is the only open-source tool that provides a comprehensive set of API for reading the memory of a VM. In particular, it provides the capability to handle translation from VA to GPA. Therefore, LibVMI plays the role of a building block for live memory access in tools such as Drakvuf [61] and Volatility [95]. The evaluation consists of four parts. Firstly, the overhead of ImEE is evaluated, in terms of component costs and the impact on the target VM due to `CR3`-update interception. Secondly, the ImEE's throughput in reading the target memory is measured. Thirdly, the introspection performance of the tools is compared with two functionally equivalent ones implemented with the LibVMI and in the kernel. Lastly, ImEE is compared with LibVMI in a setting with multiple guest VMs.

The hardware platform used to evaluate the implementation is a Dell OptiPlex 990 desktop computer with an Intel Core i7-2600 3.4GHz processor (supporting VT-x) and 4GB DRAM. The target VM in the experiments is a normal KVM instance with 1GB of RAM and 1 vCPU.

5.5.1 ImEE Overhead

Table 5.3 summarizes the overheads of the ImEE. It takes a one-time cost of 97 μs to prepare the ImEE environment where the main tasks are to make a copy of the target guest EPT as $\texttt{EPT}_T$, to set up $\texttt{GPT}_L$ and $\texttt{EPT}_L$, and to allocate and setup the ImEE vCPU context. The ImEE activation requires about 3.2 μs, and the agent loading/reloading time is around 6.5 μs. The difference is mainly due to handling of the incoming IPI by host kernel on the ImEE core in the agent reloading case. In comparison, it takes about 100 milliseconds to initialize the LibVMI setting, which is around 1,000 times slower than the ImEE setup.

Guest CR3 Update Interception To maintain `CR3` consistency with the target

Overhead	ImEE	LibVMI
Launch time	97 μs	100 ms
Activation time	3.2 μs	-
Agent reloading time	6.5 μs	-

Table 5.3: Overhead comparison between ImEE and LibVMI.

during a session, the hypervisor intercepts the `CR3` updates. To evaluate its performance impact on the target, the entailed time cost is measured and run several benchmarks to assess the VM's performance.

The cost due to interception mainly consists of VM-exit, sending an IPI, recording VMCS data, and VM-entry. In total, it takes about 2000 CPU cycles which amounts 0.58 μs in the experiment platform. Three performance benchmarks are run: LMbench [6] for system performance, Bonnie++ [2] for disk performance and SPECint 2006 [10] for CPU performance while context switches during their executions are intercepted by the hypervisor. Figure 5.7 reports the LMbench score for context switch time where the performance drops about 50%.

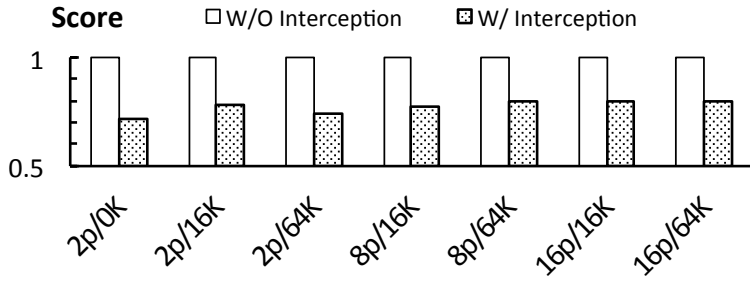


Figure 5.7: LMBench: normalized result on context switch time. The higher score means better performance.

Nonetheless, the interception does not seem to incur noticeable impact to other benchmark results such as disk I/O and network I/O, as shown in Figure 5.8, 5.9 and 5.10. This effect can be attributed to the relatively fewer number of context switches involved during the macro-benchmark runs, because the benchmark processes fully occupy the CPU time slot. It is typical for a Linux process to have between 1ms to 10 ms time-slot before being scheduled off from the CPU.

To understand the impact of `CR3` interception in real-life scenarios, three dif-

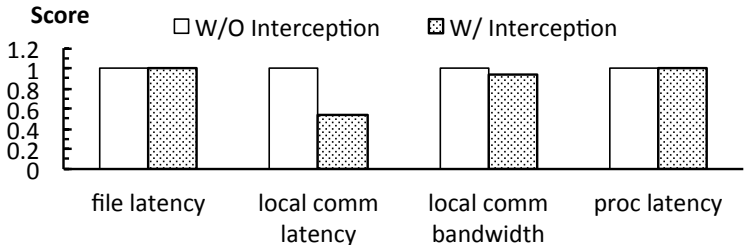


Figure 5.8: LMBench: normalized result on other system aspects. The higher score means better performance.

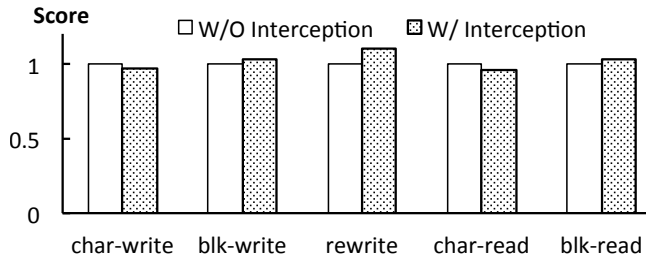


Figure 5.9: Bonnie++: normalized results on disk performance. The higher score means better performance.

ferent workloads are tested on the target VM: idle, online video streaming and file downloading. Neither test shows noticeable performance drop. When the target is under interception, the video is rendered smoothly without noticeable jitters and the file downloading still saturates the network bandwidth.

In the experiments, the introspection encounters few context switches in the target VM. To understand this phenomenon, experiments are run to measure the intervals between context switches. Figure 5.11 shows the distribution of their lengths under different workloads. The analysis shows that the context switch is expected to occur after around 40 μs, which could be used as a guideline for the VMI application to determine the duration of a session. Note that an encounter with the context switch costs about 6.5 μs for the introspection and 0.58 μs for the target VM.

Lastly, the ImEE has a small memory footprint of a few hundred KB on the host OS. LibVMI has a large memory footprint as it uses up to 14MB to perform a system call table dump.

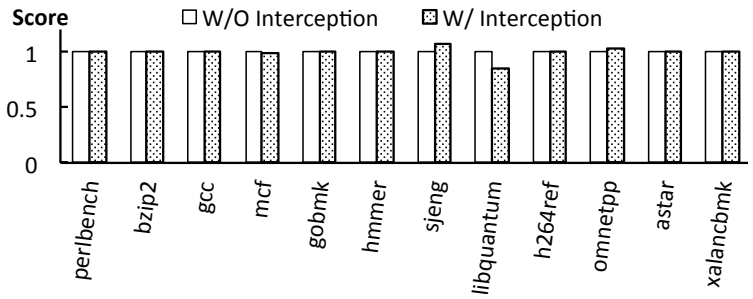


Figure 5.10: SPEC INT: normalized results on CPU performance. The higher score means better performance.

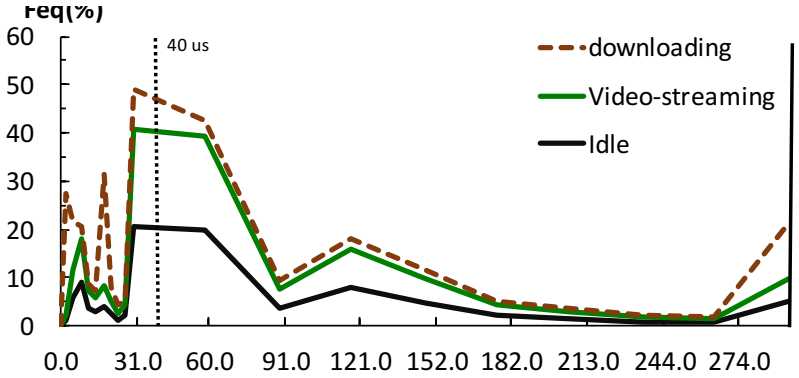


Figure 5.11: The frequency distribution of interval lengths between context switches in three workloads: idle, video streaming and file downloading. The x-axis is not displayed to the scale.

5.5.2 Guest Access Speed

The turnaround time for accessing the VM refers to the interval between sending a request and the arrival of the reply. It consists of the time spent for checking the shared buffers and the agent's execution time. To assess the efficiency of the ImEE's interface with the VMI application, the turnaround time is measured with the ImEE agent performing no task but returning immediately. The result is approximately 265 CPU cycles (or 77 ns).

To evaluate the memory-reading performance of the ImEE, experiments are run to evaluate the turnaround time with normal read requests. Table 5.4 below reports the turnaround time in comparison with LibVMI for the same workload. To make a fair comparison, LibVMI's translation cache is turned on whereas the page-level data cache is turned off.

# of Bytes	ImEE (μs)	LibVMI (μs)
4	0.353	18.4
64	0.358	18.5
128	0.389	18.4
512	1.643	18.9
1024	1.715	38.1

Table 5.4: Memory read performance comparison.

ImEE is also tested with the experiment described in Section 5.1. The experiment shows that the modification on the `cred` address is caught immediately when the malware makes the first attack. Note that with the ImEE support, it takes less than 1200 CPU cycles for the VMI application to get a DWORD from the guest, in contrast to more than 60,000 cycles using LibVMI. The maximum introspection frequency of ImEE based introspection is 2.83 MHz while an introspection using LibVMI in the experiment setting can only achieve 54 KHz in maximum.

5.5.3 Introspection Performance Comparison

The introspection tools (syscalldump, pidlist, pslit and credlist) are run in three settings: within the kernel, with ImEE, and with LibVMI. Since this set of tests concerns with real-life scenarios, LibVMI is tested on both KVM and Xen for completeness. For each of the scenario, the turnaround time of introspection is measured. The time for the processing of the semantics and the time for setting up the ImEE/LibVMI are not included in the measurement. Table 5.5 and Table 5.6 summarize the results.

Tools	Kernel module	ImEE	
		time	mode
syscalldmp	0.2	2.9	block
pidlist	10	31.6	traversal
pslist	10.4	38.6	traversal
credlist	25.3	25.6	hybrid

Table 5.5: Kernel object introspection performance in kernel and ImEE (time in μs).

The experiments show that the ImEE-based introspection has a comparable per-

Tools	LibVMI(KVM / Xen)		
	without any cache	**without page cache**	**with all cache**
syscalldmp	28.2 / 43	18.7 / 47	2 / 54
pidlist	5,887 / 2,180	2,864 / 2,041	1,568 / 490
pslist	8,319 / 1477	2,695 / 1,442	1,672 / 542
credlist	8,234 / 2,274	7,150 / 2,153	2,215 / 757

Table 5.6: Kernel object introspection performance by LibVMI (time in μs).

formance to running inside the kernel. It has a superior performance advantage over LibVMI for traversing the kernel object lists. On KVM, The LibVMI based introspection is around 50 times slower than the ImEE with all caches and 300 times slower without cache. On Xen, LibVMI is around 15 times and 70 times slower, respectively. Since the traversal only returns a few bytes from different pages, LibVMI's optimization in bulk data transferring does not result in performance gain.

5.5.4 Handling Multiple VMs

In a data center setting, a large number of VMs are hosted on the same physical server. Therefore, for a VMI solution to be effective in such a setting, the capability to handle multiple VM is important. Besides raw introspection speed, two additional capabilities are important for a VMI solution. Firstly, the VMI solution should respond quickly to requests to introspect VMs encountered for the first time. Secondly, it should also maintain swift response for introspection requests on VMs already launched.

The time taken for LibVMI and ImEE to perform a syscall table dump by the tools is measured in two scenarios. Four VMs are launched on the experiment platform. Firstly, the time taken for each solution to introspect four VMs once for each in a sequence is measured. It takes 561 ms for LibVMI and 377 μs for ImEE, respectively. In this case, LibVMI is about 1,400 times slower than ImEE. The performance of LibVMI is mainly due to the initialization needed for each newly encountered VM.

Secondly, the time taken for each solution for switching the introspection tar-

get among the four VMs that are already scanned is measured. The switching requires resetting certain data between consecutive scans. For this purpose, LibVMI is slightly modified to allow us to update the `CR3` value in the introspection context of a VM with a new one. The experiment shows that it takes 19 ms for LibVMI to perform such work while 4.4 μs for ImEE. ImEE shows around 4,300 times speed up. The reason is that LibVMI's software-based approach needs to reset a number of memory states. In contrast, ImEE only needs to fetch the current `CR3` on the target VM's vCPU and replace the ImEE `CR3`, `IP` and the EPT root pointer of the ImEE vCPU.

5.6 Discussions

5.6.1 CPU State

In-memory paging structure is only one of the factors that determines the final outcome of the translation of a virtual address. In fact, the final outcome is determined by both in-memory state and in-CPU states. The affecting in-CPU states include control registers and buffers such as the TLB. For example, the TLB can be intentionally made out-of-sync with paging structures in memory, therefore causes the introspection code to use a different mapping from the one currently used by the target. An ideal introspection solution should take into consideration both sets of states because they collectively represent the current address translation.

However, for out-of-VM live introspection, it is required that it runs on a core that is independent of the target VM. This limits the introspection's capability to utilize such in-CPU states because there is no mechanism to fetch in-CPU states from another CPU. One possible solution is to preempt the vCPU of the target on a physical core by a more privileged entity such as the hypervisor, trying to preserve as many in-CPU states as possible, including buffers and caches. However, the behavior of the buffers and caches when across VM transition is not fixed. Therefore,

without hardware assistance, attempts to implement an ideal solution is likely met with hardware-specific tweaks and hacks, making it very difficult.

5.6.2 Integration with Existing VMI Tools

The ImEE serves as the guest access engine for the VMI applications without involving kernel semantics. It is not challenging to retrofit exiting VMI tools that focus on high-level semantics to benefit from the ImEE's performance and security. Using VMST [42] as an example, we show how to combine a VMI application with the ImEE. When an introspection instruction is executed in VMST, the XED library [14] decides whether a data access should be redirected to the guest VM or not. If so, the code fetches the data from the guest memory by traversing the guest VM's page table in the same way as LibVMI. It is easy to integrate VMST with the ImEE. When a read redirection is generated by the XED library, the code simply issues a memory read request to the ImEE and waits for the reply. With the support from the ImEE, shadow TLB and shadow `CR3` proposed in VMST are no longer needed.

5.6.3 ImEE vs. In-VM Introspection

Strictly speaking, the ImEE and in-VM introspection systems are not comparable, as they are geared for different purposes. The ImEE is for effective target VM access while in-VM systems are designed for reusing the OS's capability [49, 24] or for monitoring events in the guest [81]. Since Process-Implant [49] and SYRINGE [24] rely on a trusted guest kernel, the ImEE is compared with SIM [81] from the perspective of accessing the target VM memory.

Security Address space isolation in SIM prevents the target VM kernel from tampering with SIM data and code. In a multi-core VM, it does not prevent the target VM kernel from interrupting SIM code execution by using non-maskable interrupts. By knocking down the SIM thread from its CPU core, the rootkit can safely erase the attack traces without being caught. In comparison, the entire ImEE

environment is separated from the target VM. It is much more challenging (if not feasible) for the target VM kernel to disrupt the ImEE agent's execution. Note that the manipulation on the page tables backfires on the adversary since they are shared between the adversary and the target.

Effectiveness SIM does not enforce consistent address mappings. The SIM code and the target VM threads are in separated address spaces, namely using separated page tables. The SIM hypervisor does not update the SIM page tables according to the updates in the kernel. In comparison, any update on the target VM page table takes immediate effect on the ImEE and `CR3` consistency is ensured by the hypervisor.

Performance and Usability Both SIM and ImEE make native speed accesses to the memory without emulating the MMU. ImEE uses EPT and does not require any modification on the target VM, while SIM relies on the shadow page tables and makes non-negligible changes on the target VM.

5.6.4 Paging Modes Compatibility

The design of ImEE is by nature compatible with various paging modes such as Physical Address Extension mode (PAE mode) and 64-bit paging. It only requires setting of two additional bits in the control registers, namely `PAE` bit in `CR4` register and `LME` bit in `EFER` register so that the ImEE core runs in the needed paging mode. To prevent the adversary from changing the paging mode, the hypervisor trap access to the above registers. To introspect a 64-bit VM, the agent needs to be compiled into 64-bit code as well. In fact, the ImEE performs better on a 64-bit platform, because there are more general purpose registers available, reducing the number of address space switches, and the PCID can be used to prevent the needed TLB entries from being flushed.

5.6.5 Architecture Compatibility

The ImEE's design is also compatible to other multi-core architectures such as ARM, on the condition that the hardware supports MMU virtualization. Like the x86 platform, ARM multi-core processors also feature a per-core MMU, thus each core's translation can be performed independently. As a result, a core can be set up to use the translation used by the other, by setting it to use the same root of paging structures. Moreover, by using $TTBR0$ and $TTBR1$, the hypervisor can easily separate the virtual address ranges used for the target accessing and for the local usage. It simplifies the design as both can use separated page tables. The ARM processor also grants the software more control over the TLB entries. Thus, the needed TLB entries can be locked by the agent. Therefore, it is expected to have better performance than the current design.

Chapter 6

Conclusion

The discussion in this dissertation is concerned with an adversary with the kernel privilege. The adversary may gain such privilege via infecting the system with rootkits, or by modifying the underlying OS behavior with launching attacks. Due to the high privilege, the adversary can cause serious consequences by performing arbitrary attacks. It may access encryption keys in the process memory, read files which only legitimate users can access, or even manipulate the devices directly. In a traditional system architecture, it is extremely challenging to defend against such an adversary.

By introducing an additional layer below the OS in the traditional system architecture, virtualization techniques turn out to be an effective way to defend against the adversaries with kernel privilege. In the virtualization architecture, the OS is no longer assigned the highest privilege. Instead, the hypervisor assumes it. The OS is in turn, deprivileged to run in a domain which is managed by the hypervisor. In this architecture, even if the adversary obtains the kernel privilege, it is still confined and restricted by the hypervisor. Therefore, the attacker's damage is contained.

A number of systems have been proposed based on this architecture. Some systems directly utilize the virtualization architecture and run different applications in different domains according to their trustworthiness. The trusted application runs in the trusted domain, while the untrusted in the untrusted domain. Therefore, even

if the untrusted domain is attacked, the trusted domain is still not affected because they are isolated. This approach, termed domain isolation, still requires the trusted domain to have a complete OS which is included in its TCB. Therefore, the TCB of this design is still large.

Later systems attempted to reduce this TCB by reusing certain facilities in the untrusted domain such as dynamic memory allocation and file systems. However, this design blurs the isolation boundary between the trusted domain and untrusted and leads to potential issues. The most prominent issue is the reuse of guest page tables. The guest page tables define the VA-to-GPA mappings, therefore, the address space of the trusted domain is still, in part, controlled by the untrusted domain. Existing systems realized this issue and all perform certain kind of checks on the guest page tables to ensure that the mappings in the guest page table do not violate the desired policy.

However, such checks are usually ad-hoc and not systematic. There is no clear assumptions stated about the capabilities of the attacker. Therefore, when faced with attacker with new capabilities, such checks quickly fail. As exemplified by the multicore platform attacks, i.e. the stifling attack and the VPID attack, the concurrent execution of the guest kernel allows modification of the guest page table either during the check or after the check by introducing race conditions. Therefore, the effectiveness of the systems requires re-investigation.

In the re-investigation, the analysis is based on a common pattern of the construction of these systems. These systems all need to translate a set of high-level policies into a form that can be expressed with abstractions in the virtualization context. Three elements in the policies need to be translated, namely, subjects, objects and operations. Plus, due to the idiosyncrasies of the virtualization mechanism, additional runtime considerations also need to be included. The analysis identified issues with the management of the identities of the subjects and objects as well as a few runtime enforcement issues. All these issues affect the effectiveness of the policy enforcement. In addition, two attacks are presented that shows that the ker-

nel level adversary can leverage hardware features and concurrent execution to keep stale permission, so that it can break the isolation boundary setup by the hypervisors which does not consider multicore platforms.

The FIMCE system is designed to address the issues uncovered during the analysis. FIMCE is a fully isolated and flexible environment that completely isolates the malicious kernel from sensitive applications. The memory, CPU states and any devices are all isolated with a carefully delineated boundary. In FIMCE, the identity issues do not exist because the FIMCE instances are directly managed by the hypervisor. The issues caused by concurrent execution does not exist because of the complete isolation boundary. On top of its security, FIMCE is also a flexible and nimble architecture that can be tailored for various use cases. Meanwhile, FIMCE comes with minimum interference with the execution of both the isolated application and the guest VM, thereby minimizing performance overheads.

Lastly, it is shown that the FIMCE environment can be applied in the context of VMI and the ImEE system is presented. The ImEE introspection system builds on top of the idea of FIMCE and tweaks its address mappings. The ImEE environment is configured with the same page tables used by the target VM under introspection. Therefore, it has a consistent address mapping with the guest. The guest's modification on the page table will take effect immediately inside ImEE as well, therefore, it cannot make transient changes and try to hide the presence of malicious behavior. With the help of hardware, ImEE runs with a remarkable performance boost while still keeping the consistent address mappings.

The systems presented in this document can be extended along a few directions. The integration between FIMCE and SGX can be explored to achieve strong isolation with secure I/O. The ImEE can be integrated with existing tools for improved security and performance. The designs presented can also be applied in other architectures such as ARM, which forms the cornerstone of the mobile computing world.

Acronyms

API: Application Programming Interface

APIC: Advanced Programmable Interrupt Controller

BIOS: Basic Input/Output System

CPU: Memory Management Unit

CPL: Current Privilege Level

DMA: Direct Memory Access

DPL: Descriptor Privilege Level

DRTM: Dynamic Root of Trust for Measurement

ELF: Executable and Linkable Format

EOI: End Of Interrupt

EPC: Enclave Page Cache

EPT: Extended Page Table

FIMCE: Fully Isolated Micro-Computing Environment

GDT: Global Descriptor Table

GOT: Global Offset Table

GPA: Guest Physical Address

GPT: Guest Page Table

GPU: Graphics Processing Unit

HPA: Host Physical Address

HTML: Hyper Text Markup Language

IDT: Interrupt Descriptor Table

IID: Interface Identifier

ImEE: Immersive Execution Environment

IOMMU: Input-Output Memory Management Unit

IP: Instruction Pointer

IPI: Interprocessor Interrupt

IRQ: Interrupt Request

KVM: Kernel Virtual Machine

LAN: Local Area Network

MMIO: Memory Mapped I/O

MMU: Memory Management Unit

NMI: Non-Maskable Interrupt

OS: Operating System

PA: Physical Address

PAE: Physical Address Extension

PAL: Piece of Application Logic

PCID: Process-Context Identifier

PCR: Platform Configuration Register

PLID: Pillar Identifier

PLT: Procedure Linkage Table

PTE: Page Table Entry

SLOC: Source Line of Code

SGX: Software Guard Extension

SMM: System Management Mode

SMP: Symmetric Multiprocessing

SSL: Secure Sockets Layer

TCB: Trusted Computing Base

TCS: Thread Control Structure

TEE: Trusted Execution Environment

TLB: Translation Lookaside Buffer

TPM: Trusted Platform Module

TSS: Task-State Segment

TTP: Trusted Third Party

VA: Virtual Address

VM: Virtual Machine

VMCS: Virtual Machine Control Structure

VMM: Virtual Machine Monitor

VMI: Virtual Machine Introspection

VPID: Virtual Processor Identifier

Bibliography

[1] adore-ng. Online at https://github.com/trimpsyw/adore-ng.

[2] Bonnie++. http://www.coker.com.au/bonnie++/.

[3] The current state of kernel page-table isolation. https://lwn.net/Articles/741878/. Accessed: 2018-05-08.

[4] Idetect. Online at http://forensic.seccure.net/.

[5] Lines of code of the linux kernel versions. https://www.linuxcounter.net/statistics/kernel.

[6] Lmbench - tools for performance analysis. http://www.bitmover.com/lmbench/.

[7] Lsproc. Online at http://windowsir.blogspot.com/2006/04/lsproc-released.html.

[8] PROCENUM. Online at http://forensic.seccure.net/.

[9] Red Hat Crash Utility. Online at http://people.redhat.com/anderson/.

[10] Standard performance evaluation corporation. https://www.spec.org/cpu2006/.

[11] Trusted boot, tcg group. http://tboot.sourceforge.net/.

[12] Volatilitux. Online at https://code.google.com/p/volatilitux/.

[13] Windows Memory Forensic Toolkit. Online at http://forensic.seccure.net/.

[14] XED: x86 encoder decoder. http://www.pintool.org/docs/24110/Xed/html/.

[15] J. P. Anderson. Computer security technology planning study. volume 2. Technical report, Anderson (James P) and Co Fort Washington PA, 1972.

[16] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *Proceedings of the 13rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703, 2016.

[17] 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.

[18] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102. ACM, 2014.

[19] A. M. Azab, P. Ning, and X. Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, pages 375–388. ACM, 2011.

[20] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[21] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of USENIX Annual Technical Conference, (FREENIX Track)*, pages 41–46, 2005.

[22] C. Betz. Memparser. *2005, http://www. dfrws. org/2005/challenge/memparser. shtml*, 2005.

[23] C. Bugcheck. Grepexec: Grepping executive objects from pool memory. In *Report from the Digital Forensic Research Workshop (DFRWS)*, 2006.

[24] M. Carbone, M. Conover, B. Montague, and W. Lee. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*, pages 22–41. Springer, 2012.

[25] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 555–565. ACM, 2009.

[26] 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 *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[27] Y. Cheng and X. Ding. Guardian: Hypervisor as security foothold for personal computers. In *International Conference on Trust and Trustworthy Computing*, pages 19–36. Springer, 2013.

[28] Y. Cheng, X. Ding, and R. H. Deng. Driverguard: a fine-grained protection on I/O flows. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)*, 2011.

[29] Y. Cheng, X. Ding, and R. H. Deng. Efficient virtualization-based application protection against untrusted operating system. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[30] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 565–578, Denver, CO, June 2016. USENIX Association.

[31] V. Costan and S. Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[32] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 763–780. IEEE, 2015.

[33] S. Cristalli, M. Pagnozzi, M. Graziano, A. Lanzi, and D. Balzarotti. Micro-virtualization memory tracing to detect and prevent spraying attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 431–446. USENIX Association.

[34] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–96, 2014.

[35] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium*, pages 601–615, 2012.

[36] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *ACM SIGPLAN Notices*, volume 50, pages 191–206. ACM, 2015.

[37] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 289–298. ACM, 2013.

[38] A. Desnos. Draugr-live memory forensics on linux. https://code.google.com/archive/p/draugr/.

[39] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 297–312, Washington, DC, USA, 2011. IEEE Computer Society.

[40] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (S&P), 2011 IEEE Symposium on*, pages 297–312. IEEE, 2011.

[41] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 417–426. ACM, 2010.

[42] Y. Fu and Z. Lin. Space traveling across VM: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 586–600. IEEE, 2012.

[43] Y. Fu and Z. Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (S&P), 2012 IEEE Symposium on*, pages 586–600. IEEE, 2012.

[44] Y. Fu and Z. Lin. Exterior: Using a dual-VM based external shell for guest-OS introspection, configuration, and recovery. *ACM SIGPLAN Notices*, 48(7):97–110, 2013.

[45] T. Garfinkel et al. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, volume 3, pages 163–176, 2003.

[46] 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 (SOSP)*, 2003.

[47] G. M. Garner Jr. Kntlist. *2005, http://www. dfrws. org/2005/challenge/kntlist. shtml*, 2005.

[48] 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 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[49] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*, pages 147–156. IEEE, 2011.

[50] J. Hizver and T.-c. Chiueh. Real-time deep virtual machine introspection and its applications. In *ACM SIGPLAN Notices*, volume 49, pages 3–14. ACM, 2014.

[51] 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 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[52] H. Inoue, F. Adelstein, M. Donovan, and S. Brueckner. Automatically bridging the semantic gap using c interpreter. In *Proc. of the 2011 Annual Symposium on Information Assurance*, pages 51–58, 2011.

[53] Intel Corporation. Innovative instructions and software model for isolated execution. http://privatecore.com/wp-content/uploads/2013/06/HASP-instruction-presentation-release.pdf, 2013.

[54] T. Jaeger, R. Sailer, and U. Shankar. Prima: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28. ACM, 2006.

[55] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang. The web/local boundary is fuzzy: A security study of chrome's process-based sandboxing. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 791–804. ACM, 2016.

[56] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM.

[57] C. H. Kim, S. Park, J. Rhee, J.-J. Won, T. Han, and D. Xu. Cafe: A virtualization-based approach to protecting sensitive cloud application logic confidentiality. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 651–656. ACM, 2015.

[58] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[59] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *ACM SIGPLAN Notices*, volume 51, pages 277–290. ACM, 2016.

[60] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *USENIX Security Symposium*, pages 511–526, 2013.

[61] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 386–395. ACM, 2014.

[62] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[63] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS*, 2011.

[64] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[65] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *USENIX Security Symposium*, pages 243–258, 2008.

[66] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 416–427. IEEE, 2014.

[67] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619. ACM, 2015.

[68] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (S&P), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[69] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, Apr. 2008.

[70] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 28–37, New York, NY, USA, 2012. ACM.

[71] J. M. N. L. Petroni, T. Fraser and W. A. Arbaugh. Copilot—a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, Aug. 2004.

[72] B. D. Payne. Simplifying virtual machine introspection using LibVMI. Technical Report SAND2012-7818, Sandia National Laboratories, 2012.

[73] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy (S&P), 2008 IEEE Symposium on*, 2008.

[74] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, 2006.

[75] J. Ren, Y. Qi, Y. Dai, X. Wang, and Y. Shi. Appsec: A safe execution environment for security sensitive applications. In *ACM SIGPLAN Notices*, volume 50, pages 187–199. ACM, 2015.

[76] 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.

[77] S. Rostedt. The x86 NMI iret problem. https://lwn.net/Articles/484932/. Accessed: 2015-11-10.

[78] A. Saberi, Y. Fu, and Z. Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2014.

[79] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, 2004.

[80] 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 the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[81] 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*, pages 477–487. ACM, 2009.

[82] 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 5th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009.

[83] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[84] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: an efficient out-of-VM approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 363–374. ACM, 2011.

[85] A. Srivastava and J. T. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *NDSS*. The Internet Society, 2011.

[86] 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*, pages 2–13. ACM, 2012.

[87] S. Suneja, C. Isci, E. de Lara, and V. Bala. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 133–146. ACM, 2015.

[88] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: making trust between applications and operating systems configurable. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[89] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 256–267. ACM, 2015.

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

[91] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2015.

[92] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2014.

[93] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy (S&P), 2014 IEEE Symposium on*, 2014.

[94] A. Vasudevan, B. Parno, N. Qu, V. D. Gligor, and A. Perrig. Lockdown: A safe and practical environment for security applications. *CMU-CyLab-09-011*, 14, 2009.

[95] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework, 2007.

[96] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM.

[97] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis. No-execute-after-read: Preventing code disclosure in commodity software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 35–46. ACM, 2016.

[98] Z. W. Y. Q. Y. Z. Xiaoguang Wang, Yue Chen. Secpod: a framework for virtualization-based security systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 347–360, Santa Clara, CA, July 2015. USENIX Association.

[99] X. Xiong, D. Tian, P. Liu, et al. Practical protection of kernel integrity for commodity os from untrusted extensions. In *NDSS*, volume 11, 2011.

[100] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (S&P), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[101] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2008.

[102] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[103] M. Yu, V. D. Gligor, and Z. Zhou. Trusted display on untrusted commodity platforms. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 989–1003. ACM, 2015.

[104] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Security and Privacy (S&P), 2012 IEEE Symposium on*, S&P, May 2012.

[105] Z. Zhou, M. Yu, and V. D. Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *Security and Privacy (S&P), 2014 IEEE Symposium on*, pages 308–323. IEEE, 2014.