RapidVMI: Fast and multi-core aware active virtual machine introspection

Thomas Dangl
td@sec.uni-passau.de
University of Passau
Passau, Germany

Benjamin Taubmann
bt@sec.uni-passau.de
University of Passau
Passau, Germany

Hans P. Reiser
hr@sec.uni-passau.de
University of Passau
Passau, Germany

ABSTRACT

Virtual machine introspection (VMI) is a technique for the external monitoring of virtual machines. Through previous work, it became apparent that VMI can contribute to the security of distributed systems and cloud architectures by facilitating stealthy intrusion detection, malware analysis, and digital forensics. The main shortcomings of active VMI-based approaches such as program tracing or process injection in production environments result from the side effects of writing to virtual address spaces and the parallel execution of shared main memory on multiple processor cores.

In this paper, we present *RapidVMI*, a framework for active virtual machine introspection that enables fine-grained, *multi-core aware* VMI-based memory access on virtual address spaces. It was built to overcome the outlined shortcomings of existing VMI solutions and facilitate the development of introspection applications as if they run in the monitored virtual machine itself. Furthermore, we demonstrate that hypervisor support for this concept improves introspection performance in prevalent virtual machine tracing applications considerably up to 98 times.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**.

KEYWORDS

virtual machine introspection, security, virtualization, second level address translation, semantic gap

ACM Reference Format:
Thomas Dangl, Benjamin Taubmann, and Hans P. Reiser. 2021. RapidVMI: Fast and multi-core aware active virtual machine introspection. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3465481.3465752

1 INTRODUCTION

Virtual machine introspection (VMI) is defined as the external monitoring of virtual machines [6]. This process grants a VMI application an untainted external view of a target virtual machine (VM) and enables observation and analysis of the target's behavior. The use of this technique is appealing for many applications such as malware analysis, intrusion detection, and digital forensics [10]. Advantages of the technique include the inherent isolation provided by the virtualization, the stealthiness of the observation, and a view on the monitored system that is protected from malicious manipulation.

In general, VMI-based monitoring mechanisms fall into one of two groups [13]: *Passive* (also called *asynchronous*) VMI mechanisms analyze main memory based on external events, such as a periodic timer. *Active* (also called *synchronous*) mechanisms interfere with the control flow of the monitored VM, e.g., by placing breakpoints inside the monitored VM, such that monitoring may occur at specific places in the control flow of the monitored VM.

Some active mechanisms are *intrusive* mechanisms that modify the main memory of the monitored VM. Examples in this category include injecting an agent into the VM and tracing program execution using breakpoints in main memory, whose handlers remain outside the VM. However, the practical applicability of this intrusive introspection to real-world systems is limited by the following problems that originate from the active nature and are typically not addressed in traditional VMI tooling:

First, when tracing a process using VMI, e.g. to operate high-interaction honeypots [23], the VMI application has to translate the logical address where it wants to place a breakpoint to a physical memory page in which the instruction is injected. This target page may be part of multiple virtual address spaces, because it belongs either to a shared library, to an intentionally shared memory area, or to memory marked read-only (and duplicated later with a copy-on-write strategy) after a *fork* system call.

As a consequence, the breakpoint is implicitly placed in all processes that use this shared memory region. Hence, we cannot monitor only a specific process, and this results in more tracing overhead and potentially unwanted side effects. At present, this issue is completely unaddressed by both academic and commercial VMI solutions. Regarding this problem, we demonstrate a solution called *process-bound introspection* that separates memory modifications based on address spaces.

Second, when injecting shellcode into a VM, e.g. to bootstrap an in-guest agent [4], the shellcode typically consists of multiple instructions and the injection will likely overwrite more than one instruction in the target location. Even if we limit the injection to a single virtual address space (solving the first problem), multiple threads of the same process may concurrently be executing the overwritten instructions on multiple CPU cores, and, as a result, the injection may lead to non-deterministic program behavior. The common solution of simply pausing the target VM is not sufficient, as even then, the target can be resumed with an instruction pointer somewhere within the injected shellcode. With our *core-selective introspection* approach in *RapidVMI*, we show how to inject arbitrary

ARES 2021, August 17–20, 2021, Vienna, Austria

ACM ISBN 978-1-4503-9051-4/21/08.
https://doi.org/10.1145/3465481.3465752

code on multi-core systems using the *altp2m* mechanism introduced by *DRAKVUF* and applying synchronization to the code injection to avoid conflicts with the original mapping [15, 16]. We define *multi-core aware* as a property of active introspection mechanisms that enables *process-bound* and *core-selective* introspection.

Third, active VMI mechanisms introduce a large performance penalty for the monitored VM. Each time the monitored VM is intercepted, it is paused while the VMI application processes the event. We show how to dramatically reduce the amount of time the VM is paused by introducing a minimal set of performance-enhancing VMI operations that handle CR3 write and EPT violation events directly within the hypervisor instead of within the VMI application. We refer to the solution that facilitates the use of these operations as *Hypervisor-based switching*.

Our *RapidVMI* approach extends state-of-the-art VMI solutions for active introspection and solves the aforementioned problems. We demonstrate its efficiency by providing a proof-of-concept implementation for the Xen hypervisor as open-source software. The core contributions of this paper are as follows:

- An approach for process-bound injection into shared guest memory based on EPT pointer switching;
- Core-selective code injection on concurrent systems based on vCPU-specific altp2m mappings;
- Performance enhancements of more than an order of magnitude by handling specific VMI events (CR3 write and EPT violation) directly in the hypervisor.

The remainder of the paper is structured as follows: In Section 2 we provide a brief summary of relevant background. The basic concepts of our active introspection approach are discussed in Section 3. In Section 4 we present our proof-of-concept implementation for the Xen hypervisor and in Section 5 we assess the solution in terms of performance and correctness. We discuss and compare related approaches in Section 6. Finally, Section 7 concludes the paper.

2 BACKGROUND

In this section, we present relevant background on hardware-assisted virtualization on the Intel x86 architecture and on active virtual machine introspection.

2.1 Hardware virtualization on the x86 architecture

Intel VT-x and *AMD-V* enable hardware-level virtualization on the Intel x86 architecture [30]. In this paper, we use the terminology specific to Intel VT-x.

The second generation of hardware-assisted virtualization introduced *Intel Extended Page Tables* (EPT) and *AMD Rapid Virtualization Indexing* (RVI), which implement the concept of *Second Level Address Translation* (SLAT). Instead of only translating virtual addresses to physical addresses, the CPU's MMU will first translate virtual addresses to guest-physical addresses and then guest-physical addresses to physical addresses. Since the MMU performs these translations sequentially, one refers to SLAT as *Nested Paging*.

Figure 1 shows the process of address translation with SLAT when using Intel EPT. Page Map Level 4 (PML4) is the top-level paging structure on x86-64, which refers to Page Directory Pointer Tables (PDPT). Each PDPT contains references to the actual Page Directory Tables (PDT), which in turn refer to the Page Tables (PT). With EPT, the guest PML4 maps guest-virtual to guest-physical addresses, and the hypervisor-managed EPT PML4 maps guest-physical to machine-physical addresses. The pointer to the top-level paging structure is called Extended Page Table Pointer (EPTP). In the following, we refer to an EPT PML4 configuration as a *view* according to the *altp2m* terminology.

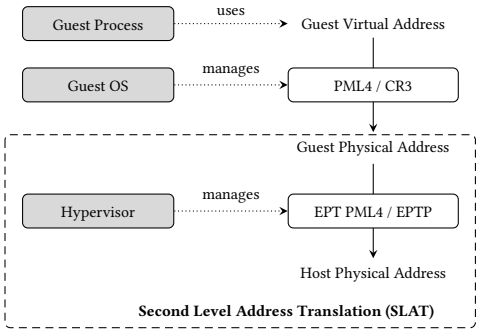


Figure 1: Address translation with SLAT (Intel EPT)

Both hardware vendors support memory access permissions within the SLAT. For example, when writing to a page marked as read-only using EPT, the translation will fail and cause an EPT violation, which triggers a VM-exit and executes a virtualization exception in VMX root mode. It has been known for some time that this concept can be used to hide code in virtual machines by creating complementary views for read, write, and execute and dynamically switching between them [34]. Since this increases stealthiness and resilience over traditional in-kernel protection, it is of no surprise that the method has been adopted for kernel hardening [25]. Other techniques can also be employed to achieve similar results such as *Shadow Walker* [28]. However, our work will focus only on modifications using Intel EPT.

The Xen hypervisor introduced the *altp2m* mechanism [15], which enables managing the guest-**p**hysical **to m**achine-physical mapping with multiple EPT (multiple *views*) for a single guest. It abstracts over the modification in the EPT such that the user of this API needs to specify only for which page he wants to make modifications, and Xen will automatically update all affected paging structures. The most common application of this mechanism is in the *DRAKVUF* malware analysis system, which uses it to remap modified pages in the EPT and change their memory permissions [16].

2.2 Active virtual machine introspection

The main problem every VMI application has to solve is the *semantic gap*, the problem of extracting high-level semantic information from low-level data sources [6]. This semantic gap can be divided into two problems: The *weak semantic gap* is the challenge of creating *VMI-based* tools. The *strong semantic gap* refers to the open problem of protecting these solutions from interference. Over the years significant progress has been made in addressing this issue, so much that the *weak semantic gap* can now be considered "a solved

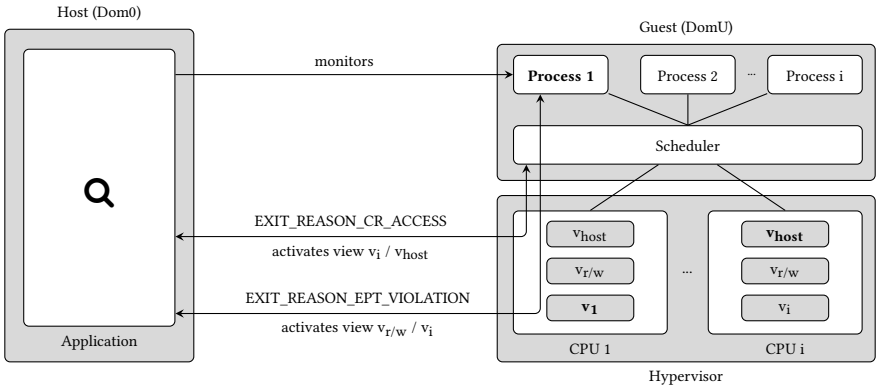


Figure 2: Process-bound and core-selective introspection architecture

engineering problem" [13]. Many of the proposed solutions for this problem can be considered active, which means they aim to overcome the external view by interacting directly with the underlying system using active VMI mechanisms, for example by placing components, hooks, or software breakpoints in the monitored virtual machine [5, 8, 9].

As previously mentioned, virtual machine introspection can be *asynchronous/passive* and *synchronous/active* [13]. *Active* introspection requires interception points in the monitored system that trigger a context switch to the hypervisor. Such a context switch is referred to as *VM-exit*. It is triggered on predefined events such as the execution of privileged instructions, interrupts, or others. A *VM-exit* traps to *VMX root mode*, thereby transferring control to the hypervisor, and executes the configured handler.

The conditions that cause a *VM-exit* may be configured in the *Processor-Based VM-Execution Controls* for Intel processors [11]. Other *VM-exit* reasons that may not be configured include the execution of privileged instructions, e.g., *CPUID*, and non-maskable interrupts, for example, *breakpoint exceptions (#BP)* caused by the execution of an *INT3* instruction.

A commonly used *VM-exit condition* in virtual machine introspection is the monitoring of writes to the *CR3* register. Since this register is the page table base register (*PTBR*) in the x86 architecture, it must hold the currently active page table. By exiting on writes to this register, we can synchronize the VMI application with the guest's scheduler [23]. As this control register may only be updated in kernel mode, we can extract further information such as the current scheduler state from the kernel.

3 ACTIVE INTROSPECTION

In this section, we introduce our solutions to the identified problems of active introspection.

For our paper, we assume that the system provides hardware support for Intel EPT. The monitored VM (a DomU) is referred to as the *guest* whereas the VM that performs the introspection, in our case the Dom0, is called the *host*.

Our introspection architecture does not necessitate any modifications to the operating system of either the guest or the host. Furthermore, the only knowledge we require about the guest is the ability for the VMI application to iterate process structures, which is implemented in readily available VMI libraries for all major operating systems [18].

3.1 Process-bound introspection

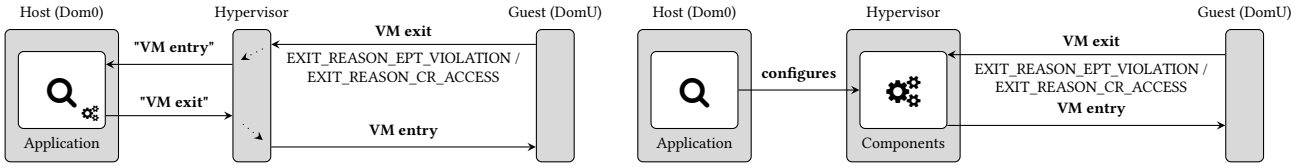
The first problem we want to tackle is that of code injection that remains local to a specific process. VMI write operations affect the underlying physical memory, and the guest OS can map a single guest-physical memory page into multiple virtual address spaces. For example, dynamically linked shared libraries and explicitly allocated shared memory regions make use of that feature. Any VMI write operation will directly modify the underlying physical memory pages, and these changes will be visible in the virtual address spaces of all processes that have this page mapped.

The same problem arises after a *fork* system call, which is usually implemented with lazy memory copying: Parent and child process share the same physical address space, with all pages marked read-only, and a *copy-on-write* (CoW) mechanism duplicates the pages upon modification. However, such CoW mechanisms only apply to guest-internal modification, not to the hypervisor-level page modifications using VMI, which will not cause a *page fault (#PF)* within the guest [12] and not trigger the CoW duplication.

Due to the modification on the original page, the system may suffer instabilities when the modified code is executed by other vCPUs in parallel or by the same vCPU in another process. The issue becomes evident when considering code injection into a shared library such as *libc*, which most user-mode applications typically have mapped in virtual memory.

To address this issue, we create a new EPT view in which we remap the affected page to a shadow page when writing to virtual memory using introspection. Instead of performing the write on the original page, we redirect this and all following write operations on the respective page to this newly created page.

Furthermore, we enable *CR3-store exiting* in the *Primary Processor-based VM-Execution Controls* [11]. When the scheduler writes the PML4 address to the CR3 register, a VM-exit will occur. At this point, the hypervisor can load the appropriate view for this process to the EPTP index using the *vmwrite* instruction. If there are active



(a) External Altp2m

(b) External Altp2m with hypervisor-based switching optimization

Figure 3: Altp2m inter-VM control flow on VM-exits

modifications for this process, we must use the modified view. Otherwise, we may apply the host guest-physical-to-physical mapping instead.

Since the new view will have its modifications on duplicated pages, we essentially separate the result of VMI write operations for each address space. Figure 2 shows the control flow between the guest and host system when performing the process-bound EPTP switching in our architecture.

3.2 Core-selective introspection

The second problem we aim to address is the code injection on multi-core systems. Because all virtual processor cores use the same main memory, our injected code is visible to every processor core. In many situations—mostly when the injected code has side-effects—this can be highly problematic as it might leave the system in an undefined state.

Another reason for core-selective code injection can be the desire to limit modified code to a single processor core for enhancing performance. One example of such a use-case is a debugger for VMs, which sets breakpoints specific for the monitored core. Since this avoids context switches for non-affected processor cores, our approach reduces the overall performance overhead of the monitored machine.

For both the application and the removal of a code injection, we must ensure that program execution remains synchronized with the injected code, i.e., we must make sure that no thread has its instruction pointer inside our modification at the time of the application/removal. As this problem requires knowledge about the inner state of the guest VM machine, we discuss its details in Section 4.2.

Instead of directly modifying the physical memory backed by the respective page, we show a method that involves manipulating the address translation in the monitored VM such that injected code remains local to one core. With this goal in mind, we create an EPT PML4 for each logical processor core that reflects the currently active modifications on that specific core. This concept, which expands upon the *altp2m* architecture [15], is visualized in Figure 2.

To apply a core-selective code injection, we must first check if we already remapped the affected page. If this is the case, we can directly perform the modifications on the copied page. Otherwise, we must create a new guest-physical page, duplicate the original contents onto this page, and finally remap the page in the EPT structure of this processor core to point at our duplicated page instead. To remove a core-selective code injection, we must undo the modification on the duplicated page. If this code injection is the last that affects the duplicated page, we restore the original mapping in the EPT configuration of this processor core. Finally, we can delete the duplicated page from guest-physical memory.

We set the permissions of this remapped, core-specific view to execute only and those of the original, unmodified EPT view to read and write. As explained earlier in Section 2.1, we can now hide the injected code by switching between those views on the violation.

3.3 Hypervisor-based switching

The third problem we intend to solve is the huge performance impact caused by context switches between the hypervisor and the VMs. Figure 3(a) shows the usual interaction between monitoring application and guest system in case an event (EPT violation or CR3 access) on the guest triggers a *VM exit*. Note that on the left side, the "VM entry" and "VM exit" between host and hypervisor do not refer to these VMX CPU instructions, but to a para-virtualized equivalent.

In this paper, we focus on reducing the required context switches between the hypervisor and host system. Note that this interaction is significantly more expensive than the interaction between guest and hypervisor, even if both comprise a pair of *VM exit* and *VM entry* operations. While the hypervisor immediately reacts to the *VM exit*, the *VM entry* on the host includes complex steps not explicitly shown, like a host kernel driver receiving the event from the hypervisor and the host scheduler activating the monitoring application. Nevertheless, optimizing for the guest–hypervisor transition still merits discussion in Section 5.3.

To reduce context switches between hypervisor and host, we introduce two optimizations—called *hypervisor-based switching* in the following—that operate by configuring the hypervisor with predefined actions from the host VM:

First, we demonstrate how to short-circuit EPTP switches required due to EPT violations in our architecture. For each virtual core, the host configures the hypervisor with the accompanying read/write and execute views. When an EPT violation occurs within the guest and thus transitions to the hypervisor, we check if there is a matching entry for this core such that we can handle the violation inside the hypervisor without context switching to the host VM.

Second, we extend this core-specific configuration with mappings for specific processes. To do this, we convert the configuration entry to a list of mappings and annotate each entry with the page table directory of the targeted process. On EPT violation, we walk

this list until we find a matching entry that is applicable for the current process. Additionally, the hypervisor must enable the *VM-exit condition* for CR3 writes and switch the EPTP index based on the newly active process on such writes.

Figure 3 shows how these optimizations avoid performance overheads caused by context switches on EPT violations and CR3 writes. In *(a)*, we see the control flow without in-hypervisor components. *(b)* depicts the optimizations using short-circuited EPTP switches using in-hypervisor components.

4 IMPLEMENTATION

We present our proof-of-concept implementation of the previously introduced concepts in this section. While our implementation targets the Xen hypervisor, the underlying concepts can easily be adapted to other hypervisors with SLAT implementations such as KVM.

4.1 Altp2m views

We make use of the following *altp2m* views: The first view, v_{host}, represents the original host guest-physical to machine-physical mapping. It may be used to perform single-steps for breakpoints, as shown in *DRAKVUF*, or as the default view on processes without active modifications [16].

Next, the view $v_{r/w}$ has the same mapping as v_{host}. However, we remove the execute permissions for each page that contains active modifications. Due to these missing execute permissions, the guest causes an EPT violation when attempting to execute such a page, thereby enabling a switch to the v_i view, which contains the active modifications.

Finally, each view v_i maps to the duplicated, modified pages active on processor core *i*. When there are any current modifications, this is the default view for the processor core *i*[1]. Because the EPT view has no read-write permissions set on modified pages, any attempt to read the modified code leads to an EPT violation, which we use to switch to view $v_{r/w}$, thus hiding the modifications from the guest. Note that this behavior and the fact that code pages are rarely being read or modified allows us to keep this view active most of the time without the VM-Exit-associated overhead.

Table 1: Overview of the implemented *altp2m* views

ID	Name	Type	Read	Write	Execute
#0	v_{host}	Host P2M	✓	✓	✓
#1	$v_{r/w}$	Read-write P2M	✓	✓	✗
#i+2	v_i	Execute P2M for core i	✗	✗	✓

An overview of the purpose and page permissions of these *altp2m* views is depicted in Table 1.

4.2 Host components

When implementing the host components of the VMI architecture, we have to address three problems:

[1]The consequence of this partitioning is that the current implementation cannot have different modifications on the same page in multiple processes. A sensible improvement could be the dynamic creation of new views at this point if there are free EPTP indices.

First, to modify a page according to the principles outlined in Section 3, it has to be duplicated and remapped in the execute views v_i. When we want to copy the page, we first determine the start of the page by aligning the virtual address to the page size and translate it to a physical address externally using VMI. All following modifications to the page are applied to the duplicated page instead. *DRAKVUF* introduced this technique in 2016 [16]. The *altp2m* mechanism of the Xen hypervisor already implements the remapping for the EPT extension.

Second, the successive application of modifications is considered non-commutative. Of course, this is also true for the removal. To ensure the correct removal order, last-in-first-out (LIFO), we keep track of the application order of the modifications. Finally, when removing a modification, we ensure that all overlapping changes, which we applied at a later point, are removed first.

Third, as outlined earlier, it is required to make sure that modifications are only applied once no thread is executing the affected area inside the guest virtual machine. To do so, we require inside knowledge about the guest operating system. In particular, we must know where the kernel stores the threading structures and the associated register values so that we can extract the current instruction pointer. When applying or removing a modification, we pause the virtual machine to check for intersections of the instruction pointers of all threads with the injected code. To address potential conflicts, we provide an API, which the monitoring application can use to implement one of two strategies:

(1) Delaying the injection until there are no further intersections. This is done by repeatedly pausing the monitored virtual machine, iterating over all tasks while comparing their instruction pointer against the modified memory region and finally resuming the virtual machine. The injection may be applied when no task is currently executing the modified memory.
(2) Relocating the injected code outside the intersected range. In this strategy, we use the highest instruction pointer of all tasks that intersect the modified area as the new starting location of modification. This technique is repeated until no further intersections are found. As the method solely relies on the instruction pointers of the tasks, we avoid the additional complexity of disassembling the affected area as long as the program is in a well-defined state.

Which of these solutions is more suitable highly depends on the situation. For example, it is possible to relocate shellcode that aims to perform a given task in the monitored virtual machine. Doing so, however, might change the semantics of the code. Given a breakpoint that is placed at a particular location in the control-path, relocation would move it to another position in the code, thereby changing the location at which the VMI application is inspecting the state of the monitored virtual machine. As the injection mechanism does not know the program semantics, we consider this choice to be the responsibility of the VMI application developer. In general, strategy 2 is preferable since it avoids multiple synchronization attempts in many cases.

4.3 In-hypervisor components

As explained in Section 3.3, we implement certain aspects of our solution in the Xen hypervisor to speed-up the execution of the guest virtual machine. These aspects include the dynamic switching of *altp2m* views on EPT violation and CR3 write events.

To achieve this goal, we patch the Xen hypervisor by hooking *monitor_domctl*, which sends the VMI events to monitoring virtual machines. When receiving such an event, we have to handle two cases:

First, if a CR3 write caused this event, we know that the guest's scheduler became active. Here we need to select the appropriate view for this process: In case there are modifications for this address space, we need to switch the *altp2m* view to v_i, with *i* being the core the CR3 write occurred on. Otherwise, we must check if there was an active modification on the previous address space and switch to v_{host} instead.

Second, if an EPT violation caused the event, we check if it occurred on the v_i or the $v_{r/w}$ view. For this case, we emulate the memory access on the opposing view. If the violation would not have occurred on that view, we can switch the active *altp2m* view to this one safely.

If our in-hypervisor components handle any of these cases, we drop the event from the queue instead of forwarding it to the host system. By avoiding the overhead of this synchronous operation, we improve performance in the guest virtual machine significantly.

To configure these in-hypervisor components from the host, we introduce additional Xen hypercalls (see Appendix A): *xc_altp2m_add_switch* / *xc_altp2m_remove_switch*. These hypercalls allow the host to set a pair of complementary view identifiers individually for each processor core and process. We store this information in the hypervisor as a linked list and use it during the event processing, as described above.

5 EVALUATION AND DISCUSSION

In this section, we assess our proof-of-concept implementation based on performance and stealthiness metrics and discuss potential limitations.

5.1 Performance

All of the following measurements are performed on a DomU equipped with three physical cores of an Intel i7-6700K processor and 4096 MB of RAM, swapping is disabled. The Dom0 is assigned the remaining core of the processor and 2048 MB of RAM. For this evaluation, we used Xen 4.14 with our patches applied. The guest and host system run the Linux kernel with versions 4.4.0 and 4.19.0 respectively.

A common use-case for code injection in active introspection is setting breakpoints that can be intercepted by the VMI application to allow synchronous mechanisms at specific locations in the control-flow of the guest virtual machine [26]. Other applications may include process forking and agent injection. As the latter are typically less performance-critical, because they only introduce a setup-cost, we base our performance evaluation on breakpoints. Many VMI-based monitoring applications intend to only trace certain suspicious processes, e.g., the user's bash, to monitor for malicious behavior, which is why part of our evaluation will also focus on the implications of this aspect.

In the subsequent considerations, we assess the breakpoint performance under the following conditions:

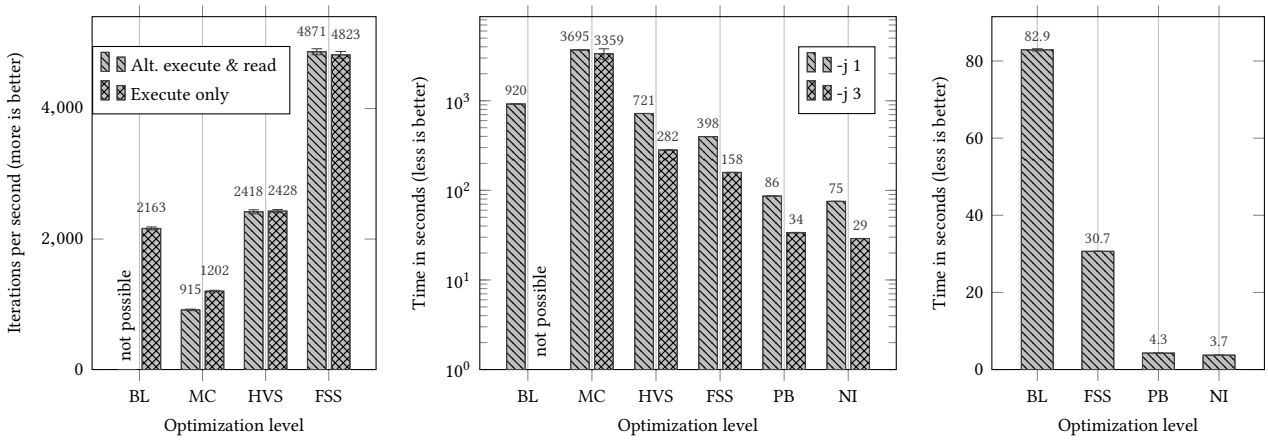
- **NI (No introspection):** This is a measurement of the given task without any on-going introspection.
- **BL (Baseline):** Initially, we employ the naïve approach of placing the breakpoint directly on the physical page, similar to the breakpoints examples of *libvmi*. When the breakpoint is hit, we execute the replaced instruction inside the VM using the monitor trap flag [21].
- **MC (Multi-core aware):** This is our implementation based on the concepts shown in Section 3.2. Instead of manipulating the memory, we restore the original instruction by switching the active EPTP index. Here we perform all operations related to *RapidVMI* in the monitoring application in Dom0.
- **HVS (Hypervisor-based switching):** In this variant, we evaluate the optimizations described in Section 3.3 on top of our core-selective breakpoint architecture. In essence, this is equivalent to *MC*, but the operations are performed at the hypervisor level instead.
- **FSS (Fast Single-Step):** On top of our optimizations in *HVS*, we use Sergey Kovalev's fast single-step implementation to reduce the context switch to the host after single-stepping [14]. By pre-configuring the hypervisor with the targeted *altp2m* view after the single-step, we can avoid the costly, synchronous event to the host application.
- **PB (Process-bound):** This measurement is the fully-featured implementation of our concepts and performs the process-bound monitoring of the current SSH session's bash process.

First, we consider breakpoints in a synthetic benchmark: We execute a program in the guest virtual machine that starts and pins a thread to each vCPU. The VMI application places a breakpoint on a procedure within this program. We measure alternating execution/read access on this function to determine the highest additional overhead caused by the VM-exit required to switch to the correct EPTP view. Additionally, we consider execution-only access as this is the most probable scenario for code modifications. By comparing the iterations achieved in a fixed time frame of one second, we can determine the relative performance impact of the breakpoint mechanism. A pseudocode representation of this synthetic benchmark is given in Appendix B.

Figure 4(a) depicts the results of this synthetic benchmark in the two scenarios with a sample size of 100 each. By comparing

Table 2: Evaluation configurations

	NI	BL	MC	HVS	FSS	PB
VMI-based tracing	✗	✓	✓	✓	✓	✓
Core-selective (3.2)	✗	✗	✓	✓	✓	✓
HV-based switching (3.3)	✗	✗	✗	✓	✓	✓
Fast Single-Step	✗	✗	✗	✗	✓	✓
Process-bound (3.1)	✗	✗	✗	✗	✗	✓



(a) Micro-benchmark (synthetic) (b) Macro-benchmark (gcc / Linux kernel) (c) Macro-benchmark (tar + gcc / Jansson)

Figure 4: Consolidated benchmarks of the RapidVMI architecture

Table 3: VM-Exit reasons (gcc 8.3.0 / Jansson)

	CR3 Write	read	write	open	close	exec	Σ
FSS	1,310,376	21,230	8,694	13,230	15,993	1,306	60,453
PB		0	0	0	0	7	7

the results of MC and HVS, we determine that our hypervisor-based switching approach practically eliminates the overhead of EPT violations in the VMI context, that is to say, their impact is negligible.

Second, we perform the breakpoint measurements under real-world scenarios: We place a breakpoint on every system call handler to monitor any system call executed by the guest. Under those conditions, we compile the Linux kernel 5.11 using the built-in "tiny" configuration. The build process in this measurement is performed both single-threaded (*-j 1*) and multi-threaded (*-j 3*) with gcc version 8.3.0. Because the breakpoint has to be removed from physical memory in the baseline measurement (BL) to perform the single-step, we cannot use this technique in the multi-threaded context. We measure these build times separately ten times each. Note that the measurements of the variant MC were only performed three times due to the enormous run time.

The results of these measurements are shown in Figure 4(b). By comparing the build times of MC and FSS, we can see a speed-up of more than an order of magnitude for system-wide system call tracing. Furthermore, the results for process-bound system call tracing indicate that depending on the ratio of monitored system calls (PB) to total system calls (FSS) our method allows for near-native execution speed during introspection. In the scenario of building the Linux kernel using gcc, this ratio was determined to be 0.0024% during our experiments. Finally, we performed additional measurements of the system-wide tracing where the method of configuring a VM-Exit for CR3 writes was not used. This allowed us to determine the overhead of this technique at 2.21% for the single-threaded case and 0.20% for the multi-threaded case. When monitoring a relatively inactive process (such as the user's bash) during heavy computational load, our optimized solution (PB) significantly outperforms the naïve approach (MC) by a factor of 42 in the single-threaded case and a factor of 98 in the multi-threaded case.

Third, we evaluate our active introspection solution targeting the process-bound monitoring of a selected subset of system calls (read, write, open, close and exec) during extraction and compilation of the *Jansson* C library for JSON (de-)serialization. This allows a direct comparison of the optimization aspect of our work to the results obtained by Taubmann et al. [26].

The results of these measurements are given in Figure 4(c). When extracting the source code from the *tar.bz2* archive and compiling it, we measure an overhead of 2119% in the unoptimized case. Using all optimizations we obtain an overhead of 720% during system-wide tracing. When considering process-bound tracing, the overhead is reduced to 14% compared to the 58% obtained by Taubmann et al.

In addition, we have also broken down the VM-Exit reasons during this monitoring. This itemized data is shown in Table 3. From these results, we obtain a ratio of monitored to total system calls of 0.00012% during these experiments.

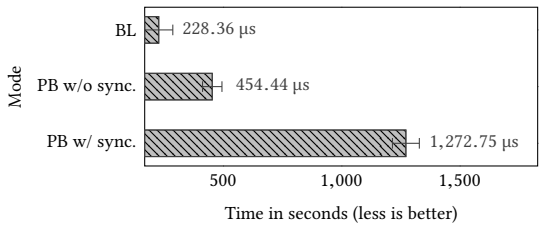


Figure 5: Setup cost of a one-page modification

Finally, we conducted measurements in regard to the setup cost of our VMI mechanism. Since we have to duplicate the targeted page first and update the EPT to reflect the change, our approach has significant overhead over writing to the original physical page directly. Additionally, we take into consideration our synchronization that prevents desynchronized execution in the guest virtual machine.

As seen in Figure 5, the technique of duplicating and remapping the affected page introduces a performance penalty of around 99%. In the case where we also apply our synchronization mechanism, this further increases to 557%. The requirement of keeping the virtual machine paused during this analysis explains the large overhead of the synchronization mechanism. Nevertheless, we do not expect issues in real-world applications stemming from the 1.3 milliseconds initial setup-up time of the modification.

5.2 Stealthiness

Like many VMI architectures, our solution is not entirely stealthy due to implementation details. The first clue that can give away the presence of active introspection is the hypervisor itself [27]. Many approaches to detect virtualization from the inside are known. Most of them revolve around timing attacks that are possible due to the enforced isolation [20], for example using the repeated execution of privileged instructions such as *CPUID* that cause a *VM-exit* on Intel CPUs. By measuring the execution time, it is possible to detect the overhead caused by virtualization. Since modern processors have many independent methods of measuring time, such as *high-precision event timers*, it is becoming increasingly difficult to hide the presence of the hypervisor.

Since the mere presence of a hypervisor, however, is usual in many environments like cloud computing, we cannot conclude that there is ongoing introspection by this fact alone. In contrast, there are also specific timing attacks that can detect out-of-guest monitoring techniques such as virtual machine introspection [19]. An instance of this class of attacks to notice introspection from inside the guest is a timing attack on the *VM-exit condition* of *MOV-to-CR3* [29], which is a common synchronous introspection operation as discussed in Section 2.2. Because our approach also makes use of this operation to synchronize with the guest's scheduler, we remain vulnerable to this type of attack.

Furthermore, implementation-specific details may lead to the detection of our approach. The use of SLAT to separate read/write from executing memory access can be detected using self-modifying code. For example, the *Microsoft Kernel Patch Protection* uses this technique in recent versions of the *Windows NT* kernel. Because write operations of a program on its description cause an EPT violation, which our implementation handles by switching the *altp2m* view, the overwritten instruction is only present on the $v_{r/w}$ view, not the v_i views. Hence, the code is no longer self-modifying and follows a different control-flow path[2].

Next, the same separation can be detected using timing attacks on the overhead of the hardware virtualization. One can achieve this either directly, e.g., by alternating read and execute access on modified pages as seen in Section 5.1, or indirectly by measuring the overhead of invalidated TLB entries due to EPTP switching as seen in [29].

Finally, our approach does not require an agent present in the virtual machine. Therefore, an attacker cannot detect any monitoring tools or components from the inside of the monitored virtual machine, for example, in the case of VMI-based malware analysis.

5.3 Limitations

In our paper, we have assumed a simplified memory management scheme, e.g., we expect affected pages to be paged in. Thus we did not consider some edge cases that may occur in real-world systems.

One situation that may occur is that the guest operating system duplicates a page after the introspection application already duplicated it externally. When this happens, the kernel will inevitably cause a switch to $v_{r/w}$ when reading the page. Since our modification may only reside on one of the executable views v_i and not on the readable view $v_{r/w}$, it is removed from the duplicated view after the guest duplicates the page on its own.

We can avoid this by introducing another view v_k, which is used exclusively in kernel-space. This view resembles the host guest-physical to machine-physical mapping of view v_{host}, but removes all access permissions on the union of all active modifications for any process. Then, when a violation occurs, we can dynamically remap the newly created physical page for the affected process and remove the entry for the old page. However, there are also instances where this improvement falls short, e.g., kernel-same-page merging [3].

Another issue that is currently unhandled is the case of multiple user-mode processes that share the same PML4. This is for example possible after the invocation of *vfork* and before the call to *exec* or *exit* [7]. As we use the address of the page table to perform EPTP switching synchronous to the guest operating system's scheduler, we cannot distinguish these processes and therefore apply the modifications to both. A potential solution that requires insider knowledge about the guest operating system could perform the matching based on the scheduler structures instead, e.g., the *task_struct* under Linux.

The general concepts outlined in this paper are also applicable to AMD CPUs that feature Rapid Virtualization Indexing (RVI). However, it has to be considered that it is not possible to set read-/execute permissions independent of each other [1]. As this issue affects many security applications of hypervisors, the issue has already been discussed and partially addressed [24]. Furthermore, AMD CPUs do not have the *Monitor Trap Flag*, which mandates a different implementation for breakpoints.

With regards to the synchronization mechanism used to eliminate side-effects in the guest virtual machine during code injection, we assume that the introspection application does not apply modifications that span over multiple Single-Entry-Single-Exit regions or unstructured code. We can achieve this by determining suitable injection locations using static analysis. Intuitively, this limitation is also given during normal code injection without any introspection.

6 RELATED WORK

Table 4 shows an overview of the most related work in regards to active virtual machine introspection.

[2]We can address this by emulating write accesses on pages that are executable by applying the write on both the original and duplicated page.

Table 4: Comparison of active introspection approaches

	[18]	[8]	[33]	[16]	[2]	RapidVMI
Physical memory unmodified	❍	●	●	●	●	●
Multi-core aware introspection	❍	❍	◗	◗	◗	●
Process-bound introspection	❍	❍	❍	❍	❍	●
Minimal performance degradation	❍	❍	❍	◗	◗	●

● Applicable ◗ Partially applicable ❍ Not applicable

In 2012 Bryan D. Payne presented a library named *libvmi* based on *XenAccess* [18]. The integrated support of existing memory forensic frameworks such as *Volatility* and *Rekall* made bridging the *semantic gap* in production environments drastically more accessible [22, 31]. While certain aspects of this library can be classified as active introspection, it mostly provided abstractions that enabled the easier development of portable VMI applications.

CXPInspector, a framework for VMI-based binary analysis, has been implemented by Willems et al. in the same year [33]. It is designed to harness the SLAT of modern systems to monitor intermodular method invocation by only keeping certain memory pages, e.g., the user-mode address space of a monitored process, executable. Therefore, it can intercept the virtual machine when it attempts to perform such an intermodular call. However, it offers neither fine-grained tracing capabilities nor the desired performance.

Gu et al. introduced a framework for active introspection called *Process Implanting* [8]. The approach involves hooking the guest operating system's scheduler using *VM-exit conditions* and 'stealing' execution time from a given process. Instead of executing the intended process, the system transfers control to the implanted process. The hypervisor manages the physical pages and the page tables of the implanted process, i.e., it makes them visible when switching to the process and hides them from the guest system after execution. This approach has two main downsides: First, one has to chose the targeted process carefully, because the missing time slices may cause issues on real-time programs. Second, it is not possible to monitor the execution in the guest virtual machine using the VMI context, e.g., by setting breakpoints; it can only operate as a non-VMI monitoring tool in this case.

Lengyel et al. presented DRAKVUF, a dynamic malware analysis system based on *libvmi* [16]. Its primary purpose is the analysis of malware samples for the *Microsoft Windows NT* operating system using the Xen hypervisor. The authors implemented the *altp2m* approach for hyper-breakpoints, which allows the application to hide the modified code from the guest virtual machine and increases performance compared to regular single-stepping or emulation. Furthermore, by manipulating the active view separately for each vCPU, the breakpoints became suitable for multi-processor systems. However, as all processors use the same set of *altp2m* views, it was still impossible to perform (overlapping) modifications on a subset of vCPUs. Finally, these breakpoints were still present in every process that had this page mapped.

In 2020 Bitdefender introduced the HVMI core library, which focuses on live protection of monitored virtual machines, unlike existing VMI projects that mostly revolve around debugging and tracing of VMs [2]. Their solution can harden a virtual machine against kernel exploits, rootkits, buffer overruns, and more. This use-case is distinct from typical VMI applications in which the control flow is being observed, not manipulated. However, their approach does not address the prevalent limitations in concurrent systems as it performs neither synchronization nor process-bound modifications.

Westphal et al. showed a monitoring language for virtual machine introspection and implemented a prototype for the VMware KVM hypervisor [32]. Their approach of programming the hypervisor with predetermined actions for VMI operations is similar to our optimizations based on *In-hypervisor components* (see Section 4.3). However, we do not intend to provide a fully-featured language. Instead, we aim to factor out the operations with the highest performance impact to the hypervisor level.

Li et al. discussed the implications of the performance overhead that occurs on virtualized systems due to VMX context switches [17]. They demonstrated that using the *VMFUNC* instruction for implementing a cross-world call mechanism can decrease latency by up to 80% compared to traditional solutions.

7 CONCLUSION

In this paper, we discussed common problems in traditional active introspection approaches that limit the practical applicability in real-world systems.

We highlighted the presence of unwanted side effects and tracing overhead caused by the direct modification of physical memory using VMI. To this avail, we have introduced the concept of *process-bound introspection* that separates the memory access of introspection applications based on virtual address spaces of the monitored system.

Furthermore, we elaborated on the implications of concurrent execution in an active introspection context. In this regard, we examined approaches to ensure correct program execution and introduced *core-selective introspection*, the property of VMI-aided memory modifications that apply only on selected processor cores. We realized this concept by maintaining the SLAT page tables separately for each processor core.

With *RapidVMI*, our framework for active virtual machine introspection, we implemented these concepts on an off-the-shelf hypervisor and demonstrated their practical use for introspection applications. Finally, we introduced optimizations in the form of precomputed state transitions at the hypervisor-level, which reduce the introspection overhead by a factor of up to 98.

ACKNOWLEDGMENTS

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 361891819 (ARADIA).

REFERENCES

[1] Advanced Micro Devices 2019. *AMD64 Architecture Programmer's Manual.* Advanced Micro Devices. Volume 2, 499–507.
[2] Bitdefender. 2020. Hypervisor Memory Introspection - Specification. https://hvmi.readthedocs.io/en/latest/index.html. Accessed: 2020-09-02.
[3] Jonathan Corbet. 2008. /dev/ksm: dynamic memory sharing. https://lwn.net/Articles/306704/. Accessed: 2020-10-08.
[4] Thomas Dangl, Benjamin Taubmann, and Hans P. Reiser. 2021. Agent-Based File Extraction Using Virtual Machine Introspection. In *Secure IT Systems - 25th Nordic Conference, NordSec 2020.* Springer, Virtual Event, 174–191.

[5] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. 2014. HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated in-VM Management. In *Proc. of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Philadelphia, PA, USA, 85–96.
[6] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *NDSS*. The Internet Society, San Diego, California, USA, 16 pages.
[7] Mel Gorman. 2004. *Understanding the Linux Virtual Memory Manager* (first ed.). Prentice Hall, Upper Saddle River, New Jersey, USA, 33–38, 352.
[8] Z. Gu, Z. Deng, D. Xu, and X. Jiang. 2011. Process Implanting: A New Active Introspection Framework for Virtualization. In *2011 IEEE 30th Int. Symposium on Reliable Distributed Systems*. IEEE, Madrid, Spain, 147–156.
[9] Kyle C. Hale, Lei Xia, and Peter A. Dinda. 2012. Shifting GEARS to Enable Guest-Context Virtual Services. In *Proc. of the 9th Int. Conf. on Autonomic Computing (ICAC '12)*. ACM, San Jose, California, USA, 23–32.
[10] Y. Hebbal, S. Laniepce, and J. Menaud. 2015. Virtual Machine Introspection: Techniques and Applications. In *2015 10th Int. Conf. on Availability, Reliability and Security*. IEEE, Toulouse, France, 676–685.
[11] Intel Corporation 2009. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Volume 3C, 48, 111.
[12] Intel Corporation 2009. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation. Volume 2A, 28.
[13] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. 2014. SoK: Introspections on Trust and the Semantic Gap. In *2014 IEEE Symposium on Security and Privacy*. IEEE, Berkeley, CA, USA, 605–620.
[14] Sergey Kovalev. 2019. x86/vm_event: add fast single step. https://patchwork.kernel.org/patch/11297787/. Accessed: 2020-09-14.
[15] Tamas K. Lengyel. 2016. Stealthy monitoring with Xen altp2m. https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/. Accessed: 2020-09-07.
[16] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Proc. of the 30th Annual Computer Security Applications Conference (ACSAC '14)*. ACM, New Orleans, LA, USA, 386–395.
[17] Wenhao Li, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2015. Reducing World Switches in Virtualized Environment with Flexible Cross-World Calls. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 375–387.
[18] Bryan D. Payne. 2012. *Simplifying virtual machine introspection using LibVMI*. Technical Report. Sandia National Laboratories.
[19] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. 2011. NEther: In-Guest Detection of out-of-the-Guest Malware Analyzers. In *Proc. of the 4th European Workshop on System Security (EUROSEC '11)*. ACM, Salzburg, Austria, Article 3, 6 pages.
[20] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. 2013. A Survey of Security Issues in Hardware Virtualization. *ACM Comput. Surv.* 45, 3, Article 40 (2013), 34 pages.
[21] Rekall. 2012. LibVMI: Simplified Virtual Machine Introspection. https://github.com/libvmi/libvmi. Accessed: 2021-02-08.
[22] Rekall. 2012. Rekall memory forensics framework. https://github.com/google/rekall. Accessed: 2020-09-02.
[23] Stewart Sentanoe, Benjamin Taubmann, and Hans P. Reiser. 2018. Sarracenia: Enhancing the Performance and Stealthiness of SSH Honeypots Using Virtual Machine Introspection. In *Secure IT Systems - 23rd Nordic Conference, NordSec 2018*. Springer, Oslo, Norway, 255–271.
[24] Satoshi Tanda. 2019. AMD-V for Hackers. Hypervisor Development Hands On for Security Researchers on Windows, Workshop, VXCON. http://tandasat.github.io/VXCON/AMD-V_for_Hackers.pdf. Accessed: 2020-10-08.
[25] Satoshi Tanda, Irvin Homem, and Igor Korkin. 2017. Detect Kernel-Mode Rootkits via Real Time Logging & Controlling Memory Access. In *Proc. of the Annual ADFSL Conference on Digital Forensics, Security and Law*. Scholarly Commons, Daytona Beach, Florida, USA, 31 pages.
[26] Benjamin Taubmann and Hans P. Reiser. 2020. Towards Hypervisor Support for Enhancing the Performance of Virtual Machine Introspection. In *Distributed Applications and Interoperable Systems- 20th IFIP WG 6.1 International Conference, DAIS 2020*. Springer, Valletta, Malta, 41–54.
[27] C. Thompson and M. Huntley. 2010. *Virtualization Detection: New Strategies and Their Effectiveness*. Ph.D. Dissertation. University of Minnesota. Accessed: 2020-09-15.
[28] Jacob Torrey. 2014. MoRE Shadow Walker: TLB-splitting on Modern x86. Black Hat USA Conference.
[29] Tomasz Tuzel, Mark Bridgman, Joshua Zepf, Tamas K Lengyel, and Kyle J Temkin. 2018. Who watches the watcher? Detecting hypervisor introspection from unprivileged guests. *Digital Investigation* 26 (2018), S98–S106.
[30] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (2005), 48–56.
[31] Volatility Foundation. 2009. Volatility memory forensics framework. https://github.com/volatilityfoundation/volatility. Accessed: 2020-09-02.
[32] Florian Westphal, Stefan Axelsson, Christian Neuhaus, and Andreas Polze. 2014. VMI-PL: A monitoring language for virtual platforms using virtual machine introspection. *Digital Investigation* 11 (2014), S85–S94.
[33] Carsten Willems, Ralf Hund, and Thorsten Holz. 2012. *CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring*. Technical Report. Ruhr University Bochum. TR-HGI-2012-002.
[34] Mingbo Zhang and Saman Zonouz. 2016. How to hide a hook: A hypervisor for Rootkits. *Phrack Magazine* 15, 69 (2016), 8 pages.

A HYPERCALLS FOR HYPERVISOR-BASED SWITCHING

```
/*
 * Hypercall to insert a mapping between a page table
 * and a set of complementary views for one processor.
 */
int xc_altp2m_add_switch(
        xc_interface *handle, uint32_t domid,
        uint32_t vcpu_id, uint64_t pgd,
        uint16_t view_rw, uint16_t view_x);

/*
 * Hypercall to remove an active mapping on one processor
 * based on the page table.
 */
int xc_altp2m_remove_switch(
        xc_interface *handle, uint32_t domid,
        uint32_t vcpu_id, uint64_t pgd);
```

B PSEUDOCODE OF SYNTHETIC MICRO-BENCHMARK

```
struct vcpu_result_t
{
        uint8_t bytes[14] = { 0 };
        bool ret = false;
        uint32_t iterations = 0;
};

thread_local vcpu_result_t res { };

__attribute__((noinline)) bool return_fn()
{
        volatile bool ret = false;
        return ret;
}

while (!interrupted)
{
        memcpy(res.bytes, (uint8_t*) &return_fn, sizeof(
             ↪ vcpu_result_t::bytes));
        res.ret = return_fn();
        res.iterations++;
}
```