Seeing Through The Same Lens: Introspecting Guest Address Space At Native Speed

Siqi Zhao
Singapore Management University

Xuhua Ding
Singapore Management University

Wen Xu*
Georgia Institute of Technology

Dawu Gu
Shanghai JiaoTong University

Abstract

Software-based MMU emulation lies at the heart of out-of-VM live memory introspection, an important technique in the cloud setting that applications such as live forensics and intrusion detection depend on. Due to the emulation, the software-based approach is much slower compared to native memory access by the guest VM. The slowness not only results in undetected transient malicious behavior, but also inconsistent memory view with the guest; both undermine the effectiveness of introspection. We propose the *immersive execution environment* (ImEE) with which the guest memory is accessed at native speed without any emulation. Meanwhile, the address mappings used within the ImEE are ensured to be consistent with the guest throughout the introspection session. We have implemented a prototype of the ImEE on Linux KVM. The experiment results show that ImEE-based introspection enjoys a remarkable speed up, performing several hundred times faster than the legacy method. Hence, this design is especially useful for real-time monitoring, incident response and high-intensity introspection.

1 Introduction

The thriving cloud computing has kept driving the research on virtual machine introspection (VMI) [12, 18, 19, 21, 23, 28, 32, 33, 34, 35] in the recent years to address the growing security concerns on virtual machines. The center of the VMI research is to bridge the semantic gap [24], namely, to reconstruct the high level kernel semantics by accessing the guest kernel's virtual address space. For instance, the VMI tool in the monitor VM extracts all running processes' identifiers in an untrusted guest VM by traversing the guest kernel's `task_struct` list.

When the tool is deployed inside the target VM, it is trivial to access the guest virtual address space. Nonetheless, such an in-VM introspection [12, 33] induces guest OS modification and is subject to attacks if the guest kernel is subverted. Placing the introspection agent outside of the guest is a more appealing approach. Such an out-of-VM introspection then faces the problem of replicating the guest's virtual address (VA) to host physical address (HPA) translation.

Existing out-of-VM introspection systems [18, 19, 32, 34] tackle the problem using a software-based address translation whereby the MMU's function is replaced by software. As a result, the software-based access is much slower than the native speed access in the guest. The speed inferiority clearly impacts introspection performance, e.g., longer turnaround time to scan the kernel's code section. Moreover, it has several negative security implications. It costs more precious time for live forensics and incident response. It is also incapable of continuously monitoring a critical memory location as the introspection loses the race against the attack running at native speed. Most importantly, it is difficult for the software-based method to maintain consistent VA-to-HPA mappings with the guest kernel, because it is not amenable to tracking and following `CR3` updates in the guest. Inconsistent mappings consequently impair the security of introspection. We stress that the cache mechanism does improve performance, however, at the cost of potential mapping and data inconsistency since the cached mappings and data could be stale.

In fact, mapping consistency can not be assumed for an in-VM introspection scheme without trusting the guest kernel, even though the memory is introspected at native speed. For instance, SIM [33] isolates its monitoring code in an isolated address space whereas it does not prevent the malicious kernel thread from using a different address mapping. The consistency issue persists in the broader scope of system monitoring. As shown by Jang et. al [25], hardware-assisted monitor systems such

*Work was mainly done when visiting SMU as a research assistant.

as Copilot [29] and KI-mon [26] are circumvented by using address translation redirection attacks which deceive the monitor into using a faked mapping.

In this paper, we propose a novel mechanism to allow the introspection code in the monitor VM to access a target guest kernel's virtual address space at native speed and with mapping consistency, despite the kernel-level attacks from the target. The code runs in a carefully designed execution environment named as the *Immersive Execution Environment* (ImEE). During a guest access, the ImEE's MMU walks the present paging structures same as the guest's, pointed to by the `CR3` registers both in the ImEE and in the guest.

We have implemented a prototype of the ImEE on Linux KVM. The experiments demonstrate a remarkable performance boost. As compared to the existing software-based guest access method, the ImEE is several hundred times faster to traverse kernel objects. The ImEE is so lightweight and nimble that it only needs 23μs to activate and 7μs to switch the introspection target, around 200 times faster than the software method. Hence, the ImEE is more attractive to applications desiring strong security, faster response and high speed, for instance, critical data monitoring, virtual machine scanning, and live forensics.

CAVEAT. Our contribution in this paper is complementary to existing out-of-VM introspection systems [19, 18, 28, 32]. Those innovations focus upon more software issues, like efficient kernel-level semantic reconstruction [19] and race conditions [28]. In contrast, it is out of our scope to deal with the high-level issues like which virtual addresses or kernel objects to read and how to reuse the existing kernel code [19]. We expect that, with modest retrofitting, those VMI applications can harness the ImEE as a powerful guest access engine to achieve better performance and stronger security.

ORGANIZATION. The next section briefly reviews the legacy method to access the target VM and analyze its weakness. We present a synopsis of the work in Section 3. The design details of the ImEE and the code running inside are presented in Section 4. The implementation and performance evaluation are described in Section 5 and 6, respectively. We then discuss several related issues in Section 7, and briefly review the literature in Section 8. Lastly, Section 9 concludes the paper.

2 Inadequacy of Software-based Guest Access

It is a common practice in the VMI literature to use the software-based method to translate virtual addresses before accessing a target guest VM. The guest's own paging structures cannot be directly replicated in the monitor VM, because it is incompatible with all software therein. In addition, there is also a security concern that the guest's code or data could be used to attack the monitor VM.

In this software-based approach, the target memory is mapped to the monitor VM as a set of read-only pages. Given a virtual address X, the introspection code walks through all levels of the paging structures, including the Extended Page Tables (EPTs[1]) in the memory to find out the corresponding HPA. It then maps the HPA to its own virtual address space, and finally issues an instruction to read it. Obviously, such a procedure incurs a much longer latency than the native access to X in the guest.

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

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

Caching techniques have been used in order to reduce the latency of guest accesses. For instance, LibVMI [30] introduces three types of caches: the page-level data cache, the VA-to-HPA translation cache and the `pid` to `CR3` cache. While promoting the performance, using the caches is detrimental to effective introspection. Since the guest continuously runs during the introspection, any cached mapping or data is not guaranteed to be consistent with the one in the memory. Moreover, it is difficult for the software-based method with caches to catch up with the pace of `CR3` updates in the guest. Since the guest kernel is untrusted, the introspection cannot presume that all

[1]Throughout this paper, we following Intel's terminology to describe the scheme. It can also be implemented on AMD processors supporting MMU virtualization.

guest threads share the same kernel address space. CR3 synchronization with the guest may lead to cache thrashing which backfires on the introspection performance.

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

3 Synopsis

3.1 Models and Scope

System Model. We consider a multicore platform supporting both CPU and MMU virtualization. Under the management of a bare metal hypervisor, the platform runs a trusted monitor VM and a set of untrusted guest VMs which are the targets of introspection. The platform administrator runs VMI applications inside the monitor VM to introspect the *live* kernel states in the targets without modifying or suspending them.

To avoid ambiguity, we use the "target" to refer to the virtual machine under introspection, and use "guest" with its hardware virtualization notion as in a "guest physical address" (GPA) which refers to the physical address a kernel uses inside a hardware-assisted virtual machine.

Trust Model. We assume all hardware and firmware in the platform behave as expected. We trust the hypervisor and the software in the monitor VM and assume that the adversary cannot compromise the hypervisor or the monitor VM's kernel at launching time and runtime. We do not trust any software running in the target, including the kernel.

Scope of Study. The adversary we cope with resides in the target kernel. Its goal is to stage a *fake* kernel address space view to the VMI application. Namely, its attack causes the VMI application to read those memory bytes that are "thought" to be used by kernel threads but are actually not. Attacks that aim to beat the VMI logic, e.g., manipulating a function pointer not known to the introspection logic, are beyond and orthogonal to our scope of study. Side-channel attacks or denial-of-service attacks are not considered either.

3.2 Basic Idea

Our idea is to create a special computing environment called *Immersive Execution Environment* (ImEE) with a twisted address mapping setting (as in Figure 1). The ImEE's CR3 is synchronized with the target VM's active CR3 so that its MMU directly uses the target's VA-to-GPA mappings. Its GPA-to-HPA mappings are split into two. The GPAs for the intended introspection are translated with the same mappings as in the target VM; the GPAs for the local usage (indicated by the dotted box in Figure 1) are mapped to the local physical pages via separated GPA-to-HPA mappings. With this setting, memory accesses are automatically directed by the MMU into the target and the local memory regions according to the paging structures.

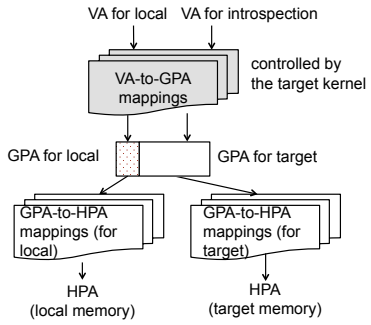

Figure 1: Illustration of the idea of direct usage of the target VM's VA-to-GPA mappings and splitting in GPA-to-HPA mappings. Note that the shadow box is fully controlled by the target (i.e., the adversary).

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

3.3 Challenges

Suppose that the ImEE has been set up following the idea above with an introspection agent running inside and accessing the target memory. The following design chal-

lenges need to be addressed in order to achieve a successful introspection.

Functionality Challenge. The ImEE agent's virtual address space comprises of the executable code, data buffers to read and write, and the target kernel's address space. Since the agent code and data are logically different from the target kernel, we need a way to properly *split* the GPA domain so that VAs for the local uses are not mapped to the target and VAs for introspection are not mapped to the agent memory.

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

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

A more subtle, yet important issue, is that the introspection *blind spot*, namely the set of virtual addresses in the target which are not reachable by the ImEE agent. As shown in Figure 2, a VA for introspection is in the blind spot if and only if it is mapped to the GPA for local use. This is because the full address translation ends up with a local page, instead of the target VM's page. The malicious target can turn its pages into the blind spot by manipulating its guest page table. The blind spot issue has two implications. First, detecting its existence efficiently is challenging. Note that it is time-consuming to find out all VAs in the blind spot, because the guest page tables have to be traversed to obtain the GPA corresponding to a suspicious VA. Second, the attacker can manipulate VA to GPA mappings in an attempt to disrupt the execution of the ImEE agent. By manipulate the mappings, the attacker tries to cause invalid code to be executed inside the environment, or cause the introspection to read arbitrary data.

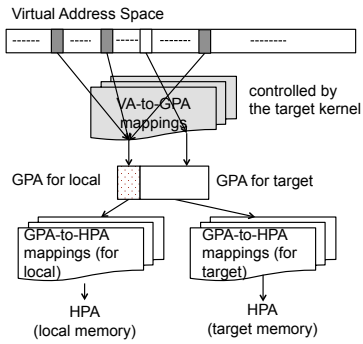

Figure 2: Illustration of the blind spot comprising three virtual pages (in the dark color). Target kernel objects in those pages cannot be introspected since they are mapped to the local memory.

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

Besides the above three major challenges, there are other minor issues related to the runtime event handling, such as page faults and the target VM's EPT updates. The requirement of Out-of-VM introspection is to minimize intrusive effects on the target. For example, the hypervisor is refrained from modifying the target VM's guest page tables because it leads to execution exceptions in the target. Therefore, the minor issues also need careful treatment.

3.4 System Overview

The ImEE is in essence a special virtual machine which is created and terminated by the hypervisor based on the VMI application's request. Like a normal VM, the ImEE hardware consists of a vCPU core and a segment of physical memory, both (de)allocated by the hypervisor when needed. No I/O device is attached to the ImEE. The ImEE does not have an OS and the only software running in it is the ImEE agent which reads the target memory. Figure 3 depicts an overview of the whole system.

The VMI application can launch the ImEE, put it into

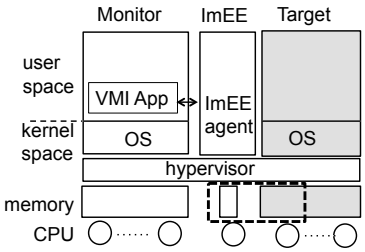

Figure 3: Overview of ImEE-based introspection. The box with dashed lines illustrates the mixture of physical memory. The shadowed regions belong to the target and are not trusted.

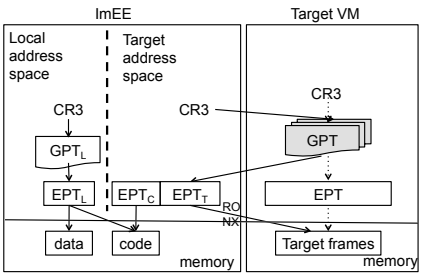

Figure 4: The solid arrows describe the translation for a VA within the ImEE, while the dotted arrows describe the translation inside the target. All target frames accessible to the ImEE agent are set as read-only and non-executable in EPT_T.

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

4 The Design Details

In this section, we first explain the internals of the ImEE with the focus on the paging structures, and then explain the ImEE agent. We show our design choices for performance where appropriate. Lastly, we describe the lifecycle of ImEE, focusing on the runtime issues such as transitions between sessions.

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

4.1 ImEE Internals

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

According to the CR3 content, the agent runs either in the *local address space* or the *target address space*, as depicted in Figure 4. When in the local address space, the agent interacts with the VMI application while it runs in the target address space to read the target memory. The code frame is mapped into both spaces while the data frame is mapped in the local address space only.

Local Address Space. The paging structures used in the local address space comprise GPT_L and EPT_L, which map the entire space to the ImEE frames. GPT_L only consists of two pages as shown in Figure 5. The global flag on the GPT_L is set so that the local address space mappings in the TLB are not flushed out during CR3 update. Specifically, only one virtual page is mapped to the data frame while all others are mapped to the code frame. With this setup, the agent code can execute from all but one page. Moreover, the GPAs of the ImEE frames are *not* within the GPA range the target VM uses, which avoids conflict mappings used in the target address space.

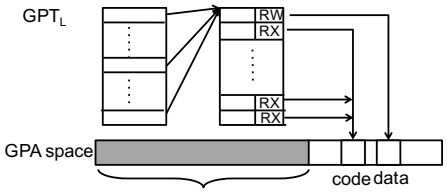

Figure 5: The Illustration of GPT_L. All entries in the page table directory point to the same page table page which has one PTE points to the data frame and all other to the code frame.

Target Address Space. The target address space implements our idea in Figure 1. To run the agent in this space, the ImEE CR3 register is synchronized with the target CR3, so that they use the same guest page tables. The GPA-to-HPA mapping used in this space are governed by EPT_T and EPT_C.

All GPAs are mapped to the target frames by EPT_T, except one page is redirected by EPT_C to the ImEE code frame. Specifically, EPT_T is populated with the GPA-to-HPA mappings from the target VM's EPT, except that all target frames are guarded by read-only and non-executable permissions. This stops the agent from modifying the target memory for the sake of non-intrusiveness. It also prevents the adversary from injecting code, because the adversary can place arbitrary bi-

naries to those frames. The permission of the mapping defined by EPT_C is set as *executable-only*. Namely, it cannot be read or written from the target address space.

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

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

4.2 ImEE Agent

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

Our description below involves many addresses. We use Table 1 to define the notations.

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

Table 1: Address notations. For instance, GP_c is the guest physical address of the ImEE code page in the local address space.

Overview. The main logic of the agent is as follows. Initially, the agent runs in the local address space and reads an introspection request from the data frame. Then it switches to the target address space and reads the targeted memory data from the target memory into the registers. Finally, it switches back to the local address space, dumps the fetched data to the data page and fetches the next request.

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

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

Figure 6: The sketch of the ImEE agent's pseudo code

The agent is loaded at P_c in the local address space by the hypervisor. P_c is chosen by the hypervisor such that it is an executable page according to the target's guest page table. Because GPT_L maps the entire VA range (except one page) to the code frame. Therefore, there is an overwhelming probability that P_c is also an executable page in the local address space[2]. Therefore, the agent can execute in the two address spaces back and forth which explain Line 12 and 17 can run successfully without re-

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

location.

Impact of TLB. No matter whether there is an attack or not, TLB retention has no adverse effect on the introspection. Suppose that the mappings in the local address space are cached in the TLB. When the agent runs in the target address space, the only VAs involved are for the instructions (P_c) and the target addresses (P_t). For VAs in P_c, the cached mapping remains valid because the address mappings are not changed. There are two exclusive cases for P_t. If $P_t \neq P_d$, the translation does not hit any TLB entry because it is never used in the local address space. Otherwise, the TLB entry for P_d is still considered as a miss because of different PCIDs. The same reasoning also applies to the cached mappings in the target address space.

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

4.3 Defeating Attacks via the Blind Spot

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

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

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

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

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

- $GP_t \neq GP'_c$, and GP_t is within the pre-assigned GPA range for the target VM. In this case, the ImEE's MMU walks the target VM's GPT and fetches the data in the same way as in the target VM. In other words, the mapping consistency between the ImEE and the target VM is still guaranteed. Although the agent may read invalid data, its execution is not affected by such mappings. The attack has no harm to the execution as it is equivalent to feeding poisonous contents to the VMI application, in the hope to exploit a programming vulnerability. We remark that this is the inevitable risk faced by any memory introspection and can be coped with software security countermeasures.

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

4.4 Operations of ImEE

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

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

Activation. Based on the VMI application's request, the hypervisor launches the ImEE wherein the agent runs in

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

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

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

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

Page Fault Handling. Although it is rare for kernel introspection, it is possible to encounter a page fault due to absent pages in the target VM. One possible reason is that the malware inside the target attempts to evade introspection by swapping out page content to disk. In this case, since the mapping inside ImEE is consistent with the one in the target VM, introspection on the swapped-out page results in a page fault inside ImEE. We remark that this behavior is the expected consequence of maintaining mapping consistency between ImEE and the target. The effectiveness of ImEE's introspection is not undermined because once the swapped-out page is swapped in, it is visible to ImEE immediately.

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

5 Implementation

In this section, we report the details of our ImEE prototype implementation. We describe our prototype based on KVM and the introspection tools we implemented on top of our prototype.

5.1 ImEE on KVM

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

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

5.2 Specialized Agent

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

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

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

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

5.3 Usability

The simple interface of ImEE allows easy development of introspection tools. For common introspection tasks that focus on kernel data structures, the development requires a selection of the agent type, and a set of memory reading parameters including the starting virtual address, the number of bytes to read, and the offset(s) used for traversal. Based on this method, we have developed four user space VMI programs that collect different critical kernel objects and have distinct memory reading behaviors. The objectives and logics of the four programs are explained below.

- syscalldmp It dumps totally 351 entries of the guest's system call table pointed to by `sys_call_table`. A continuous block of 1404 bytes from the guest is returned to the program.
- pidlist It lists all process identifiers in the guest. It traverses the `task_struct` list pointed to by the kernel symbol `init_task`, and records the `PID` value of every visited structure in the list. In total, 4 bytes are returned while 8 bytes are read from the guest for each task.
- pslist It lists all tasks' identifiers and task names stored in `task_struct`. A task's name is stored in the member `comm` with a fixed size of 16 bytes. Hence, 24 bytes are returned for each task node.
- credlist It lists all tasks' credential structures referenced by the `task_struct`'s `cred` pointer. In total, 116 bytes including the credential structure to the application for each task node. Hence it takes more time than pidlist and pslist.

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

6 Evaluation

We evaluate our prototype from four aspects with LibVMI as the baseline. LibVMI [30] is a cross-platform introspection library which a variety of tools depend on. To the best of our knowledge, LibVMI is the only open-source tool that provides a comprehensive set of API for reading the memory of a VM. In particular, it provides the capability to handle translation from VA to GPA. Therefore, LibVMI plays the role of a building block for live memory access in tools such as Drakvuf[**?**] and Volatility[36]. Our evaluation consists of four parts. Firstly, we consider the overhead of ImEE, in terms of component costs and the impact on the target VM due to `CR3`-update interception. Secondly, we measure the ImEE's throughput in reading the target memory. Thirdly, we compare the introspection performance of the tools with two functionally equivalent ones implemented with the LibVMI and in the kernel. Lastly, we compare ImEE with LibVMI in a setting with multiple guest VMs.

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

6.1 ImEE Overhead

Table 3 summarizes the overheads of the ImEE. It takes a one-time cost of 97 μs to prepare the ImEE environment where the main tasks are to make a copy of the target guest EPT as $\texttt{EPT}_T$, to set up $\texttt{GPT}_L$ and $\texttt{EPT}_L$, and to allocate and setup the ImEE vCPU context. The ImEE activation requires about 3.2 μs, and the agent loading/reloading time is around 6.5 μs. The difference is mainly due to handling of the incoming IPI by host kernel on the ImEE core in the agent reloading case. In comparison, it takes about 100 milliseconds to initialize the LibVMI setting, which is around 1,000 times slower than the ImEE setup.

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

Table 3: Overhead comparison between ImEE and LibVMI.

Guest CR3 Update Interception. To maintain `CR3`

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

The cost due to interception mainly consists of VM-exit, sending an IPI, recording VMCS data, and VM-entry. In total, it takes about 2000 CPU cycles which amounts 0.58 μs in our experiment platform. We run three performance benchmarks: LMbench [2] for system performance, Bonnie++ [14] for disk performance and SPECint 2006 [15] for CPU performance while context switches during their executions are intercepted by the hypervisor. Figure 7 reports the LMbench score for context switch time where the performance drops about 50%.

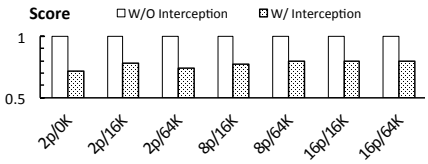

Figure 7: LMBench: normalized result on context switch time. The higher score means better performance.

Nonetheless, the interception does not seem to incur noticeable impact to other benchmark results such as disk I/O and network I/O, as shown in Figure 8, 9 and 10. We attribute this effect to the relatively fewer number of context switches involved during the macro-benchmark runs, because the benchmark processes fully occupy the CPU time slot. It is typical for a Linux process to have between 1ms to 10 ms time-slot before being scheduled off from the CPU.

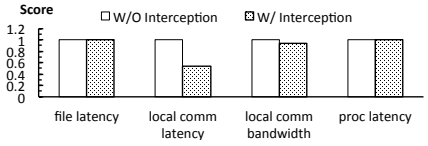

Figure 8: LMBench: normalized result on others system aspects. The higher score means better performance..

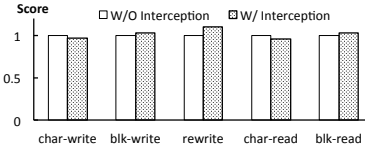

Figure 9: Bonnie++: normalized results on disk performance. The higher score means better performance.

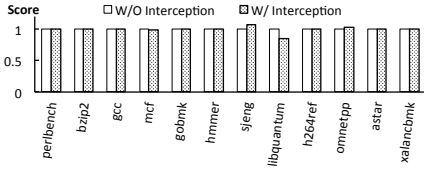

Figure 10: SPEC INT: normalized results on CPU performance. The higher score means better performance.

To understand the impact of CR3 interception in real-life scenarios, we test it with three different workloads on the target VM: idle, online video streaming and file downloading. Neither test shows noticeable performance drop. When the target is under interception, the video is rendered smoothly without noticeable jitters and the file downloading still saturates the network bandwidth.

In our experiments, we find that the introspection encounters few context switches in the target VM. To understand this phenomena, we run experiments to measure the intervals between context switches. Figure 11 shows the distribution of their lengths under different workloads. The analysis shows that the context switch is expected to occur after around 40 μs, which could be used as a guideline for the VMI application to determine the duration of a session. Note that an encounter with the context switch costs about 6.5 μs for the introspection and 0.58 μs for the target VM.

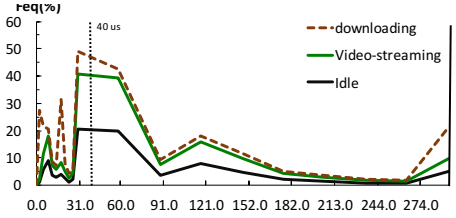

Figure 11: The frequency distribution of interval lengths between context switches in three workloads: idle, video streaming and file downloading. The x-axis is not displayed to the scale.

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

6.2 Guest Access Speed

The turnaround time for accessing the VM refers to the interval between sending a request and the arrival of the reply. It consists of the time spent for checking the shared buffers and the agent's execution time. To assess the efficiency of the ImEE's interface with the VMI application,

we measure the turnaround time with the ImEE agent performing no task but returning immediately. The result is approximately 265 CPU cycles (or 77 ns) in our setting.

To evaluate the memory-reading performance of the ImEE, we run experiments to evaluate the turnaround time with normal read requests. Table 4 below reports the turnaround time in comparison with LibVMI for the same workload. To make a fair comparison, LibVMI's translation cache is turned on whereas the page-level data cache is turned off.

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

Table 4: Memory read performance comparison.

We have also tested ImEE with the experiment described in Section 2. The experiment shows that the modification on the `cred` address is caught immediately when the malware makes the first attack. Note that with the ImEE support, it takes less than 1200 CPU cycles for the VMI application to get a DWORD from the guest, in contrast to more than 60,000 cycles using LibVMI. The maximum introspection frequency of ImEE based introspection is 2.83 MHz while an introspection using LibVMI in our setting can only achieve 54 KHz in maximum.

6.3 Introspection Performance Comparison

We run introspection tools (syscalldump, pidlist, pslit and credlist) in three settings: within the kernel, with ImEE, and with LibVMI. Since this set of tests concerns with real-life scenarios, we tested LibVMI on both KVM and Xen for completeness. For each of the scenario, we measure the turnaround time of introspection. The time for the processing the semantics and the time for setting up the ImEE/LibVMI are not included in the measurement. Table 5 summarizes the results.

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

6.4 Handling Multiple VMs

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

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

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

7 Discussions

7.1 CPU State

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

Tools	Kernel module	ImEE		LibVMI(KVM / Xen)		
		time	mode	without any cache	without page cache	with all cache
syscalldmp	0.2	2.9	block	28.2 / 43	18.7 / 47	2 / 54
pidlist	10	31.6	traversal	5,887 / 2,180	2,864 / 2,041	1,568 / 490
pslist	10.4	38.6	traversal	8,319 / 1477	2,695 / 1,442	1,672 / 542
credlist	25.3	25.6	hybrid	8,234 / 2,274	7,150 / 2,153	2,215 / 757

Table 5: Kernel object introspection performance (time in μs).

tion both sets of states because they collectively represent the current address translation.

However, for out-of-VM live introspection, it is required that it runs on a core that is independent of the target VM. This limits the introspection's capability to utilize such in-CPU states because there is no mechanism to fetch in-CPU states from another CPU. One possible solution is to preempt the vCPU of the target on a physical core by a more privileged entity such as the hypervisor, trying to preserve as many in-CPU states as possible, including buffers and caches. However, the behavior of the buffers an caches when across VM transition is not fixed. Therefore, without hardware assistance, attempts to implement an ideal solution is likely met with hardware-specific tweaks and hacks, making it very difficult. We leave this issue as future work and present a primitive solution in the Appendix.

7.2 Integration with Existing VMI Tools

The ImEE serves as the guest access engine for the VMI applications without involving kernel semantics. It is not challenging to retrofit exiting VMI tools that focus on high-level semantics to benefit from the ImEE's performance and security. We use VMST [19] as an examples to briefly discuss how to combine a VMI application with the ImEE. When an introspection instruction is executed in VMST, the XED library [8] decides whether a data access should be redirected to the guest VM or not. If so, the code fetches the data from the guest memory by traversing the guest VM's page table in the same way as LibVMI. It is easy to integrate VMST with the ImEE. When a read redirection is generated by the XED library, the code simply issues a memory read request to the ImEE and waits for the reply. With the support from the ImEE, shadow TLB and shadow `CR3` proposed in VMST are no longer needed.

7.3 ImEE vs. In-VM Introspection

Strictly speaking, the ImEE and in-VM introspection systems are not comparable, as they are geared for different purposes. The ImEE is for effective target VM access while in-VM systems are designed for reusing the OS's capability [23, 12] or for monitoring events in the guest [33]. Since Process-Implant [23] and SYRINGE [12] rely on a trusted guest kernel, we compare the ImEE with SIM [33] from the perspective of accessing the target VM memory.

Security. Address space isolation in SIM prevents the target VM kernel from tampering with SIM data and code. In a multicore VM, it does not prevent the target VM kernel from interrupting SIM code execution by using non-maskable interrupts. By knocking down the SIM thread from its CPU core, the rootkit can safely erase the attack traces without being caught. In comparison, the entire ImEE environment is separated from the target VM. It is much more challenging (if not feasible) for the target VM kernel to disrupt the ImEE agent's execution. Note that the manipulation on the page tables backfires on the adversary since they are shared between the adversary and the target.

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

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

7.4 Paging Modes Compatibility

The design of ImEE is by nature compatible with various paging modes such as Physical Address Extension mode (PAE mode) and 64-bit paging. It only requires setting of two additional bits in the control registers, namely `PAE` bit in `CR4` register and `LME` bit in `EFER` register so that the ImEE core runs in the needed paging mode. To prevent the adversary from changing the paging mode, the hypervisor trap access to the above registers. To intro-

spect a 64-bit VM, the agent needs to be compiled into 64-bit code as well. In fact, the ImEE performs better on a 64-bit platform, because there are more general purpose registers available, reducing the number of address space switches, and the PCID can be used to prevent the needed TLB entries from being flushed.

7.5 Architecture Compatibility

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

8 Related Work

The fundamental problem of VMI is to acquire the kernel's semantic by reconstructing the kernel objects. Significant efforts have been spent in directly recovering the kernel's data structures from the raw bytes. It can be based on expert knowledge (e.g., Memparser [10], GREPEXEC [11], Draugr [17], and others [1, 3, 4, 5, 6, 7, 22, 31]) and automatic tools (e.g, SigGraph [27], KOP [13], and MAS [16]). These studies usually involve a large amount of engineering work and are useful for memory forensic analysis. Since they do not emphasize on live memory introspection, the security and effectiveness of accessing the guest's live state are not their main concerns. In general, they are orthogonal to our study in this paper.

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

In-VM Introspection. In general, in-VM introspection schemes aim to save the engineering efforts by relying on the guest kernel's capabilities. Process Implanting [23] loads a VMI program such as *strace* and *ltrace* into the guest VM and executes it with the camouflage of an existing process. SYRINGE [12] runs the VMI application in the monitor VM and allows the introspection code to call the guest kernel functions under a guest thread's context. When the guest kernel is not trusted, the security and effectiveness are totally broken, because it is straightforward for a rootkit to evade or tamper with the introspection. Hence, these in-VM introspection schemes are only useful to monitor the user space behavior in the guest VM. SIM [33] is an in-VM monitoring scheme against rootkits. To run the monitoring code inside the untrusted guest, it creates a SIM virtual address space isolated from the guest kernel. Hooks are placed in the guest to intercept events. The address switches between the kernel and the SIM code is guarded by dedicated gates.

Out-of-VM Introspection. The out-of-VM introspection code stays outside of the target guest. Therefore, it is capable of introspecting the guest VM to detect kernel-level malicious activities without directly facing the attack. Virtuoso [18] generates the introspection code by training the monitor application in a trusted VM and reliably extracting the introspection related instructions from the application. The execution trace is replayed in a trusted VM when performing introspection, whose data accesses are redirected to the guest VM's memory. VMST [19] is another out-of-VM introspection technique. It manages to reuse the kernel code by running the introspection application in a monitor VM emulated by QEMU[9]. A taint analysis runs in the monitor VM and relevant data accesses are redirected to the guest's live memory. Hybrid-bridge [32] is a hybrid approach which combines the strengths of both VMST and Virtuoso. Similarly, the VMI application is running in the trusted monitor VM and the OS code is reused. The kernel data accesses which are related to the monitoring functionality are identified and redirected to the guest kernel memory when needed. EXTERIOR [20] is another space traveling approach inspired by VMST, which supports not only guest VM introspection but also reconfiguration and recovery of the guest VM.

Process Out-Grafting [34] relocates the monitored process from the guest VM to the monitor VM. The monitor VM always forwards system calls to the guest. The guest kernel handles it and return back the results to the monitored process. This approach requires the implicit assumption that the guest kernel is trusted.

TxIntro [28] is an out-of-VM and non-blocking approach designed for timely introspection. It mainly focuses on retrofitting the hardware transactional memory to avoid reading inconsistent kernel states. In its design, the VMI code runs on an implanted core and can also access the guest memory at a native speed. Nevertheless, it lacks sufficient security concerns and also fails to help

the introspection code have a consistent memory view with the guest's. In order to make the VMI code see the same mapping with the guest VM's kernel, the L4 entries of kernel addresses in its page table directly point to the L3 page entries existing in the guest VM's memory. However, there is no guarantee that the guest kernel uses these L3 page entries to translate kernel address indeed during its execution. The L4 page table entries can be changed on-the-fly during an introspection run and the guest kernel can have completely different page tables to translate addresses by using another CR3 value. In fact, unless the introspection code always keeps using the same CR3 value with the guest's directly when reading the guest like ImEE, any change is able to happen on the address mapping used in the guest and it is infeasible for the VMI tool note that. Therefore by following its design, a consistent address translation cannot be achieved and the effectiveness of the introspection is lost.

9 Conclusion

To summarize, we have shown that the software-based address translation widely used in existing out-of-VM introspection systems is not effective to bridge the address gap. We then present the ImEE which provides the architectural support for effective target accesses. The ImEE agent reads the target VM memory at the native speed as its kernel, and the address translation is performed by the hardware in the same way as in the guest. ImEE's native access speed allows consistent memory view with that of the target VM.

Acknowledgement

This work is supported in part by a research grant from Huawei Technologies, Inc.

References

[1] Idetect. http://forensic.seccure.net/.

[2] Lmbench. http://www.bitmover.com/lmbench/.

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

[4] PROCENUM. http://forensic.seccure.net/.

[5] Red Hat Crash Utility. http://people.redhat.com/anderson/.

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

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

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

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

[10] BETZ, C. Memparser. *2005, http://www. dfrws. org/2005/challenge/memparser. shtml* (2005).

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

[12] CARBONE, M., CONOVER, M., MONTAGUE, B., AND LEE, W. Secure and robust monitoring of virtual machines through guest-assisted introspection. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 22–41.

[13] CARBONE, M., CUI, W., LU, L., LEE, W., PEINADO, M., AND JIANG, X. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 555–565.

[14] COKER, R. Bonnie. http://www.coker.com.au/bonnie++/.

[15] CORPORATION, S. P. E. SPEC CINT2006 benchmarks. https://www.spec.org/cpu2006/CINT2006/.

[16] CUI, W., PEINADO, M., XU, Z., AND CHAN, E. Tracking rootkit footprints with a practical memory analysis system. In *USENIX Security Symposium* (2012), pp. 601–615.

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

[18] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 297–312.

[19] FU, Y., AND LIN, Z. 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* (2012), IEEE, pp. 586–600.

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

[21] GARFINKEL, T., ROSENBLUM, M., ET AL. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of NDSS* (2003), vol. 3, pp. 191–206.

[22] GARNER JR, G. M. Kntlist. *2005, http://www. dfrws. org/2005/challenge/kntlist. shtml* (2005).

[23] GU, Z., DENG, Z., XU, D., AND JIANG, X. Process implanting: A new active introspection framework for virtualization. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on* (2011), IEEE, pp. 147–156.

[24] JAIN, B., BAIG, M. B., ZHANG, D., PORTER, D. E., AND SION, R. SoK: Introspections on trust and the semantic gap. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014).

[25] JANG, D., LEE, H., KIM, M., KIM, D., KIM, D., AND KANG, B. B. ATRA: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[26] LEE, H., MOON, H., JANG, D., KIM, K., LEE, J., PAEK, Y., AND KANG, B. B. KI-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *Proceedings of the 2013 USENIX Security Symposium* (2013).

[27] LIN, Z., RHEE, J., ZHANG, X., XU, D., AND JIANG, X. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *NDSS* (2011).

[28] LIU, Y., XIA, Y., GUAN, H., ZANG, B., AND CHEN, H. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2014), IEEE, pp. 416–427.

[29] N. L. PETRONI, T. FRASER, J. M., AND ARBAUGH., W. A. Copilot—a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium* (Aug. 2004), pp. 179–194.

[30] PAYNE, B. D. Simplifying virtual machine introspection using LibVMI. Tech. Rep. SAND2012-7818, Sandia National Laboratories, 2012.

[31] PETRONI, N. L., WALTERS, A., FRASER, T., AND ARBAUGH, W. A. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation 3*, 4 (2006), 197–210.

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

[33] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 477–487.

[34] SRINIVASAN, D., WANG, Z., JIANG, X., AND XU, D. Process out-grafting: an efficient out-of-VM approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 363–374.

[35] SUNEJA, S., ISCI, C., DE LARA, E., AND BALA, V. Exploring vm introspection: Techniques and trade-offs. In *Proceedings of the 11th ACM International Conference on Virtual Execution Environment (VEE'15)* (2015).

[36] WALTERS, A. The volatility framework: Volatile memory artifact extraction utility framework, 2007.

Appendices

A TLB-inclusive Introspection

Since the hardware does not automatically maintain the consistency between the TLB entries and the PTEs in the memory, the target VM's adversary can leverage this hardware behavior to defeat introspection. After accessing a page at VA, the adversary then modifies the PTE to map VA to anther GPA without updating the TLB. An introspection based on the page tables then results in a different memory view from the adversary.

The ImEE scheme can be extended to access the target memory through the TLB used by the running target thread. The hypervisor traps the target's core in the same way as describe before. Note that with the new VPID technique from Intel, the TLB entries used by the target are not evicted due to VM-exit. Our basic idea is to load the agent to the trapped vCPU and to set up the identical context used for TLB lookup.

The strongest method is that the hypervisor injects the introspection agent to the thread's address space, by either directly modifying the target memory or using EPT redirection as in the ImEE scheme. The execution of the agent on the target's core uses the TLB for translation since it is in the same address space. Note that it differs from the in-VM introspection, because the agent execution is independent of the target OS. Obviously, this method is intrusive as it changes the target states and may affect the execution of other target's threads involving the modified memory or mappings.

A non-intrusive way is to run the agent in an external address space. As shown in Figure 12, the hypervisor creates a new page table directory with all its entries being copied from the target's except that one entry is mapped to a separated page storing the mappings for the agent. It loads the target's `CR3` with the new page table base. Note that the PCID in the original `CR3` is not changed. When the agent runs, the TLB entries that match the targeted VAs are used by the MMU (if the entry has the same PCID). In case of TLB misses, the agent still introspects the memory in the same way as in the ImEE. The consistency is maintained because the target's thread is not active during introspection. We have experimented with this method. The result shows that the agent does use the mappings in the TLB to read the global page of the target, instead of following the mapping in the page table.

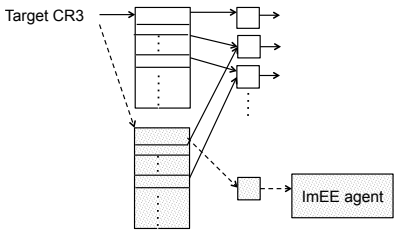

Figure 12: Basic idea of TLB-inclusive introspection. The dashed arrows are used for introspection. The shadowed pages are allocated out of the target's GPA range so that the target's core does not have TLBs for the page table pages.

CAVEAT. The two methods above are only applicable to check the intercepted thread. The adversary can still use a secret PCID to hide its TLBs. It remains as a challenging problem to detect those entries. TLB-inclusive introspection is not equivalent to checking the mappings inside the TLB. Without using special hardware techniques, it is infeasible to for software to inspect every TLB entries.