4-2017

On the effectiveness of virtualization based memory isolation on multicore platforms

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

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

Citation

On the Effectiveness of Virtualization Based Memory Isolation on Multicore Platforms

Siqi Zhao
School of Information Systems
Singapore Management University
siqi.zhao.2013@smu.edu.sg

Xuhua Ding
School of Information Systems
Singapore Management University
xhding@smu.edu.sg

Abstract—**Virtualization based memory isolation has been widely used as a security primitive in many security systems. This paper firstly provides an in-depth analysis of its effectiveness in the multicore setting; a first in the literature. Our study reveals that memory isolation by itself is inadequate for security. Due to the fundamental design choices in hardware, it faces several challenging issues including page table maintenance, address mapping validation and thread identification. As demonstrated by our attacks implemented on XMHF and BitVisor, these issues undermine the security of memory isolation. Next, we propose a new isolation approach that is immune to the aforementioned problems. In our design, the hypervisor constructs a fully isolated micro computing environment (FIMCE) that exposes a minimal attack surface to an untrusted OS on a multicore platform. By virtue of its architectural niche, FIMCE offers stronger assurance and greater versatility than memory isolation. We have built a prototype of FIMCE and measured its performance. To show the benefits of using FIMCE as a building block, we have also implemented several practical applications which cannot be securely realized by using memory isolation alone.**

1. Introduction

Recent advances of hypervisor-based security systems [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] against a subverted kernel are by and large attributed to the memory isolation feature introduced by modern hardware virtualization technology. The cornerstone of memory isolation is the hypervisor's privilege and capability of specifying the access permissions to physical page frames. To prevent a physical memory region from being illicitly accessed by untrusted threads (in particular, of kernel privilege), the hypervisor is instructed to configure the so-called Extended Page Tables (EPTs) on an Intel's x86 platform[1]. The permissions in the EPT supersedes the access permissions declared in the guest page table managed by the guest kernel.

For instance, TrustVisor [7] uses the EPT to isolate memory regions used by a sensitive security task. InkTag [9] and AppShield [10] isolate the entire address space of

---

[1]. The equivalent data structure is called the Nested Page Tables on AMD's processors and called the Stage 2 translation on ARM processors.

a process also using EPT. Lares [12] protects its hooks in guest kernel by setting EPT entries. SecVisor [8] ensures lifetime kernel integrity via setting access permissions in the EPT. BitVisor [4] manages guest's device access by intercepting such access by EPT. On ARM platforms, OSP [11] constructs an efficient TEE using Stage 2 translation in cooperation with TrustZone.

Most existing systems [7], [10], [9], [8] built on memory isolation implicitly assume a unicore hardware platform. While XMHF [13] and OSP [11] explicitly described their mechanisms on a multicore setting, in their designs essentially a single-threaded execution model is forced onto a multicore platform. Although the unicore assumption simplifies the design and verification of the systems, it is increasingly distant from recent real-world hardware platforms. Even many low-end mobile phones are equipped with a dual-core processor, not to mention desktop computers and high-performance servers. Running multiple CPU cores in the platform has a non-negligible impact on security, because the adversarial thread on one core can attack a trusted thread executing on another core, which is infeasible in a unicore setting. Hence, there exists an urgent need to carefully re-evaluate memory isolation security on multicore systems.

In this work, we conduct an in-depth analysis of memory isolation in the multicore context. After scrutinizing both the hardware architecture and the system design, we conclude that the memory isolation method adopted in the literature is *ineffective* on a multicore platform. This is evident from the attacks we have schemed and implemented on XMHF [13] and BitVisor [4], respectively. These two attacks allow the rootkit in the guest to bypass the permission checks enforced by the EPT. The analysis shows that memory isolation in the multicore setting faces several intractable security issues whose solutions (if exist) take a heavy toll on the hypervisor's code size and performance.

The deep-seated reason of ineffectiveness lies in the fact that memory isolation is not complete in that the physical memory is only one of several types of resources involved in code execution. From this perspective, we propose to construct a *Fully Isolated Micro Computing Environment* (FIMCE) as a new building block to construct secure systems on multicore platforms. A FIMCE is fully isolated

from, and yet tightly-coupled with, the untrusted operating system. Owing to its architectural niche, FIMCE offers strong security and great versatility. Specifically, it allows for I/O operations, has a malleable hardware setting, and can run continuously together with the OS. Because of these features, FIMCE is a more useful and powerful security primitive than memory isolation. We have demonstrated how to use it to tackle certain challenging problems, e.g., runtime kernel attestation. We have implemented a prototype of FIMCE with its micro-hypervisor. We have also measured its performance by running benchmarks and built four applications on top of it to showcase its versatility. The experimental results confirm that FIMCE's strong security and high usability are at the cost of performance since the OS has one less CPU core to use when the FIMCE is in operation.

In short, we make the following contributions in this paper:

1) We show that the widely used virtualization based memory isolation method is not effective in multi-core platforms. Our assertion is supported by concrete attacks on XMHF and BitVisor and an in-depth and general examination on design defects of memory isolation.

2) Based on the analysis of memory isolation short-comings, we propose a new isolation primitive called FIMCE. The new scheme provides stronger security than memory isolation. Moreover, it has a much wider application spectrum attributing to its unique architectural niche.

3) We implement a prototype of FIMCE and use it as a building block to construct several applications to showcase its valuable features including the modularized software infrastructure, I/O support, the malleable hardware setting and runtime trust anchor. We measure its performance and argue that the security gain outweighs the system's performance loss incurred by the core removal.

ORGANIZATION. In the next section, we explain the background of address translation and memory access 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. Section 6 evaluates the security and performance based on experiments with a prototype of FIMCE. We discuss the related work in Section 7 and conclude the paper with Section 8.

2. Background

This section explains the memory access paradigm on a symmetric multiprocessor (SMP) system with hardware-assisted virtualization. A virtual address (VA) is firstly translated into a Guest Physical Address (GPA) by using the guest page table managed by the kernel. A GPA is then mapped to a Host Physical Address (HPA) by using the Extended Page Table managed by the hypervisor. In the SMP setting, each CPU core independently accesses the shared main memory. Each core has its own MMU to walk through the guest page table and the EPT. Typically, a common set of EPTs are used globally by all CPU cores. This approach avoids synchronization of EPT updates and improves efficiency. Each CPU core has its own `CR3` register pointing to the guest page table in use. They use the same or different guest page tables depending on the kernel's scheduling. To reduce the latency due to two-level address translation, each core is equipped with its own independent Translation Lookaside Buffer (TLB), which caches recently used entries in the guest page table and the EPT. Note that both address mapping and the associated access permission are stored in the TLB. To access a virtual address from a core, its TLB is looked up first. If a TLB entry is hit, the corresponding page table is not used during this translation. We use Figure 1 to illustrate the paradigm.
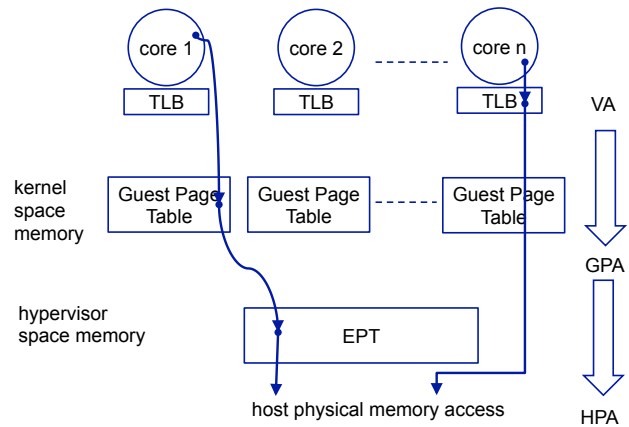

Figure 1. The paradigm of memory access in an SMP setting. Core 1 has TLB misses and accesses the memory via two page tables whereas Core n has TLB hits and accesses the memory without consulting any page table.

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

Moreover, the hardware does not enforce coherence among the TLBs on different cores. All such operations need to be explicitly carried out by software as well. When more than one core accesses an address space, the core that changes the mapping is supposed to perform *TLB shootdown* to invalidate any existing entries on other cores. Typically, it is achieved by using the Interprocessor Interrupts (IPIs) for this purpose. Specifically, the initiating core fires an IPI to each core that needs to invalidate its TLBs. On modern x86 platforms, the Advanced Programmable Interrupt Controller (APIC) interfaces with the bus for receiving and sending IPIs. The IPI is received by the other cores and treated exactly the same way as an external interrupt. A handler is invoked and the specified TLB entry is invalidated. In this way, the consistent view of the address space is maintained across all CPU cores.

3. Attacks

As a prologue to our in-depth analysis and the proposal, we present concrete attacks on two open-source micro-hypervisors running on a PC with multiple cores. The success of the attacks indicates that memory isolation on the multicore setting requires a meticulous checking of all implementation details.

Consider a typical isolation scenario where a security sensitive application requests the hypervisor to protect a given page by setting its access permission as, for example, read-only in the EPT. The objective of the attack is for the malicious OS to successfully access the protected page without complying to the EPT permission setting. The general idea behind the attacks is to use a stale TLB entry. Obviously, if a CPU core still stores the TLB entry for the protected page after the change on the EPT, the core can access the page with the priorly granted permissions. Although the TLB invalidation is widely known as an indispensable step, it is not trivial to implement it without any flaw.

3.1. Stifling Attack

In order to invalidate the TLBs in other cores, the hypervisor has to preempt the running threads on those cores. A typical way of communicating with other cores is the IPI. For this purpose, the hypervisor sets the External-interrupt Exiting bit in all logic cores' VMCS structures during system initialization. After configuring the EPT to protect a page specified by an application, the hypervisor broadcasts an IPI which triggers VM-exit in all receiving cores.

The *stifling attack* is to prevent a CPU core under the malicious OS's control from responding to the hypervisor's IPI so that this core's TLB is not invalidated by the hypervisor. The attack exploits a hardware design feature to block all maskable external interrupts, including IPI. On x86 platforms, the IPI handler is expected to perform a write to the `EOI` register in the local APIC before executing `iret`. This operation signals the end of the current IPI 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. Note that using the interrupt masking bit (namely `EFLAGS.IF`) cannot achieve the same malicious goal, because it is overridden by the External-interrupt Exiting bit in the VMCS set by the hypervisor. Also note that setting the External-interrupt Exiting bit does not mandate automatic EOI write by hardware, it is always the software's responsibility to perform such write, by either the hypervisor or the OS.

Given this observation, we scheme the attack to bypass the EPT's access control. Suppose that the victim application occupies core_v while the kernel runs in two threads in core_1 and core_2 respectively. The steps of the attack are described below in the temporal order with a visualization in Figure 2.

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 its TLB. It also disables interrupt and preemption so that it is not scheduled off from core_1 in order to avoid any TLB invalidation.
3) At core_2: Another thread of the malicious kernel sends an IPI to core_1 by using an 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 buffer without trigger any EPT violation exception, because the core's MMU uses the EPT entry in the TLB with the permissions which are set prior to the hypercall.
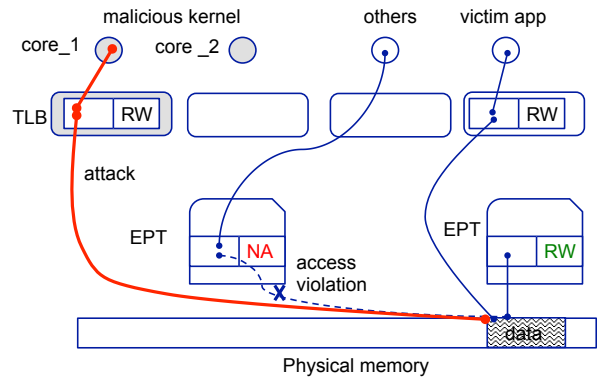

Figure 2. Illustration of the stifling attack bypassing the EPT's access control over the victim's data. Grey regions are controlled by the attacker.

We have implemented the attack on top of the BitVisor [4] with slight changes on its EPT management function and interrupt handling. The experiment shows that the kernel successfully continues its write access to the target page without triggering any page fault, after the page's permission in the EPT is changed into read-only.

One possible solution to the stifling attack is to virtualize local APICs so that the hypervisor intercepts the external interrupts and enforces EOI writes. However, this approach is against the goal of performance optimization proposed in [14], [15] which advocate removing the hypervisor from the code path handling interrupts. Moreover, it is against the purpose of using a tiny hypervisor as it increases the hypervisor's code size and complexity.

3

An alternative is to resort to non-maskable interrupts (NMIs) instead of IPIs. NMIs are delivered immediately by the local APIC to the CPU core as they are usually sent by hardware such as watchdogs to indicate critical hardware failure which needs immediate attention. However, it is strongly discouraged to use software to generate NMIs because of its complex handling. Moreover, it requires a high level of expertise to implement a proper NMI handler [16] because it needs to deal with recursive execution. Briefly speaking, once an NMI is delivered to a core, subsequent NMIs are blocked until the core executes `iret`. If the NMI handler causes any exception, the exception handler's `iret` immediately allows the next NMI to be delivered while the present one is still in processing. From the system perspective, it has risks to use the hypervisor to issue and handle NMIs.

3.2. VPID Attack

XMHF [13] is a micro-hypervisor on x86 platforms that explicitly takes the multicore setting into the design consideration. In fact, XMHF [13] enforces a *single threaded* execution model for the hypervisor. When one core traps to the hypervisor space, it "quiesces all other cores" by broadcasting an NMI which causes 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, although XMHF has been formally verified.

The early hardware virtualization flushes the TLB on VM Exits, because the hypervisor uses an independent address space from that of the guest. Since this approach incurs a great overhead due to TLB misses after the VM (re)entry, more recent x86 processors introduce a feature called Virtual Processor ID (VPID). It assigns identifiers to address spaces of each virtual core and of the hypervisor and also tags TLB lines with the respective identifiers. When an TLB entry is hit during translation, it is valid only when its VPID tag matches the VPID of the present address space. As a result, a VM exit does not imply the invalidation of all TLB entries of a core.

This performance improvement unfortunately dents the security of the hypervisor. Since not all TLBs are evicted by the hardware during a trap to the hypervisor, the stale entries must be 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 during the handling the quiesce-NMI. Out of this observation, we devise the following attack by which the guest OS can continue its access to a page despite the access policy update at the EPT.

1) At core_v: The victim application starts execution, it allocates a page which expects memory isolation.
2) At core_1: The malicious kernel running on core_1 maps the buffer into its own space, reads it once so that an TLB entry is loaded by the MMU. It

disables interrupt and preemption so that the TLB entry is not evicted.
3) At core_v: The victim application performs a hypercall to the XMHF hypervisor. The hypervisor issues an NMI used to quiesce other cores and clears the access permission for the page in the EPT after all other cores are paused.
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 still exists. The guest OS continues to read and write the isolated buffer, regardless what the permission on the EPT is set.

We have implemented the attack on the latest version of XMHF on an Intel platform. As a proof of concept, we build a hypapp based on the APIs provided by the XMHF core. The hypapp takes an address of a physical page as input and sets its access permission in EPTs as inaccessible via the EPT manipulation API. 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 is able to successfully read and write the target page without triggering EPT violation, although it is marked as inaccessible in the EPT.

Caveat. We reiterate that the intention of presenting the attacks above is not to show the vulnerability of existing schemes, but to emphasize that applying memory isolation on the multicore setting is *not* as straightforward as one thinks. The low-level system implementation subtleties, if not treated properly, may compromise the security. We remark that the TLB invalidation problem is only the tip of the iceberg of the difficulty of securely implement memory isolation on a multicore setting.

4. In-depth Examination of Memory Isolation

Memory isolation is the primitive used in almost all virtualization based security systems [9], [10], [17], [18], [8], [7], [13], [19], [8]. 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, without carefully examining the technical details. In this section, we first revisit the existing memory isolation technique and then analyze the complications.

Before proceeding to the analysis, we spell out the system model and the threat model used in the rest of the paper. The system in consideration is a commodity multicore platform running a bare-metal micro-hypervisor with a single domain called the guest. The adversary is the kernel of the guest subverted by a rootkit. With the kernel's privilege, the rootkit can launch arbitrary software-based attacks, such as illicit memory accesses and manipulating the execution context. It may even have a CPU emulator to emulate the

hypervisor's behavior. We assume that BIOS, the firmware and the hardware in the platform are of integrity in the sense that their behaviors comply with the respective specifications and are not subverted by the adversary. We assume that the micro-hypervisor code, data and control flow are not compromised either during boot-up or at runtime. We do not consider side channel attacks or denial of service attacks. For the convenience of presentation, we use "the guest" and "the kernel" throughout the paper to refer to the virtual machine in the system and its kernel, respectively. Note that the models described above are the same used in the literature [9], [10], [17], [18], [8], [7], [13], [19].

4.1. The Common Practice

Early memory isolation schemes (e.g., Overshadow [2]) use para-virtualization with the shadow page tables and the hypervisor effectively manages all address mappings. With the advances of CPU virtualization, most of recent systems (e.g., [13], [7]) rely on the access control feature provided by EPT to realize memory isolation. To the best of our knowledge, most schemes are presented without stating whether they run on the multicore setting, except XMHF [13]. We consider a generic procedure in the unicore setting to explain the common practice of memory isolation widely used in the literature, including InkTag[9], AppShield[10], Heisenbyte[17], SPIDER[18], SecVisor[8], TrustVisor[7] and Sentry[19]. The isolation runs in the following steps.

- **Step 1.** The hypervisor is instructed to isolate a page at the virtual address V_a (e.g., via a hypercall) so that V_a 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 walks through the present kernel page table to obtain the corresponding guest physical address G_a and then walks through the EPT to locate the entry δ that maps G_a to the corresponding host physical page H_p. 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 sets back the $\hat{p}$ permission on δ.

An variant of the above 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 execution thread on the core. SPIDER [18] and Heisenbyte [17] are exemplary systems using this approach.

4.2. Complications of Memory Isolation In Multicore Setting

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

4.2.1. Cumbersome EPT Management. As shown in Section 2, each CPU core in the multicore setting 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, in contrast to permission switches described earlier in the unicore setting.

This fact gives rise to two implications. Firstly, it is no longer sufficient to maintain a single set of EPTs with all cores given 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 for memory isolation. The hypervisor has to record $k + 1$ different versions of permission settings. In the worst case, the hypervisor has to properly install n different EPTs at runtime for each core. A more complicating issue is that the hypervisor should have an algorithm to detect potential conflicts and/or inconsistency among the permission policies. The load 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. non-accessible). The privilege downgrade mandates that all related permission records have to be updated in an atomic way, including the page table entries and the TLB entries. Our attacks in Section 3 have shown that it requires knowledge of the low-level hardware behavior and sophisticated skills to update or invalidate all obsolete permissions.

4.2.2. Insecure Guest Page Table Checking. In the common practice described in Section 4.1, the trusted thread and untrusted threads use distinct EPT settings. However, they may use the same segment register and the guest page tables which are managed by the guest kernel and affect the CPU core's memory access. Stephen et. al. have used the Iago attack [20] 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. Because once verified, they are not subject to malicious modification because there is only one core, so that the guest OS is paused. Following this approach, InkTag [9], TrustVisor [7] and AppShield [10] have implemented kernel page table verification/protection *when* the hypervisor sets

up the isolation environment and *while* the protected thread is in execution.

However, although these schemes are secure in a unicore setting, they are vulnerable to the 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 sets the permission bits in the EPT. InkTag, TrustVisor and AppShield do not enforce core quiesce. Therefore, the guest kernel and the hypervisor can execute simultaneously. Therefore, the race condition attack can cause the access control policy to be enforced on unintended pages.

We use the following example to illustrate this. Suppose that a security-sensitive program is just launched and the hypervisor needs to setup its isolated environment. In order to lock and verify the current guest page table, the hypervisor has to find where it resides. In other words, it has to find out all physical memory pages used to store the guest page table, so that it is able to configure the corresponding EPT to lock it. Unfortunately, the hypervisor does not have the 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 `CR3` in order to find out the physical locations. Page table walking with software is a lengthy operation because it involves a number of mappings 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 address are easier to attack because they are exposed with a longer time gap. In our experiment, it takes around 120,000 CPU cycles to lock the entire page table used by a simple user-space 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 could still be vulnerable to the Iago-like attack. More generally, any data structure used during the address translation such as the segment descriptors are also potential attack vectors.

Note the core quiesce technique used in XMHF can defeat the aforementioned race-condition attack, since it freezes all untrusted execution while the hypervisor is in execution. However, it incurs a remarkably high performance cost. The hypervisor needs to find out all physical pages for the guest page table, sets the corresponding EPT to prevent the kernel's modification, and verifies whether the guest page table harbors any poisonous mappings. Note that the system is effectually frozen throughout these three steps.

Another solutions is to use the shadow page table instead of the EPT when the guest is launched. Obviously, the method does not fully leverage the hardware virtualization advantage. It affects the platform performance and significantly expands the hypervisor's size. We argue that it is not an ideal solution for the micro-hypervisor like XMHF and TrustVisor.

4.2.3. 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 the 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., a process or a 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 by using the present `CR3` register value (e.g., as in TrustVisor [7]) or the combination of `CR3` register value and the address of the kernel stack (e.g., as in AppShield [10]).

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 those hardware-related information, such as the instruction pointer stored in the `EIP` register and the page table root address stored in the `CR3` registers. Note that the application semantics of those data is translated by the kernel. For instance, the `EIP` register stores the virtual address of the next instruction to execute. It requires the mapping defined by the kernel to derive its guest physical address. Since the kernel is a potential adversary, it is infeasible for the hypervisor to correctly infer the logical representation of the subject from the hardware information.

Overshadow [2] 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 as weak. In the following, we consider a security-sensitive program P whose data buffer has been isolated by the hypervisor and show the impersonation attacks `CR3` and ASID based identification.

Impersonation Attack. Suppose that the `CR3` register is used to identify the subject. The malicious guest OS can launch a malicious process P' with the same `CR3` content as P, but with a different VA-to-GPA mapping. When P' issues a hypercall, the hypervisor is fooled to believe that the subject is program 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 does not improve the situation 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 the 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.

4.2.4. Summary. To summarize, memory isolation in the multiple core setting is challenging. Most complications are caused by two factors. One factor is due to the multicore setting. Namely, a mixture of trusted and untrusted threads run at the same time. The second factor is due to the design defect of the current memory isolation technique used in the literature. Namely, the isolation technique is *incomplete*. The boundary between the trusted and the untrusted is only drawn on the GPA-to-HPA mapping. At least, the VA-to-GPA mappings and the CPU core are still controlled by the kernel, which exposes a large attack surface to the adversary.

5. Full Isolation on Multicore Platforms

Considering the pitfalls of memory isolation in multi-core systems, we propose to isolate an entire computing environment including the CPU core, the memory region, and (optionally) the needed peripheral device. To be differentiated from the existing memory only isolation, we name the new isolation paradigm as *fully isolated micro computing environment* or FIMCE.

In the following, we describe the hardware and software architecture of FIMCE, and then present its versatile usages beyond isolation.

5.1. FIMCE Architecture

Figure 3 depicts the architectural difference between the memory isolation used in existing schemes and the full isolation of FIMCE. The main distinction is the boundary between the trusted and the untrusted.
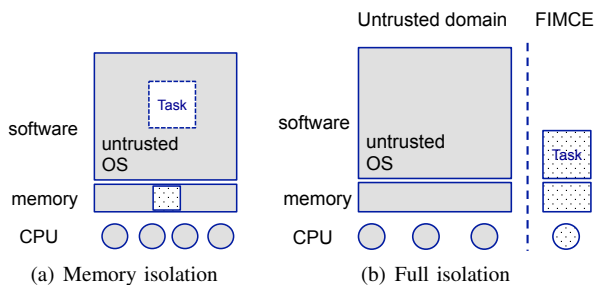

Figure 3. The comparison between memory isolation and full isolation. The grey areas denote resources controlled by the adversary while the areas with dots denoted isolated resources.

Core Isolation. Memory isolation neglects the CPU cores entirely, which is the reason why it faces the challenges of managing the EPT and identifying threads as explained in Section 4. The CPU core used by the protected task is isolated from the untrusted OS for two reasons. One is to avoid the same flaw as in memory isolation and the second is that the untrusted OS may use the inter-core communication mechanisms such as INIT signals to attack the task. Note that core isolation does *not* mean that a CPU core is permanently dedicated to a protected task. In fact,

the task can migrate from one core to another. However, whenever it runs, it exclusively occupies the CPU and is not preemptible by other threads, until it quits 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.

Once VMX is enabled on an Intel platform, the hardware automatically triggers VM Exit when receiving INIT signal and SIPI. To intercept NMI and external interrupts (including IPI), the hypervisor sets the NMI exiting bit and the External-interrupt exiting bit in the pin-based VM-Execution control bitmap of the VMCS structure. If an external interrupt has a non-empty handler installed inside the FIMCE, the hypervisor passes it to the FIMCE; otherwise, the interrupt is dropped.

CAVEAT. Our notion of CPU isolation has a different implication from the one in the system literature. The latter is mainly for performance loading and does not consider the kernel as an adversary.

Memory Isolation. Memory isolation still plays an important role in FIMCE. Existing memory isolation schemes in essence only separate the host physical address space of the security task from the rest. Since the guest page table is managed by the guest kernel, the security task's virtual address space and guest physical address space are still under the guest kernel's control.

In contrast, memory isolation in FIMCE is complete. 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 is allocated from a region reserved by the hypervisor. Since those pages are never mapped into the guest, the guest does not have any valid TLB entry to access it.

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 a FIMCE has exclusive accesses to peripheral devices annexed to it. Firstly, any I/O command issued from the guest to the FIMCE device should be blocked. For port I/O devices, the hypervisor sets corresponding bits in the guest's I/O bitmap. For MMIO devices, the hypervisor configure the guest's EPT to trap accesses to the MMIO region of the device.

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

5.2. Modularized Software Infrastructure of FIMCE

It is widely recognized that memory isolation schemes do not furnish the isolated task with the dependent software modules, which is one of the reasons why they require the task to be self-contained. In contrast, FIMCE has the inborn support for dynamically setting up the software infrastructure e.g., libraries, drivers and interrupt handlers, to cater to the task's needs.

Although it is theoretically possible to load a full-fledged OS into a FIMCE, our design goal is to use a FIMCE as a nimble environment hosting critical tasks within a program. Therefore, we propose to use a structured way to construct the needed software infrastructure. Based on their functionalities, a set of software modules called *pillars* are stored in the disk in the form of 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. To ensure the security, the hypervisor also takes a whitelist approach to verify their integrity and finally links them with the task.

The FIMCE only has a single and non-stopping execution thread since it is exclusively used by the protected task. When I/O operations are needed, the hypervisor also loads the interrupt handler table to the environment. When the protected task only accesses the enclosed memory, it can be optionally granted with Ring 0 privilege so that no context switch is needed in dealing with interrupts.

5.3. Applications of FIMCE

Although the basic idea of FIMCE is intuitive, it is advantageous over other isolation techniques on x86 platforms in terms of versatility. We briefly describe below several ways to use FIMCE. Due to the length limit, we leave the details in the full version of the paper.

Isolation. FIMCE surely supports conventional isolation applications e.g., to protect a decryption or authentication function. FIMCE offers better efficiency and stronger security than memory based isolation, since all those hassles described in Section 4.2 are no longer applicable. FIMCE also allows I/O operations which is infeasible for either SGX or memory isolation. In Section 6, we report our experiments of FIMCE with password-based decryption and Apache.

Malleability. The isolated FIMCE environment can be configured and used in non-standard ways to cater to the security goals. For instance, the hypervisor can twist the CPU registers and even the TPM configuration as needed.

To illustrate the benefit of a malleable FIMCE, we consider the challenge of ensuring that an application P's long term secret k can only be accessed in an isolated environment. Suppose that k has been initially encrypted with the binding to the isolated environment. The difficulty lies in how to authenticate the thread that requests to enter into the isolated environment and to access k.

Since the application cannot hide any secret in the unprotected memory against the OS, both have the equal knowledge and capability in terms of presenting the authentication information to the hardware such as the TPM chip or the SGX enclave. One may suggest to leverage the hypervisor to perform authentication as shown in [7]. However, as we have analyzed in Section 4.2, it is also challenging for the hypervisor to securely authenticate the application.

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

Note that `PCR17` and `PCR18` are in Locality 4 and 3 respectively. The hardware ensures that they cannot be reset by any software. During the boot up, the DRTM extends these two registers with the loaded modules. Their correct content implies the loading time integrity of the hypervisor. Since the OS is in Locality 0, it does not have the privilege to extend or reset `PCR20`, even though it can prepare the same input used by the hypervisor and application P. Other (malicious) applications in their own FIMCEs cannot impersonate P either. `PCR20` bears the birthmark of a FIMCE instance because the code in an 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. Besides the stronger security bolstered by the hardware, it is beneficial to minimize the TCB size and supports process migration.

Runtime Trust Anchor. Another noticeable strength of FIMCE is its ability to provide a secure environment that runs in parallel, and yet tightly-coupled, with the untrusted OS. 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 secure systems below.

The first system is to protect sensitive files in the disk from being modified or deleted by the untrusted OS. This problem has been considered by Guardian [21] and Lockdown [22]. The scheme used in Guardian cannot be scaled to protect arbitrary files chosen by applications while Lockdown suffers from performance loss as every disk I/O operation causes a context switch (if not optimized). In our approach, the hypervisor isolates the disk to the FIMCE. A

disk I/O filter is loaded in the FIMCE. It continuously loads the disk DMA request placed by the OS in a shared buffer. If the request is compliant with the security policy, the filter forwards it to the disk controller. Otherwise, it is dropped. All disk interrupts are channeled to the OS so that the filter is not necessarily involved in handling them. We expect a performance advantage of FIMCE based disk protection over Lockdown, because there is no context switches during disk operations. Note that a similar scheme can be used for filtering network packets.

The second system is about the runtime attestation of the OS behavior. Most existing remote attestation schemes [23], [24], [7] focus on loading time integrity check. It is challenging to realize runtime attestation because it requires the attestation agent to run securely inside the attesting platform managed by an untrusted OS. FIMCE has the natural fit to this problem. Intuitively, the agent resembles a kernel module which is shielded by the FIMCE and runs side-by-side with the OS. The attestation agent can read the kernel objects without facing the challenging semantic gap problem [25], [26], [27], [28]. To support kernel memory read, the entire kernel page table is reused in 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, for memory isolation schemes to achieve the same security goals described above. One of the fundamental reasons is that the OS in those schemes still manages the CPU cores. The OS can schedule off the protected execution thread from its CPU core before its attacks. In contrast, the FIMCE is immune from the OS's tampering and can function continuously.

6. Evaluation

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

6.1. Security Analysis

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

Complication Free. Recall that Section 4.2 has enumerated three security complications due to the multicore setting, namely complex EPT management, insecure guest page table checking and incapable thread identification. We remark that they all vanish in FIMCE.

- The EPT management of FIMCE is rather tidy. The EPT used for the OS (and the applications) are not affected by FIMCE while the EPT used for the FIMCE is static after it is launched. 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 does not use any 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 4.2.

- 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 instance created for it through its whole lifetime. It exclusively accesses the memory. The task may continue the execution without being preempted by other threads under the OS's control. In case that it relinquishes the CPU, its FIMCE hibernates without changing ownership. In other words, all memory states and the CPU context are saved. The CPU states are cleaned up before be handed to the OS. When needed, the FIMCE is re-activated from the saved state. Therefore, subject identification is not needed.

Minimal Attack Surface. The malicious kernel in memory isolation systems enjoys a large attack surface, as it has the full control over the CPU cores and over the VA-to-GPA mapping, which leads to various attacks and design complications described in Section 3 and 4. In contrast, the attack surface exposed by our scheme is minimal.

Owing to the full isolation approach, the FIMCE's hardware and software are beyond the kernel's access, interference and manipulation. The kernel cannot access the FIMCE core's registers, L1 and L2 caches. L3 cache is not effectively accessible either because the FIMCE's host physical address region is never mapped to the guest. Although the kernel may use IPI or NMI to interrupt the FIMCE, the worst consequence is equivalent to a DoS attack. Since the isolated code responds to the interrupts by itself, an IPI or NMI only cause 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 tear down an 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 cope with such attacks.

Small TCB Size. The hypervisor is the *only* trusted code in the system. Owing to its simple logic, our hypervisor has a tiny code base with around 6K source lines of code for runtime execution. The concurrency issue of FIMCE is not difficult to handle. Because only the setup and teardown code are possible to execute concurrently on different cores, they can be guarded with simple spinlocks. Note that each FIMCE instance does not have overlapping regions, which also simplifies the concurrency handling.

6.2. Implementation

We have implemented a prototype of FIMCE on a desktop computer configured with an Intel Core i7 2600 quad-core processor running at 3.4 GHz, Q67 chipset, 4GB of DDR3 RAM and a TPM chip. The platform runs a guest Ubuntu 12.04 with the stock kernel version 3.2.0-84-generic. We have implemented the FIMCE hypervisor of around 6000 SLOC. The TCB of FIMCE only consists of the hypervisor. It exports two hypercalls, i.e., `FIMCE_start()` and `FIMCE_term()`, for starting and terminating a FIMCE respectively. We slightly modify Intel's open source TXT bootloader *tboot*[2] as the DRTM to load our hypervisor. During hypervisor initialization, a set of EPT entries are initialized such that a chunk of physical memory is reserved for exclusive use by the hypervisor. During the OS kernel initialization, all cores are set to use the same set of EPT, 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 [3], a TPM driver pillar of 20KB size. The implementation also encloses a pillar management code of 413 SLOC which verifies and links pillars in the FIMCE.

6.3. Benchmarks

Since a FIMCE occupies a CPU core exclusively, the OS has less computation power at its disposal while a 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 running inside a FIMCE.

For the multithreaded *SPECint_rate 2006*, we set the concurrency level to four. Figure 4 shows that it has 15% percent performance drop in 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 levels concurrency. The results are reported in Table 1.

2. http://sourceforge.net/projects/tboot/
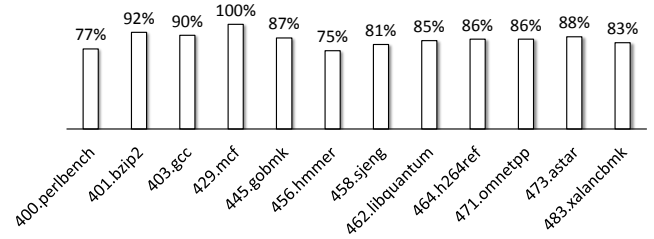
3. https://tls.mbed.org/

Figure 4. SPECint_rate 2006 results. The numbers are the percentage of the score with FIMCE to the score without FIMCE.

TABLE 1. 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

The two sets of experiments indicate that the relative performance loss grows with the degree of concurrency, mainly due to more frequent context switches. Nonetheless, the loss is bounded by the inverse of the number of physical cores in the platform (namely 25% in our setting).

To verify our estimation that FIMCE does not incur much performance for single-threaded applications, we run Lmbench, Netperf and Postmark with and without FIMCE. Figure 5 shows that most tasks of *Lmbench* are not affected by FIMCE, except one task has 8% performance drop. Similar results are also found for Netperf as in Table 2 and Postmark as in Table 3.
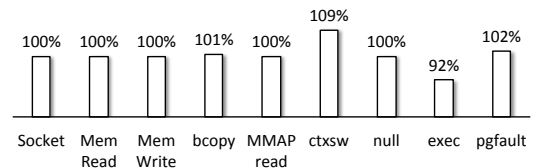

Figure 5. Lmbench results. The numbers are the percentage of the score with FIMCE to the score without FIMCE.

TABLE 2. NETPERF BANDWIDTH WITH AND WITHOUT FIMCE RUNNING, IN MBPS.

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

TABLE 3. SINGLE-THREADED POSTMARK PERFORMANCE WITH AND WITHOUT FIMCE RUNNING, IN SECOND

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

6.4. Component Costs

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

On average, a null hypercall on our platform takes 0.31 millisecond. FIMCE setup takes about 47.33 milliseconds which is the interval between the `FIMCE_start` hypercall to the `INIT` signal prior to start of FIMCE execution. The code loading time depends on the total binary size of the loaded pillars and the security task. Table 4 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 $2\mu s$. Pillar loading also involves integrity

TABLE 4. 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

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

6.5. Application Evaluation

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

Password based decryption. Through this case study, we demonstrate FIMCE's advantage over memory isolation in terms of supporting I/O operation. It is challenging to protect tasks with I/O operations using memory isolation, mainly because I/O operations are normally in the kernel level with a large and dispersed code base and are interactive with devices. Therefore, Driverguard[5] relies on manually instrumenting the driver code, which is tedious and error-prone, whereas TrustPath [29] has to relocate the entire driver into the isolated user space code, which not only requires significant changes on the user space, but also burdens the hypervisor with complex functions. As a result, there are a lot of hypercalls when issuing I/O commands

TABLE 5. MODIFIED APACHE PERFORMANCE, SSL HANDSHAKE 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

TABLE 6. OVERHEAD OF OTHER PROTECTION SCHEMES

Schemes	Overhead
TrustVisor [7]	9.7% to 11.9% depending on concurrent transaction
InkTag [9]	2% in throughput, 100 concurrent request
Overshadow [2]	20% to 50% on a 1Gbps link, 50 concurrent request

and handling interrupts, which incurs a heavy performance drop because of frequent expensive VM exits.

FIMCE offers a much tidier solution. The code running inside a 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.

In this case study, a program performs an AES decryption after converting a user password through the keyboard into the decryption key. When a 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. In average, it takes 0.94 milliseconds to decrypt the ciphertext with 1kilobytes, which is only 5.2% slower than in the guest.

Apache Server. In this case study, we utilize FIMCE to harden an SSL web server by isolating its RSA decryption of SSL handshakes into a FIMCE. As noted previously, existing systems [7], [2] on Apache protection is not secure under the multicore setting. In their schemes, the isolated code runs in the *same* thread as its caller. As a result, it incurs frequent VM-exits and VM-entries as the control flow enters and leaves the isolated environment. FIMCE does not incur context switches at runtime because the isolated task in a FIMCE runs as a separated thread in parallel as others.

In the experiment, we customize the Apache source code so 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 LAN and run ApacheBench with different concurrency levels. The Apache server hosts an 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 5.

It is evident that at low concurrency level up to four,

the FIMCE-enabled Apache server performs almost equally well as the native multithreaded Apache. It outperforms existing schemes listed in Table 6 due to the fact that FIMCE does not involve costly context switches. However, its performance drops with the concurrency level increasing, but is bounded by 25%. This is because of the single-threaded of FIMCE cannot match the performance of a multithreaded Apache which can uses all four cores to perform concurrent decryption. The performance of TrustVisor[7], InkTag[9] and Overshadow[2] is not affected by concurrency, albeit they are *not* secure in a multicore system due to the stifling attack.

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 word web transactions, the time spent for RSA decryption accounts for a much smaller portion of the entire transactions as compared to in the benchmark testing, because of (1) longer network delays in the Internet; (2) more SSL sessions using the same master key decrypted from one SSL handshake; (3) more time needed to generate or locate the need web pages. Therefore, 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 case. We implemented a prototype to bind a long term secret to a FIMCE instance. The system is booted using DRTM. A simple security task and the TPM pillar are loaded into a FIMCE. During loading the hypervisor extends `PCR 20` accordingly. The security task seals and unseals a long term secret to `PCR17, PCR18` and `PCR20` which bear the birthmark of this FIMCE instance. We measured time taken by the TPM seal and unseal operations inside FIMCE and compared it with the performance inside the guest. For running inside the guest, we converted the TPM pillar to a kernel module and run it inside the kernel space. In both experiments, a 20 byte long secret is used. The results are in Table 7.

TABLE 7. TPM PERFORMANCE TEST, IN SECONDS

	TPM Seal	TPM Unseal
Guest	0.54	0.96
FIMCE	0.41	0.94

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

It is also straightforward to use this facility by the security task. It only needs to instruct the hypervisor to load the TPM pillar and invoke corresponding functions inside.

Compared to existing approaches that virtualize the TPM using software such as [7], our approach places the trust anchor directly on the hardware TPM chip. In contrast, virtualizing TPM requires one more entity which is 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 in such a way that eliminate the requirement, shrinking the attack surface and TCB.

Runtime Kernel Introspection with Attestation. We implement a prototype of the introspection. We use a security task to read the `mm_struct` member of the `init_task` structure and measured the performance. 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 [28] 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 a software to walk the page table. Our system also provides stronger security because the introspection code runs with the genuine `CR3` presently used in the guest, which implies the same address layout and mapping as the kernel.

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 case implementation, the introspection code inside the FIMCE uses the crypto pillar to signs 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 to the guest OS. It takes 3.47 seconds in average to perform the entire procedure, including the time for TPM quote operation.

7. 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 [1] and Proxos [30] 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 mainstream commodity platforms nowadays enjoy hardware support for CPU, memory and I/O virtualization. By taking advantage of having a more privileged bare-metal hypervisor (a.k.a. virtual machine monitor or VMM) than the OS, various systems [8], [12], [29] have been proposed in the literature for diversified security purposes. Despite of the 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 attribute bits in relevant page table entries.

Two typical examples of kernel protection are SecVisor [8] and Lares [12]. 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 locations of kernel. Both uses the hypervisor to prevent the kernel code from being modified. TrustPath[29] and Driverguard[5] 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 the sensitive code against an untrusted operating system. TrustVisor [7] is a tiny hypervisor that builds such an environment for a self-contained PAL. It further enhanced the environment with a software-implemented TPM (called μTPM) in the hypervisor space. μTPM can protect 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 a large code base. Based on TrustVisor, XMHF [13] provides an open-source hypervisor framework providing security functionality including memory protection. Also based on TrustVisor, Minibox [31] combines the hypervisor with Native Client [32] to provide a two-way sandbox for the cloud. Another line of research is to protect the entire application. Overshadow [2], InkTag [9], and AppShield [10] 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 the OS behaviors by using the paraverfication technique which mandates changes on the kernel's code. A common challenge for isolating an application is to handle the system calls. All three scheme requires intensive work on system call adaption and parameter marshaling. Different from the coarse-grained cross VM isolation used in Terra [1], these systems provides fine-grained in-VM isolation. Unfortunately, as we shown later, their security hardly withholds under a multicore setting.

Isolation With Other Techniques.

Flicker [33] 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 Dynamic Root of Trust Measurement (DRTM). The late launch technique sets up a secure and measured environment to protect a piece of code and data. The drawback is its high latency due to the slow speed of the TPM chip. Moreover the protected code cannot interact with the rest of the platform.

The recently announced Intel Software Guard Extensions (SGX) [34] 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.

Trustzone [35] in ARM platforms provides a more versatile secure execution environment. As shown in TZ-RKP [36], a security monitor residing in the secure world established by TrustZone can protect the OS kernel in the normal world at runtime.

Virtual Ghost [37] uses 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 [38] creates an isolated execution environment on GPU. Being an isolated device from the CPU with its own memory, GPU provides a natural ground for building an isolated execution environment. In the past, programming on GPU had been hard because of its highly specialized hardware, however, modern GPU is becoming ever-increasingly more programmable so that running code for execution on GPU is easier. Nonetheless, this approach still requires significant development effort because there is little support from current systems.

SICE [39] isolates a program that ranges from an instrumented application to a complete VM from the guest OS using System Management Mode (SMM). Compared to microhypervisor approach, it consists of a smaller TCB since the TCB only consists of the hardware, BIOS and SMM. 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.

8. Conclusion

In this paper, we have undertaken an in-depth study on virtualization based isolation on multicore platforms. Our results show that the current mainstream approach of using the page tables to isolate a sensitive memory region is cumbersome and ineffective. We have demonstrated two specific attacks on XMHF and BitVisor to show their design flaws. Furthermore, we propose FIMCE, a stronger and tidier isolation scheme for multicore systems. FIMCE places

13

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 is featured with strong security due to its minimal attack surface and with great nimbleness and versatility due to its architectural advantages. We have implemented FIMCE and experimented it with several testing cases which demonstrate various advantages over alternative techniques.

Acknowledgments

References

[1] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," in *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[2] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

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

[4] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "Bitvisor: a thin hypervisor for enforcing I/O device security," in *Proceedings of the 5th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2009. [Online]. Available: http://doi.acm.org/10.1145/1508293.1508311

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

[6] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: wimpy kernels for on-demand isolated i/o," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 308–323.

[7] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (S&P)*, 2010.

[8] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[9] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

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

[11] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 565–578. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/cho

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

[13] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, implementation and verification of an extensible and modular hypervisor framework," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[14] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: Bare-metal performance for i/o virtualization," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

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

[16] S. Rostedt, "The x86 NMI iret problem," https://lwn.net/Articles/484932/, accessed: 2015-11-10.

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

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

[19] A. Srivastava and J. Giffin, "Efficient protection of kernel data structures via object partitioning," in *Proceedings of the 28th annual computer security applications conference*. ACM, 2012, pp. 429–438.

[20] S. Checkoway and H. Shacham, "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)*, 2013.

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

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

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

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

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

[26] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 297–312.

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

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

[29] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building Verifiable Trusted Path on Commodity x86 Computers," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, ser. S&P, May 2012.

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

[31] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference*, 2014.

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

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

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

[35] A. Limited, "Arm security technology - building a secure system using trustzone technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[36] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervison across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, 2014.

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

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

[39] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 375–388.