6-2018

FIMCE: A fully isolated micro-computing environment for multicore systems

ZHAO

Xuhua DING
Singapore Management University, xhding@smu.edu.sg

DOI: https://doi.org/10.1145/3195181

Published in ACM Transactions on Information and System Security, 2018 August, Volume 21, Issue 3
https://doi.org/10.1145/3195181

FIMCE: A Fully Isolated Micro-Computing Environment for Multicore Systems

SIQI ZHAO and XUHUA DING, Singapore Management University, Singapore

Virtualization-based memory isolation has been widely used as a security primitive in various security systems to counter kernel-level attacks. In this article, our in-depth analysis on this primitive shows that its security is significantly undermined in the multicore setting when other hardware resources for computing are not enclosed within the isolation boundary. We thus propose to construct a fully isolated micro-computing environment (FIMCE) as a new primitive. By virtue of its architectural niche, FIMCE not only offers stronger security assurance than its predecessor, but also features a flexible and composable environment with support for peripheral device isolation, thus greatly expanding the scope of applications. In addition, FIMCE can be integrated with recent technologies such as Intel Software Guard Extensions (SGX) to attain even stronger security guarantees. We have built a prototype of FIMCE with a bare-metal hypervisor. To show the benefits of using FIMCE as a building block, we have also implemented four applications which are difficult to construct using the existing memory isolation method. Experiments with these applications demonstrate that FIMCE imposes less than 1% overhead on single-threaded applications, while the maximum performance loss on multithreaded applications is bounded by the degree of parallelism at the processor level.

CCS Concepts: • **Security and privacy** → **Virtualization and security**;

Additional Key Words and Phrases: Virtualization, isolation, multicore platform, hypervisor

ACM Reference format:
Siqi Zhao and Xuhua Ding. 2017. FIMCE: A Fully Isolated Micro-Computing Environment for Multicore Systems. *ACM Trans. Priv. Secur.* 21, 3, Article 15 (April 2018), 30 pages.
https://doi.org/10.1145/3195181

1 INTRODUCTION

Hardware-assisted virtualization has become one of the mainstream features of modern commodity processors in high-performance servers, personal computers, and even mobile phones. With hardware virtualization support, a hypervisor running directly on the hardware is more privileged than operating systems (OSes). The literature on virtualization-based security has shown that compact bare-metal hypervisors can be used to harden the kernel [28] and to cope with

This research effort is supported by the Singapore National Research Foundation under the NCR Award Number NRF2014NCR-NCR001-012.

The conference version of this work [40] focuses more on uncovering security pitfalls of virtualization-based memory isolation and sketches out FIMCE as a solution. This manuscript elaborates the system design details of FIMCE (Section 5.2) and compares it with other virtualization-based isolation (Section 5.3). Other major additions include a discussion on SGX and FIMCE (Section 6), a detailed description of FIMCE's modularized software infrastructure (Section 7), and more implementation details (Section 9.2).

Authors' addresses: S. Zhao and X. Ding, School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902; emails: {siqi.zhao.2013, xhding}@smu.edu.sg.

the kernel-level attacks on applications [6, 9, 10, 16, 18, 24, 38] and I/O channels [8, 29, 42]. The security primitive commonly used in these systems is virtualization-based memory isolation via a special paging structure introduced by Memory Management Unit (MMU) virtualization. For example, Intel's virtualization technology uses the so-called Extended Page Tables[1] (EPTs) which are traversed by the hardware during address translation. The EPTs (and their counterparts on non-Intel processors) are solely managed by the hypervisor and are not accessible to any guest software, including the kernel. Thus, by setting the access permission bits on the EPTs, the hypervisor is empowered to police memory accesses in the guest so that sensitive memory pages can be isolated from untrusted software.

Most existing systems [9, 18, 24, 28] built on memory isolation implicitly assume a unicore platform. Although the eXtensible and Modular Hypervisor Framework (XMHF) [35] and the On-demand Software Protection (OSP) [10] explicitly describe their mechanisms in a multicore setting, their designs essentially impose a single-threaded execution model for the hypervisor. Nonetheless, the prevalence of multicore processors in a wide spectrum of computing devices (from mobile phones to high-performance servers) necessitates revisiting the memory isolation primitive in the multicore context. We observe that parallel execution on a multicore platform makes a subtle, yet far-reaching impact on security because the adversarial thread on one core can attack the protected thread on another, a scenario that never occurs on a unicore system.

Our work presented in this article begins with an in-depth analysis of memory isolation in a multicore setting. After scrutinizing both the hardware architecture and the system design, we identify several security and performance complications, including cumbersome EPT management, insecure page table verification, and incapable thread identification. To validate our assertion that the memory isolation primitive is *ineffective*, we have schemed and implemented two attacks on XMHF [35] and BitVisor [29], respectively, to successfully invade protected memory pages.

The deep-seated reason for ineffective isolation is that the current isolation schemes ignore other critical elements in a computing environment including, most notably, the Central Processing Unit (CPU) core. The incomplete isolation boundary thus becomes an exploitable vulnerability in a multicore setting as it allows for parallel executions. We hence propose the *Fully Isolated Micro Computing Environment* (FIMCE) as a new isolation primitive for multicore platforms. FIMCE integrates memory isolation with core isolation and, optionally, I/O device isolation. It offers stronger security than the existing primitive with a much reduced attack surface, and it is also more versatile to support more types of applications.

FIMCE features a malleable hardware setting which can serve different security needs. Its software infrastructure in the isolated environment is constructed in a modular fashion according to application demands. Thus it does not require the protected application code to be self-contained. Due to its architectural niche, it can also be used as a trusted environment running in parallel with the untrusted kernel in order to monitor kernel activities. We have implemented a prototype of FIMCE including a micro-hypervisor. We have also measured its performance by running benchmarks, and we built four security systems to showcase its versatility. The experimental results confirm FIMCE's strong security and high usability, with a modest cost of performance due to a lesser degree of parallelism available for the OS to use.

Recent years have witnessed the booming adoption of Intel's Software Guard Extensions (SGX) technology to isolate the memory and execution of security-sensitive code. With hardware enforcing memory access policies, SGX-based isolation attains stronger security than hypervisor-based isolation. Nonetheless, SGX lacks the capability of I/O device isolation and requires the (untrusted)

[1]The equivalent structure is called the Nested Page Table on AMD's processors and the Stage 2 page table on ARM processors.

OS to set up and configure enclaves. In contrast, FIMCE offers built-in I/O support and has no dependence on the OS. In this article, we also compare FIMCE and SGX in detail and show how the two techniques can be integrated to complement each other.

The main contributions of our work are summarized here:

(1) We show that the existing virtualization-based memory isolation primitive is ineffective on multicore platforms. Our assessment is based on an in-depth examination of design defects and is evidenced by concrete attacks on XMHF [35] and BitVisor [29].
(2) We propose FIMCE as a new isolation primitive for multicore systems. It offers stronger security and better versatility than its predecessor. The key idea behind FIMCE is to isolate all hardware resources needed by the protected execution, instead of memory pages only.
(3) We implement a prototype of FIMCE and assess its performance with benchmarks. We also use it as a building block to construct several security applications in order to showcase its attractive features including the modularized software infrastructure, I/O support, and the malleable hardware setting.

Organization. In the next section, we explain the background of address translation and memory accesses in multicore systems. Then, we show two concrete attacks on XMHF and BitVisor in Section 3, followed by an in-depth examination of memory isolation in Section 4. We propose the design of FIMCE in Section 5 and compare it with SGX in Section 6. Next, we present the software architecture of FIMCE in Section 7, and we discuss possible applications of FIMCE in Section 8. Section 9 evaluates security and performance based on experiments with a FIMCE prototype. We describe related work in Section 10 and conclude the article with Section 11.

2 BACKGROUND

Address Translation with Virtualization. With hardware-assisted memory virtualization, address translation proceeds with two stages for any memory access from 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 aborts if the memory access conflicts with the permissions specified in these page tables.

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

Memory Access in Symmetric Multiprocessor (SMP) Systems. In an SMP setting, multiple cores access the shared physical memory, as shown in Figure 1. Nonetheless, 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 page tables. The GPA-to-HPA mappings are controlled by the hypervisor. By default, *all* cores use the same set of EPTs since the GPA-to-HPA mappings are for the entire virtual machine and the guest kernel's thread scheduling is transparent to the hypervisor.

In the SMP setting, it is more complicated to maintain TLB consistency since the threads in each core may modify the shared page tables while the TLBs are local to each core. Typically, the thread

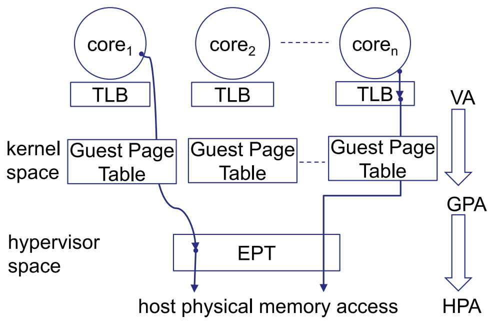


Fig. 1. The paradigm of memory access in an SMP setting. The first core has TLB misses and accesses the memory via the guest page tables and the EPTs, while the last core has TLB hits and accesses the memory without consulting any page table.

that modifies the paging structures performs the so-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 affiliated core. The Advanced Programmable Interrupt Controller (APIC) is responsible for receiving and sending IPIs. Its proper behavior affects the success of TLB shootdown and consequently TLB consistency.

3 ATTACKS

As a prologue to our in-depth analysis of memory isolation and the proposal of FIMCE, we present concrete attacks on two open-source micro-hypervisors, BitVisor [29] and XMHF [35], running on a desktop computer with multiple cores. The intention is to exemplify the weakness of the widely used memory isolation primitive instead of targeting these two systems alone. The success of the attacks indicates that it requires meticulous checking of all system design details to realize memory isolation in multicore settings.

Without loss of generality, 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 our 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 continues to use the write permission granted before the EPT update. Next, we describe the stifling attack and the Virtual Processor ID (VPID) attack.

3.1 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.

Our 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.

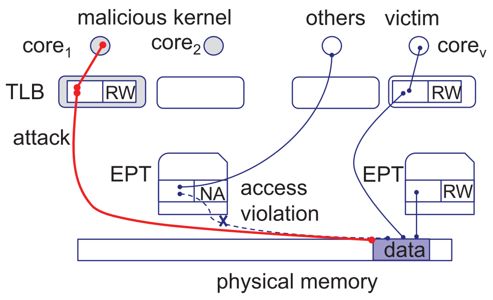


Fig. 2. Illustration of the stifling attack bypassing the EPT's access control over the victim's data. The attacker controls core$_1$ and core$_2$.

We explain the attack details with an example. As depicted in Figure 2, suppose that the victim application occupies core$_v$ while two malicious kernel threads occupy core$_1$ and core$_2$. The attack steps are as follows:

(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 triggering 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.

We have implemented the attack on BitVisor [29] 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 [17, 33].

3.2 Virtual Processor ID (VPID) Attack

XMHF [35] 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 Nonmaskable 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 attached to their TLB entries. When a TLB entry is hit during translation, it is valid 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 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, we devise the following attack for 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 2.

(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 and 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 Application Programming Interfaces (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. We observed that the malicious thread keeps a stale TLB entry and successfully writes the target page without triggering EPT violation.

Caveat. We reiterate that the intention of presenting this attacks is not to show the vulnerability of these schemes, but to emphasize that applying memory isolation in the multicore setting is *not* as straightforward as it seems. The low-level system subtleties, if not treated properly and thoroughly, weaken the security strength. The next section provides more insights into this issue.

4 IN-DEPTH EXAMINATION OF MEMORY ISOLATION

Memory isolation is the primitive used in almost all virtualization-based security systems [9, 13, 18, 24, 28, 32, 35]. It denies any unauthorized access to the concerned physical memory pages. Most existing schemes in the literature focus on how to make use of it to achieve their high-level security goals, assuming that the primitive itself is secure. In this section, we examine its security under the multicore context.

4.1 System and Threat Models

The system in consideration is a commodity multicore platform running a bare-metal micro-hypervisor with a single guest virtual machine. The adversary is the guest kernel, which is subverted and controlled by malware. With kernel privilege, the adversary can launch arbitrary software-based attacks, such as illicit memory accesses and execution context manipulation. It may even use a CPU emulator to emulate the hypervisor's behavior. We assume that the Basic Input/Output System (BIOS), the firmware, and the hardware in the platform are not compromised by the adversary and behave in compliance with their respective specifications. We trust the micro-hypervisor code, data, and control flow throughout the lifetime of the platform. We do not consider side channel attacks nor denial-of-service attacks. For the convenience of presentation, we use "the guest" and "the kernel" throughout this article to refer to the virtual machine in the system and its kernel, respectively. Note that the models just described mirror those used in the literature [9, 13, 18, 24, 28, 32, 35].

4.2 The Common Practice of Memory Isolation

Early memory isolation schemes (e.g., Overshadow [6]) use para-virtualization with shadow page tables so that the hypervisor effectively manages all address mappings. With the advent of CPU and MMU virtualization, most recent systems rely on the access control feature provided by the EPTs. These schemes do not explicitly declare whether they run on the multicore setting or not, except XMHF [35]. We consider a generic procedure in the unicore setting to explain the common practice of memory isolation widely used in the literature, including SecVisor [28], TrustVisor [24], SPIDER [13], InkTag [18], AppShield [9], and Heisenbyte [32]. The primitive consists of the following steps:

- **Step 1.** The hypervisor is instructed to isolate a data page at the virtual address VA_{data} (e.g., via a hypercall) so that VA_{data} can be accessed by the authorized code with permission p and by any unauthorized code with permission $\hat{p}$. For instance, p can be read and write while $\hat{p}$ can be read-only so that unauthorized code cannot modify the page.
- **Step 2.** The hypervisor traverses the present kernel page table to obtain the corresponding guest physical address GPA_{data} and then traverses the EPT to locate the entry δ that maps GPA_{data} to the host physical page HPA_{data}. It sets the permission bits on δ according to $\hat{p}$.
- **Step 3.** At runtime, if the hypervisor determines that the requested access is from the authorized code, it sets the permission bits on δ according to p. When the hypervisor detects that the authorized code execution is to be scheduled off from the CPU (e.g., due to interrupts), it flushes the TLBs and restores permission $\hat{p}$ on δ.

A variant of the preceding method is to switch the mapped physical address inside δ so that different views of physical memory are presented depending on the trustworthiness of the running thread on the core. SPIDER [13] and Heisenbyte [32] are exemplary systems using this approach.

4.3 Complications of Memory Isolation In Multicore Setting

The aforementioned memory isolation procedure runs securely on a unicore platform. However, its vulnerability is exposed in the multicore setting due to parallel execution. A malicious thread on one core can attack the trusted thread on another core. In the following, we present an in-depth analysis of the complications from three perspectives.

4.3.1 Cumbersome EPT Management. As shown in Section 2, each CPU core makes independent accesses to the physical memory. Therefore, the access permission p for the trusted code and the permission $\hat{p}$ for the untrusted code may co-exist in the system, which has two implications.

First, it is no longer sufficient to maintain a single set of EPTs that gives all cores the same access rights. At least two sets of EPTs are needed, with one for the trusted execution and the other for the untrusted. In general, consider a system with n CPU cores and k applications requesting memory isolation. The hypervisor has to maintain $k + 1$ different versions of permission settings. In the worst runtime scenario, the hypervisor has to properly install n distinct EPTs simultaneously. A more complicating issue is that the hypervisor should have an algorithm to detect potential conflicts and/or inconsistency among the permission policies. The workload of managing multiple EPTs not only expands the hypervisor code size, but also significantly complicates its logic.

The second issue is about the switch from the high-privilege permission (e.g., read and write) to the low-privilege permission (e.g., nonaccessible). Privilege downgrade mandates that *all* relevant EPT entries and TLB entries be updated consistently. Our attacks in Section 3 have shown that it requires a deep understanding of the low-level hardware behavior and sophisticated programing skills to make the switch.

4.3.2 Insecure Guest Page Table Checking. In the common practice described in Section 4.2, the trusted and untrusted threads use distinct EPT settings. However, they may use the same guest page tables managed by the guest kernel. Checkoway et al. have proposed the Iago attack [5] to show that the malicious kernel can manipulate the VA-to-GPA mapping to attack memory isolation. On a unicore system, the hypervisor can arguably verify such data structures before entering the isolated environment. After being verified, they are not subject to malicious modification since the guest OS is not running at the same time. Following this approach, InkTag [18], TrustVisor [24], and AppShield [9] have implemented kernel page table verification/protection *when* the hypervisor sets up the isolation environment and *while* the protected thread is in execution.

However, this approach is not secure due to race condition attacks in the multicore setting. Note that the verification of the guest page table cannot be instantly completed. The hypervisor has to walk through the entire guest page table and set the permission bits in the EPT. InkTag, TrustVisor, and AppShield do not enforce core quiesce. The guest kernel and the hypervisor can execute simultaneously. Therefore, a race condition attack can cause the access control policy to be mistakenly enforced on unintended pages while the target pages remain unprotected.

We use the following example to illustrate this. Suppose that a security-sensitive program is just launched and the hypervisor needs to set up its isolated environment. In order to lock and verify the current guest page tables, the hypervisor has to find where they reside. In other words, it has to locate all physical memory pages occupied by the guest page tables so that it is able to configure the corresponding EPT to lock them. Unfortunately, the hypervisor does not have this prior knowledge because the guest page table is priorly managed by the kernel. Therefore, the hypervisor has to traverse all guest page table entries starting from the root pointed to by the `CR3` register in order to find their physical locations. Page table traversal by software is a lengthy operation because it involves a great amount of mapping and memory access operations. 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 but before it is locked. If the traversal is along the ascending order of the address space, the page table pages for the lower end addresses are easier to attack because they are exposed for a longer time interval. In our experiment, it takes around 120,000 CPU cycles to lock the entire page table used by a simple application, which is long enough for the kernel to tamper with one page table entry. As a result, the guest page table locked up by the hypervisor is not the actual one used by the security-sensitive program, which makes it vulnerable to the Iago-like attack.

Note that the core quiesce technique used in XMHF can defeat the aforementioned race condition attack since it stalls all untrusted execution. However, it incurs a remarkably high performance cost. The hypervisor needs to find all physical pages for the guest page table, set the corresponding EPT entry to prevent the kernel's modification, and verify whether the guest page table harbors any poisonous mappings. All guest threads are paused during the hypervisor's processing.

4.3.3 Incapable Thread Identification. Since the trusted threads and the untrusted threads run in parallel, the hypervisor has to differentiate them and apply the appropriate EPTs for their respective execution. Therefore, a prerequisite of secure isolation is to correctly identify the subject that intends to access the protected memory pages.

The subject identity of the security-sensitive program piggybacks on a kernel-level abstraction (e.g., process or thread). A high-level access policy is in the form of "Process X is allowed to read and write page #n; and other processes cannot access." To enforce such a policy in the EPT, the hypervisor maintains the association between the process X and the protected physical pages. Typically, it is implemented using the present the `CR3` register content (e.g., as in TrustVisor [24]) or a combination of `CR3` and the address of the kernel stack (e.g., as in AppShield [9]).

It is a challenging task for the hypervisor to correctly identify the subject requesting memory access. The hypervisor sits underneath the OS and lacks the semantic knowledge of the execution. It is only able to acquire the hardware-related information, such as the instruction pointer stored in the `EIP` register and the page table root address stored in the `CR3` register. Note that interpretation of their application semantics involves kernel objects. For instance, the `EIP` register stores the virtual address of the next instruction to execute. It requires the guest page table managed by the kernel to derive its guest physical address. Since the kernel is untrusted, it is not straightforward for the hypervisor to *correctly* infer the logical representation of the subject from the hardware information.

Overshadow [6] uses the hypervisor-supplied Address Space ID (ASID) and its associated thread context's address to identify the subject. Nonetheless, the guest page table is also involved in storing and retrieving the ASID. Therefore, its security remains weak. In the following, we consider a security-sensitive program P whose data buffer has been isolated by the hypervisor, and we show the impersonation attacks upon `CR3`- and ASID-based identification.

Impersonation Attack. Suppose that the `CR3` register is used to identify the subject. The malicious guest OS can schedule off P and launch a malicious process P' with the same `CR3` content as P's but with different contents in the page table root page. When P' issues a hypercall, the hypervisor mistakenly believes that the subject is P. As a result, P' may exfiltrate the secret of P and tamper with its data. Enclosing kernel-related objects such as the stack address is not secure either because they are still subject to forgery. Suppose that a hypervisor-supplied object such as the ASID is used to identify the subject. Program P needs to store its ASID and to explicitly supply it to the hypervisor in order to access its isolated memory. Nonetheless, this method ends up with the chicken-and-egg dilemma. On one hand, if P's ASID is unprotected, it can be used by P' with the kernel's assistance. On the other hand, if the ASID is restricted to be used by P only, the hypervisor does not have the clue to decide when the ASID is used by P or P', which is also an identification problem.

To summarize, the memory isolation primitive in the multiple core setting is more complicated than it appears. Setting the permission bits on the EPT does not suffice to attain the desired security. The complications are mainly caused by two factors. The first factor is the design defect of the current memory isolation technique used in the literature. The boundary between the trusted and the untrusted is only drawn on the GPA-to-HPA mapping. The VA-to-GPA mappings and the CPU core are still controlled by the adversary, which exposes a large attack surface. The second factor is execution parallelism supported by the multicore platform. When the trusted and untrusted threads run at the same time, the adversary still has the hardware resources to launch attacks. Therefore, the design defect becomes an exploitable vulnerability.

5 FULL ISOLATION ON MULTICORE PLATFORMS

In this section, we propose a new isolation paradigm to enclose an entire computing environment including the CPU core, the memory region, and (optionally) the needed peripheral device(s). Our new system is named *fully isolated micro-computing environment* or FIMCE. It not only avoids all aforementioned security pitfalls of virtualization-based memory isolation, but also has several advantages over other memory isolation systems.

5.1 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 while 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 of a set of *pillars*, as well as the security task itself. A pillar is in essence a self-contained library that 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 are 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. First, it avoids the same pitfall as in the existing memory isolation primitive. Second, it prevents an untrusted OS from interfering with the FIMCE by using 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 deprives the kernel of the chance to influence the mappings used inside the FIMCE. Hence the address-mapping and TLB-related issues discussed in Section 4 do not arise.

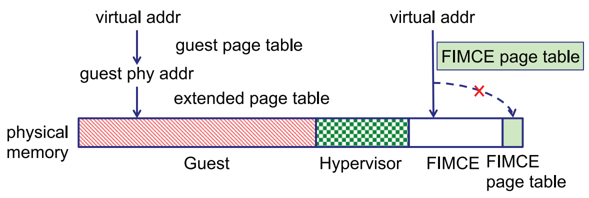


Fig. 3. Memory isolation for FIMCE without EPT.

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 3 explains the memory setting for a FIMCE.

When launching the 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 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 interpretations of the same paging structure which may have loopholes allowing the FIMCE code to breach memory isolation.) Last, the hypervisor configures Input-Output Memory Management Unit (IOMMU) page tables to prevent illegal Direct Memory Accesses (DMAs).

I/O Device Isolation. We utilize 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. First, 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.

Second, 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 to other cores of the guest.

5.2 The Lifecycle of FIMCE

When the platform is powered on, its Dynamic Root of Trust for Measurement (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 completes its job, it issues another hypercall within the FIMCE to request FIMCE shutdown. We describe here the main procedures the hypervisor performs during FIMCE launch, runtime, and termination.

5.2.1 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 Interrupt ReQuests (IRQs) 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 it is not accessible from the guest. To set up 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 our design because disk data can easily be protected by cryptographic means.

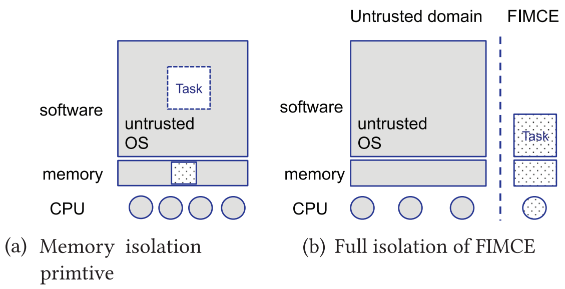
Environment Initialization. After all hardware resources are allocated, the hypervisor sets up the FIMCE environment for the task's code to run inside. First, the hypervisor initializes a minimum set of data structures required by the hardware architecture. These include a Global Descriptor Table (GDT) with a code segment descriptor, a data segment descriptor, and a task-state segment (TSS) descriptor. We use 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 elevating 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 either because each FIMCE instance is independent of others 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. Last, 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 7.

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.

5.2.2 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 nonmaskable interrupts. All software exceptions and critical hardware interrupts are considered system failures and trigger the hypervisor to terminate the FIMCE. Other interrupts are handled by the empty handlers by default,



(a) Memory isolation primtive (b) Full isolation of FIMCE

Fig. 4. 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.

which simply return unless their corresponding handlers are installed during FIMCE launch (as part of a pillar).

5.2.3 Termination. A FIMCE can be shut down due to the security task's termination hypercall or due to system failure interrupts. To turn off a FIMCE, the hypervisor zeros its memory partition and invalidates the TLBs. It then switches the current VMCS to the one used for the guest OS and reenables the EPT on the current core. It undoes any device assignments and returns them to the OS. Last, it notifies the OS that the core is returned.

5.3 Comparisons to Memory Isolation Primitive

Figure 4 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.

Compared to domain isolation [16] and memory isolation, FIMCE possesses their virtues without their drawbacks. It has a 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 [16]. FIMCE also achieves page-level granularity, as in a memory isolation scheme (e.g., TrustVisor [24]). 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 allow the guest OS to do so. For example, the guest OS can extract sensitive information via page swapping, as demonstrated in Xu et al. [37]. 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 Section 4. The design of FIMCE also avoids the address space layout verification used in TrustVisor [24] and InkTag [18], which is vulnerable to race condition attacks.

6 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 remains 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 [4] rely on cryptographic techniques with a high performance toll. In the following, we compare SGX and FIMCE from various aspects and then explore potential integration.

6.1 Comparisons

FIMCE and SGX are designed with different goals. SGX offers 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 [11]. 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 are also encrypted by the hardware when they are written to EPC pages. It is hence secure against bus snooping attacks. FIMCE isolation leverages hardware-assisted virtualization techniques. In addition to 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 has stronger security from this perspective.

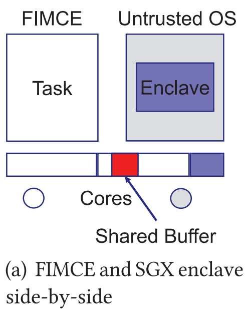
Autonomous Execution. SGX and FIMCE give different treatments to CPU scheduling of the isolated program. SGX leaves thread scheduling inside the enclave to the OS. This design decision allows for a page fault attack [37], wherein the guest OS interrupts the enclave execution and extracts sensitive data.

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. Page faults inside the FIMCE are handled internally. It is hence a highly autonomous system with no runtime dependency on the OS.

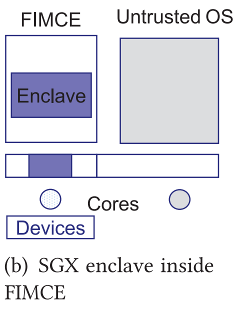
I/O Capability. SGX does not support I/O operations. Secure data exchange between an enclave and the outside world becomes a challenging task. As shown by Haven [4] and SCONE [1], a middle layer between the isolated application and the OS is introduced to ensure data confidentiality and authenticity 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 [1], 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 in an exclusive manner, cryptographic protection is no longer needed. Note that context switch costs are also saved as no context switch is required inside the FIMCE.

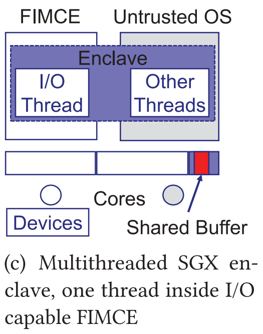
Verifiable Launch. SGX and FIMCE also differ in controlling the launch of the isolated environment. Although SGX supports attestation, it does not enforce policies for enclave launch. The permission to launch an enclave is decided by the untrusted OS. In FIMCE's design, the trusted



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



(b) SGX enclave inside FIMCE



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

Fig. 5. FIMCE-based isolated I/O for SGX enclaves.

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.

6.2 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, we discuss possible ways to combine them.

Share Memory Between Enclave and FIMCE. A naive integration is to run the SGX enclave and the FIMCE side by side (as in Figure 5(a)) with a shared memory buffer used for intercommunication. Unfortunately, it is hard to secure FIMCE-enclave data exchange in this fashion.

We observe that 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 Section 4. 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 5(b)). The security benefits of the combination are twofold. As compared to SGX isolation alone, the adversary has no further 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.[2] 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 earlier, 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 5(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 5(c).

The idea is to leverage SGX multithread support. The main thread protected 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 5(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 belonging 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 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 OS scheduling.

7 MODULARIZED SOFTWARE INFRASTRUCTURE

It is widely recognized that existing memory isolation schemes require the protected task to be self-contained. In contrast, FIMCE has inborn support for dynamically setting up the software infrastructure (e.g., libraries, drivers, and interrupt handlers) to cater to the task's needs. We propose using a structured method 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 a pillar and how to check the correctness of linking without significantly increasing the hypervisor's code size.

7.1 Pillars

Pillars provide services that are otherwise not easily available to the security task due to the absence of an 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. First, a pillar must be self-contained. The code of a pillar function does not depend on any code outside of the pillar. Second, the ELF format of a pillar has a new section, called a *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 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.

7.2 Pillar Verification and Linking

Once the needed pillars and the security task are loaded into the FIMCE, the hypervisor passes 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.

7.2.1 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 time, the dynamic loader is 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.

7.2.2 Dynamic Linking. Although the kernel has the capability of linking pillars with the security task, we 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.

We 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 it returns to the security task after execution.

8 APPLICATIONS OF FIMCE

In addition to isolation, FIMCE can be applied to other types of applications. In the following, we discuss two new types of applications. 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.

8.1 Malleability

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

To demonstrate the benefit of malleability, we 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 binding to the isolated environment. The difficulty lies in how to authenticate the thread that requests entry 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 our adversary model, the default locality-based access control is not adequate.

Since the application cannot hide any secret in the unprotected memory from the OS, both have equal knowledge and capability in terms of presenting authentication information to the hardware. One may suggest leveraging the hypervisor to perform authentication, as shown in McCune et al. [24]. However, as we have analyzed in Section 4.3, it is challenging for the hypervisor to securely authenticate the application without sufficient knowledge of kernel-level semantics.

With a malleable environment, FIMCE offers an elegant solution. The hypervisor uses TPM Locality 2 and assigns the OS with Locality 0 and the code inside a FIMCE with Locality 1. During bootup, 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 bootup, 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 our method is that the hypervisor does not hold any secret and is oblivious to the application's logic and semantics. In addition to the stronger security bolstered by the hardware, it has a small TCB and supports process migration.

8.2 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 a 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, we sketch out two systems here. 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 [36] and Guardian [7]. 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 our 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 [21, 24, 27] focus on a 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 [14, 15, 19, 30]. 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.

We observe that 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 resume it afterward. Hence, the attestation does not reflect the genuine state of the kernel.

9 EVALUATION

In this section, we first discuss the security of FIMCE by a comparison with memory isolation. We then report our prototype implementation as well as the benchmarking and experimental results.

9.1 Security Analysis

It remains as an open problem to formally prove the security of a system design (not implementation). Therefore, the security analysis given here is informal. We first argue that the multicore complications plaguing memory isolation systems are not applicable to our design. We then evaluate FIMCE security based on its attack surface and TCB size.

Complication Free. Recall that Section 4.3 enumerated three security complications due to the multicore setting: namely, complex EPT management, insecure guest page table checking, and incapable thread identification. Since the FIMCE is a fully isolated environment, its design does not face these three complications.

- 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 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 4.3.
- Memory isolation schemes need subject identification to choose the proper EPT setting. This challenging problem does not exist in our 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 execution without being preempted by other threads under the OS's control. In the 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 reactivated from the saved state. Therefore, subject identification is not needed.

Reduced Attack Surface. The malicious kernel in memory isolation systems enjoys a large attack surface as it has full control over the CPU cores and the VA-to-GPA mapping, which leads to the various attacks and design complications described in Sections 3 and 4. 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 or 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. We acknowledge that the protected code is subject to such attacks if no proper input checking is in place. However, it is out of scope of our work to sanitize inputs.

Strength of Full Isolation. 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 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

Table 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

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, our hypervisor has a tiny code base with around 6,000 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 helps simplify concurrency handling.

9.2 Implementation

We have implemented a prototype of FIMCE on a desktop computer with an Intel Core i7 2600 quad-core processor running at 3.4GHz, with a Q67 chipset, 4GB of DDR3 RAM and a TPM chip. The platform runs a Ubuntu 12.04 guest with the stock kernel version 3.2.0-84-generic. We have implemented the FIMCE hypervisor of around 6,000 Source Lines 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. We slightly modify Intel's open source TXT bootloader *tboot*[3] to load our hypervisor. The modification is to make the bootloader jump to the FIMCE hypervisor's entry at the end of the bootup 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 OS kernel initialization, all cores are set to use the same set of EPTs, thus ensuring a uniform view of the memory.

To showcase the applications of FIMCE, we have also developed three pillars: a 7KB serial port driver pillar that supports keyboard I/O, a comprehensive crypto pillar of 451KB size based on the *mbed* TLS library,[4] and 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 1, the user space FIMCE library provides two interfaces and three macros. The two interface functions are used to start and stop the FIMCE,

Table 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 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

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 fromthat in the OS.)

Internals of `start_FIMCE()`. This user space function runs in two steps in the guest domain. First, 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 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 *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.

9.3 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, we choose multithreaded *SPECint_rate 2006* and *kernel-build* as well as single-threaded *lmbench*, *postmark*, and *netperf* as the performance benchmarks. We run them on top of the OS without any FIMCE running and then repeat the evaluation with an infinite loop as the security task in the FIMCE.

For the multithreaded *SPECint_rate 2006*, we set the concurrency level to four. Figure 6 shows that it has a 15% percent performance drop on average due to the presence of FIMCE. In the kernel-build experiment, we compile the Linux kernel v2.6 using the default configuration with four-level concurrency. The results are reported in Table 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 (25% in our setting).

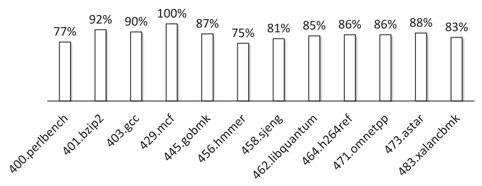


Fig. 6. SPECint_rate 2006 results. The numbers are the percentage of the score with FIMCE compared to the score without FIMCE.

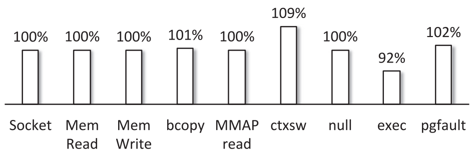


Fig. 7. Lmbench results. The numbers are the percentage of the score with FIMCE compared to the score without FIMCE.

Table 4. Single-threaded Postmark Performance with and without FIMCE Running (in Seconds)

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

Table 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

To verify our estimation that FIMCE does not incur much performance cost for single-threaded applications, we run Lmbench, Netperf, and Postmark with and without FIMCE. Figure 7 shows that most tasks of *Lmbench* are not affected by FIMCE, except that one task has an 8% performance drop. Similar results are also found for Netperf (Table 3) and Postmark (Table 4).

9.4 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 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.

On average, a null hypercall on our platform takes 0.31 milliseconds. 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 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 our platform, every 4KB memory read and write cost about 2μs.

Pillar loading also involves integrity verification. Our 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 from 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.

9.5 Application Evaluation

We have implemented four use cases 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

Table 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

dispersed code base and is interactive with devices. Driverguard [8] relies on manual driver code instrumentation, which is tedious and error-prone. TrustPath [41] 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, and this 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.

We implement a program that reads the user's password inputs from the keyboard, converts it to the decryption key, and then performs an Advanced Encryption Standard (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. We run the program with FIMCE protection for 100 times. On 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, we utilize FIMCE to harden a Secure Sockets Layer (SSL) web server by isolating its RSA decryption of SSL handshakes. As noted previously, existing systems [6, 24] on Apache protection are 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 separate thread in parallel with others.

In the experiment, we customize the Apache source code 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.

We connect our server to a Local Area Network (LAN) and run ApacheBench with different concurrency levels. The Apache server hosts a Hypertext Markup Language (HTML) page of 500KB. We compare it 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 6.

It is evident that at a 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 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 [24], InkTag [18], and Overshadow [6] are not affected by concurrency, albeit they are *not* secure in a multicore system.

Table 7. Overhead of Other Protection Schemes (Numbers Are Excerpted from Respective Paper)

Schemes	Overhead
TrustVisor [24]	9.7% to 11.9% depending on concurrent transaction
InkTag [18]	2% in throughput, 100 concurrent request
Overshadow [6]	20% to 50% on a 1Gbps link, 50 concurrent request

Table 8. TPM Performance (in Seconds)

	TPM Seal	TPM Unseal
Guest	0.54	0.96
FIMCE	0.41	0.94

However, we remark that the design of FIMCE can certainly be extended to support concurrent FIMCE instances at the expense of more cores dedicated for security. We also note that, in real-world web transactions, the time spent for RSA decryption accounts for a much smaller portion of the entire transactions as compared to that 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, and (3) more time needed to generate or locate the needed web pages. Therefore, we expect the performance loss of using FIMCE for a real web server does not appear as discouraging as in our experiments.

Long-Term Secret Protection. We demonstrate the malleability of FIMCE architecture via the long-term secret protection application. We implement a program 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`. We measure the time taken by the TPM seal and unseal operations inside the FIMCE and compare it with the baseline experiment wherein the TPM pillar runs as a kernel module. The results of protecting a 20-byte-long secret are in shown Table 8.

FIMCE shows slight speedup 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 its simple structure, FIMCE does not have such overhead.

Compared with existing approaches that virtualize the TPM using software, such as in McCune et al. [24], our 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. We implement a program for kernel introspection. 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 $3.04\mu s$ to read a kernel object, which is comparable with the time (around $3.11\mu s$) needed by the kernel itself. Our introspection system is more efficient than existing schemes, such as Fu and Lin [15], because it runs natively on the hardware in the same fashion as running inside the kernel. According to our 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.

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 our 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 on average to perform the entire procedure, including the time for TPM quote operation.

10 RELATED WORK

Virtualization-Based Security. Our work is directly related to virtualization-based security systems. The immediate benefit of virtualization is that the resources of a platform can be partitioned such that two virtual domains cannot interfere with each other. Following this idea, TERRA [16] and Proxos [31] were proposed to partition a system into a trusted domain and an untrusted domain, where critical applications run in the former while others run in the latter. Although this coarse-grained approach is effective and easy to implement, its security is undermined by the large TCB as it encloses the operating system, which is widely regarded as vulnerable to attacks.

With the development of hardware techniques, the current mainstream commodity platforms enjoy hardware support for CPU, memory, and I/O virtualization. By taking advantage of a bare-metal hypervisor (a.k.a. virtual machine monitor or VMM), various systems [26, 28, 41] have been proposed in the literature for various security purposes. Despite different designs, the fundamental building block commonly used by them is the hypervisor's capability of regulating the guest VM's memory accesses by properly setting permission bits in relevant page table entries.

Two typical examples of kernel protection are SecVisor [28] and Lares [26]. The former proposed a mechanism to use the hypervisor to protect kernel integrity while the latter monitors and analyzes events in kernel space by inserting hooks into arbitrary kernel locations. Both use the hypervisor to prevent the kernel code from being modified. TrustPath [41] and Driverguard [8] were proposed to protect the I/O channel between a peripheral device and an application.

Hypervisor-based memory access control also allows for memory isolation, a technique widely used to set up a secure execution environment to protect data security and execution integrity of sensitive code against an untrusted OS. TrustVisor [24] is a tiny hypervisor that builds such an environment for a self-contained Piece of Application Logic (PAL). It further enhances the environment with a software-implemented TPM (called μTPM) in the hypervisor space. μTPM protects the PAL's long-term secret and allows for remote attestation. Since the PAL is required to be self-contained, TrustVisor is not an ideal solution to protect complex tasks that involve I/O operations or that depend on libraries with large code bases. Based on TrustVisor, XMHF [35] provides an open-source hypervisor framework providing security functionality including memory protection. Also based on TrustVisor, Minibox [22] combines the hypervisor with Native Client [39] to provide a two-way sandbox for the cloud. Taking the idea further, SeCage [23] isolates multiple compartments that comprise code, data, and secrets inside an application. It also provides a method to automatically generate such isolation compartments. Another line of research is to protect the entire application. Overshadow [6], InkTag [18], and AppShield [9] are exemplary works in this category which are capable of isolating a whole application from the untrusted OS. In both Overshadow and InkTag, the memory regions isolated for the application are encrypted when the OS takes control. While Overshadow and AppShield are mainly designed for application data secrecy and integrity, InkTag is concerned about verifying OS behaviors by using the paraverification technique which mandates changes on the kernel's behavior. A common challenge for isolating an application is to handle the system calls. All three schemes require intensive work on system call adaption and parameter marshalling. Different from the coarse-grained cross-VM isolation used in Terra [16], these systems provide fine-grained in-VM isolation. Unfortunately, as shown earlier, their security hardly holds under a multicore setting.

Isolation with Other Techniques. Flicker [25] 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 SGX [20] 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 [2], a security monitor that resides in the secure world established by an ARM TrustZone can protect the OS kernel in the normal world at runtime. Virtual Ghost [12] 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 [34] creates an isolated execution environment on Graphics Processing Units (GPUs). Being an isolated device from the CPU with its own memory, the GPU provides a natural ground for building an isolated execution environment. In the past, programming on the GPU was difficult because of its highly specialized hardware. However, modern GPUs are becoming increasingly more programmable so that running code for execution on the GPU is easier. Nonetheless, this approach still requires significant development effort because there is little support from current systems. SICE [3] 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.

11 CONCLUSION

To conclude, we show that the existing virtualization-based memory isolation primitive is ineffective in the multicore setting. We propose FIMCE, a stronger and tidier isolation primitive for multicore systems. FIMCE places the protected task into a fully isolated computing environment where neither hardware nor software resources are accessible to any code in the untrusted domain. Our design features strong security with a reduced attack surface and great nimbleness and versatility. We have implemented FIMCE and experimented with several test cases, demonstrating various advantages over alternative techniques. The performance overhead ranges from zero to

1% on single-threaded applications and is bounded by the reduction of the core-level concurrency on multithreaded applications.

Our future work aims to extend FIMCE along several directions. We plan to substantiate and implement the design of integrating FIMCE and SGX. We will also explore more security applications of FIMCE. One plausible topic is to use FIMCE as a runtime trust anchor to enforce access control policies in the untrusted kernel (e.g., to regulate/sanitize disk file operations).

ACKNOWLEDGMENTS

We are grateful to Virgil Gligor for his constructive suggestions.

REFERENCES

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux containers with intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 689–703.

[2] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. 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 (CCS'14)*. ACM, New York, 90–102. DOI: http://dx.doi.org/10.1145/2660267.2660350

[3] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 2011. 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 (CCS'11)*. ACM, New York, 375–388. DOI: http://dx.doi.org/10.1145/2046707.2046752

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[5] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, 253–264. DOI: http://dx.doi.org/10.1145/2451116.2451145

[6] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. 2008. 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'08)*. ACM, New York, 2–13. DOI: http://dx.doi.org/10.1145/1346281.1346284

[7] Yueqiang Cheng and Xuhua Ding. 2013. Guardian: Hypervisor as security foothold for personal computers. In *Proceedings of the International Conference on Trust and Trustworthy Computing*, Michael Huth, N. Asokan, Srdjan Čapkun, Ivan Flechais, and Lizzie Coles-Kemp (Eds.). Springer, Berlin, 19–36.

[8] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. 2011. DriverGuard: A fine-grained protection on I/O flows. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS)*, Vijay Atluri and Claudia Diaz (Eds.). Springer, Berlin, 227–244.

[9] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. 2015. Efficient virtualization-based application protection against untrusted operating system. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS'15)*. ACM, New York, 345–356. DOI: http://dx.doi.org/10.1145/2714576.2714618

[10] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, Denver, CO, 565–578.

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *IACR Cryptology ePrint Archive* (2016), 86.

[12] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. 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'14)*. ACM, New York, 81–96. DOI: http://dx.doi.org/10.1145/2541940.2541986

[13] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. SPIDER: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*. ACM, New York, 289–298. DOI: http://dx.doi.org/10.1145/2523649.2523675

[14] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceeding of the 2011 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 297–312. DOI: http://dx.doi.org/10.1109/SP.2011.11

[15] Yangchun Fu and Zhiqiang Lin. 2012. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 586–600.

[16] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. 2003. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, 193–206. DOI: http://dx.doi.org/10.1145/945445.945464

[17] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. 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'12)*. ACM, New York, 411–422. DOI: http://dx.doi.org/10.1145/2150976.2151020

[18] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. 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'13)*. ACM, New York, 265–278. DOI: http://dx.doi.org/10.1145/2451116.2451146

[19] Hajime Inoue, Frank Adelstein, Matthew Donovan, and Stephen Brueckner. 2011. Automatically bridging the semantic gap using C interpreter. In *Proceedings of the 2011 Annual Symposium on Information Assurance*. University at Albany, State University of New York (SUNY), Albany, NY, 51–58.

[20] Intel Corporation. 2013. Innovative Instructions and Software Model for Isolated Execution. Retrieved from http://privatecore.com/wp-content/uploads/2013/06/HASP-instruction-presentation-release.pdf.

[21] Trent Jaeger, Reiner Sailer, and Umesh Shankar. 2006. PRIMA: Policy-reduced integrity measurement architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT'06)*. ACM, New York, 19–28. DOI: http://dx.doi.org/10.1145/1133058.1133063

[22] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. 2014. MiniBox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 409–420.

[23] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, New York, 1607–1619. DOI: http://dx.doi.org/10.1145/2810103.2813690

[24] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 143–158.

[25] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for Tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys'08)*. ACM, New York, 315–328. DOI: http://dx.doi.org/10.1145/1352592.1352625

[26] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 233–247.

[27] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. 2004. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium*. USENIX Association, Berkeley, CA.

[28] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, 335–350. DOI: http://dx.doi.org/10.1145/1294261.1294294

[29] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. 2009. 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'09)*. ACM, New York, 121–130. DOI: http://dx.doi.org/10.1145/1508293.1508311

[30] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. 2015. Exploring VM introspection: Techniques and tradeoffs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15)*. ACM, New York, 133–146. DOI: http://dx.doi.org/10.1145/2731186.2731196

[31] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 279–292.

[32] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, New York, 256–267. DOI: http://dx.doi.org/10.1145/2810103.2813685

[33] Cheng-Chun Tu, Michael Ferdman, Chao-tung Lee, and Tzi-cker Chiueh. 2015. A comprehensive implementation and evaluation of direct interrupt delivery. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'15)*. ACM, New York, 1–15. DOI: http://dx.doi.org/10.1145/2731186.2731189

[34] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. PixelVault: Using GPUs for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*. ACM, New York, 1131–1142. DOI: http://dx.doi.org/10.1145/2660267.2660316

[35] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2014. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 430–444.

[36] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig. 2009. Lockdown: A safe and practical environment for security applications. *CMU-CyLab-09-011* 14 (2009).

[37] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 640–656.

[38] Jisoo Yang and Kang G. Shin. 2008. 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'08)*. ACM, New York, 71–80. DOI: http://dx.doi.org/10.1145/1346256.1346267

[39] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 79–93.

[40] Siqi Zhao and Xuhua Ding. 2017. On the effectiveness of virtualization based memory isolation on multicore platforms. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, Washington, DC.

[41] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. 2012. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 616–630.

[42] Zongwei Zhou, Miao Yu, and Virgil D. Gligor. 2014. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, Washington, DC, 308–323.