Retrofitting AMD x86 processors with active virtual machine introspection capabilities

Thomas Dangl[1] (✉), Stewart Sentanoe[1], and Hans P. Reiser[2]

[1] University of Passau, Innstr. 43, 94032 Passau, Germany
{td,se}@sec.uni-passau.de
[2] Reykjavík University, Menntavegur 1, Reykjavík, Iceland
hansr@ru.is

Abstract. *Active virtual machine introspection* mechanisms intercept the control flow of a virtual machine running on top of a hypervisor. They enable external tools to monitor and inspect the state at predetermined locations of interest synchronous to the execution of the system. Such mechanisms, in particular, require support from the processor vendor by facilitating interpositioning. This support is missing on *AMD x86* processors, leading to inferior introspection solutions. We outline implicit assumptions about active introspection mechanisms in previous work, offer constructions for solution strategies on *AMD* systems and discuss stealthiness and correctness. Finally, we show empirically that such retrofitted software solutions exhibit performance metrics in the same order of magnitude as native hardware solutions.

Keywords: virtual machine introspection, monitoring, system security, reliability, stealthiness, cloud computing

1 Introduction

Virtual machine introspection (VMI) is a popular approach for monitoring virtual machines (VMs) at the hypervisor level. VMI is desirable for many practical purposes, including intrusion detection, malware analysis, and main memory forensics. To perform VMI, we have to provide a monitoring application with the means to access a target virtual machine's internal state.

There are two flavors of VMI: *active* and *passive*. *Passive* VMI, or polling VMI, refers to unsynchronized, read-only access to the virtual machine's state. This approach involves periodically polling the virtual machine's memory and registers to gather information about its state. *Active* VMI, or event-based VMI, involves intercepting the virtual machine's control flow and executing custom code in response to specific events. This approach allows for more fine-grained control over the target virtual machine, but can be more complex to implement.

Significant hardware differences exist between *Intel* and *AMD x86* processors, which majorly impact the use of *active* VMI. The usual approaches used by VMI tools and libraries on *Intel* processors do not work on *AMD* processors due to fundamental differences in their design. Specifically, *AMD* processors

support hardware-assisted virtualization through their *Secure Virtual Machine* (SVM) extension and enable *Second Level Address Translation* (SLAT) using *Rapid Virtualization Indexing* (RVI). This is different from *Intel* processors, which use *Virtual Machine Extensions* (VT-x) and support SLAT with *Extended Page Tables* (EPT). Additionally, while *Intel* processors have a feature called the *Monitor Trap Flag* that simplifies single-stepping a virtual machine, *AMD* systems do not offer an equivalent capability.

It is essential to address the current lack of VMI support on *AMD* processors to expand the applicability of VMI-based tools. Notably, within the *SmartVMI* project[3], our objective is the development of VMI toolchains for generating training data sets for the next generation of VMI-based security tools, and neglecting *AMD* platforms as a valuable source of real-system data sets would severely hinder the generalizability of our outcomes. Therefore, this paper introduces a software implementation that retrofits missing hardware features onto current *AMD x86* processors, extending their introspection capabilities. Our approach seamlessly integrates with popular introspection APIs and off-the-shelf hypervisors, enabling developers of introspection applications to quickly port their existing software to *AMD* systems. Our contributions include the following key aspects:

1. We analyse two significant architectural differences between *Intel* and *AMD x86* processors that affect the realization of active introspection mechanisms taking into account related research.
2. We conceptualize a mechanism that addresses the shortcomings of previous approaches through retrofitting virtualization features in software. This mechanism enables virtualized single-stepping on systems that support regular, non-virtualizable single-stepping.
3. Based on this principle, we develop a proof of concept implementation for the *KVM* hypervisor and the *LibVMI* introspection library, based on KVMi [8]. We publish this implementation as an open source software.

This paper is structured as follows: Section 2 provides background knowledge on VMI. Section 3 summarizes existing work that targets VMI on *AMD* processors and highlights their limitations. Section 4 presents our *guided single-stepping* approach, including a proof-of-concept implementation using *KVMi*. In Section 5, we evaluate the correctness, stealthiness, and performance of our solution. Finally, Section 6 concludes the paper.

2 Background

In this section, we present relevant background on hardware-assisted virtualization for *AMD* and *Intel x86* systems as well as on virtual machine introspection.

[3] `https://www.smartvmi.org/`

2.1 Hardware-assisted Virtualization

The predominant approach to system virtualization on *x86* is *hardware-assisted virtualization*, which was introduced by Intel in 2005 through *VT-x / VMX* (Virtual Machine Extensions). This extension featured new processor modes for virtualization: *VMX root mode*, used as privileged mode by the hypervisor, and *VMX non-root mode*, where the guest system executes in a non-privileged mode. Our particular interest are context switches from the unprivileged to the privileged mode, i.e., from the guest virtual machine to the hypervisor, through traps, refered to as *VM exit* (the opposite direction, from the hypervisor to the virtual machine, is called *VM entry*). The configuration of the virtual machines running under *hardware-assisted virtualization*, including *VM exit* conditions, is performed in the *Virtual Machine Control Structure* (VMCS) [16].

AMD processors also support *hardware-assisted virtualization* since the introduction of the *Secure Virtual Machine* (SVM) extension[4] in 2006. In this extension, the new processor modes are called *host mode* (privileged) and *guest mode* (unprivileged). The hypervisor configures the hosted virtual machines through the *Virtual Machine Control Block* (VMBC) [17].

Starting with the second generation of processor extensions for *hardware-assisted virtualization*, the vendors implemented a concept known as *Second Level Address Translation* (SLAT). While traditional paging solely translates logical, virtual addresses to physical addresses, SLAT extends capabilities of the *Memory Management Unit* (MMU) by another dimension: The translation of the physical address within the virtual machine to the physical address on the host machine. For *Intel* processors, the SLAT implementation is called *Extended Page Tables* (EPT). *Intel* refers to the top-level paging structure in the guest that translates from *guest virtual addresses* (GVA) to *guest physical addresses* (GPA) as *Page Map Level 4* (PML4). For the new dimension, the corresponding paging structure is called EPT PML4 and translates from GPA to *host physical addresses* (HPA) [5]. The `CR3` register references the PML4, while the VMCS stores the EPT PML4 in the EPT Pointer (EPTP) field.

AMD x86 processors implement SLAT with *Rapid Virtualization Indexing* (RVI) or *Nested Paging* (NP). In this implementation, the *guest page tables* (gPT) translate *guest linear addresses* (GLA) to *guest physical addresses* (GPA). For the second level address translation, *nested page tables* (nPT) are used to convert GPA to *system physical addresses* (SPA) [18]. The `CR3` register in the guest is referred to as *guest CR3* (gCR3) and holds the reference to the gPT. The hypervisor loads the nPT value into the *nested CR3* (nCR3) field in the VMBC [1]. Besides completely different terminology, there are also significant implementation differences between AMD's RVI and Intel's EPT.

[4] Newer publications refer to the same extension as *AMD Virtualization* (AMD-V) [18].

2.2 Virtual Machine Introspection

Virtual machine introspection is the "approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it." Garfinkel and Rosenblum characterize VMI by three main properties [4]: Isolation (between monitoring and monitored system), introspection (monitoring software has a full, untampered view of the whole system), and interposition (interception of operations in the virtual machine).

VMI-based monitoring mechanisms can be categorized as either *passive (or polling)*, which means they analyze the main memory of the virtual machine based on external triggers, or as *active (or event-triggered)*, which means they interposition themselves with the control flow of the virtual machine, e.g., by placing breakpoints [6]. The active interpositioning allows introspection applications to perform their introspection task at specific, predetermined locations in the control flow of applications running inside the guest virtual machine.

In our work, we investigate two forms of *active* VMI: The first type of introspection mechanism is memory access tracing based on the SLAT feature of modern processors. For such mechanisms, first, the VMI application modifies the memory access permissions of a page within the SLAT. Second, accessing these pages triggers a trap to the hypervisor, which then emits an event to the introspection application. Besides these two basic steps, there are also mechanisms that involve additional actions, such as creating new views (top-level paging structures for SLAT) and dynamically switching between these views [9].

The second kind of introspection mechanism involves the use of hyper-single-stepping functionality. Unlike regular single-stepping, which transfers control to the guest kernel after each instruction and can be used, e.g., by guest-level debuggers, hyper-single-stepping executes a single instruction in the guest and then traps to the hypervisor, which then can notify the VMI tool. When combined with a software hyper-breakpoint [14], this mechanism is particularly valuable. This combination involves replacing an instruction in the guest with a breakpoint instruction. Upon reaching this instruction, the guest traps to the hypervisor. The VMI tool handles this breakpoint by restoring the original instruction and activating single-stepping. After the guest executes the original instruction, the single-step triggers another hypervisor trap and the VMI tool re-inserts the breakpoint.

3 State of the Art

While VMI is a promising technique for practically any efficiently virtualizable architecture, current industry and academia work focuses mainly on the *x86* architectures. Yet as we have alluded to in earlier parts of this work, there are several architectural differences between the two main *x86* vendors, namely *Intel* and *AMD*, that limit the applicability of previous active introspection research on *AMD* processors. These limitations exist because most works in the literature conducted their development on *Intel* processors with the *Intel VT-x* processor

extensions. Subsequently, we will describe the two main architectural differences and summarize the state of the art regarding addressing the open problems arising from them.

3.1 SLAT-based Mechanisms

Zhang and Zonouz have shown that the combination of SLAT controls and events is suitable for hiding injected code from the guest [20]. By maintaining a set of complementary paging structures for read/write/execute operations and switching between them dynamically, it is possible to have a different mapping for reads to a page compared to an instruction fetch. Hence, injected code practically becomes invisible to the guest. While their approach used a hypervisor to perform the code hiding, we do not consider their approach as virtual machine introspection due to the non-flexible design. Instead, they created the technique specifically for rootkits.

The first hypervisor to offer support for a wide range of options to manipulate the SLAT for introspection purposes was Xen with the altp2m mechanism [9]. It allows the introspection application to manage multiple *guest-**p**hysical* **to** ***m**achine-physical* mappings for a single virtual machine, manipulate these mappings, switch between them, and directly handle the related events. However, as of now, altp2m is only available on *Intel* processors.

Tanda was the first to identify the realization of SLAT memory access permissions on *AMD* processors as an issue regarding implementing the usual code hiding technique [13]. Whereas *Intel EPT* allows configuring read, write, and execute permissions of access to a specific page, *AMD RVI* merges the read and execute permissions. Hence, it is not possible to set them separately, which adversely affects the earlier-mentioned code-hiding technique. Tanda has also outlined two partial workarounds for this problem. However, when used with virtual machine introspection, these partial solutions rely on the availability of hyper-single-stepping, which, as we will see in the following, is also missing. Furthermore, neither of the approaches he proposed reaches performance characteristics close to utilizing the hardware implementation on *Intel*.

Therefore, we can conclude that SLAT-based introspection mechanisms on *AMD* processors are equally powerful to those on *Intel* if and only if hyper-single-stepping is available. Henceforth, this paper will focus on realizing this requirement and thus provide adequate support for active introspection mechanisms on *AMD64*.

3.2 Hyper-Single-Stepping

The prevalent way to realize hyper-single-stepping for VMI architectures in environments based on hardware-assisted virtualization is through virtualized processor capabilities. For example, *Intel* processors feature the *Monitor Trap Flag*, which can be set in the VMCS by the hypervisor. When this flag is enabled, the processor will trigger a *VM exit* after each execution of an instruction.

The two most popular open-source hypervisors with VMI support for hyper-single-stepping – *Xen*[5] [2] and *KVMi* [8] – employ this functionality. However, as mentioned earlier, this requires support from the architecture and, ultimately, the processor vendor. Currently, an equivalent of the *Monitor Trap Flag* does not exist on AMD. Hence, the feature is unavailable on *AMD* processors.

Yet, some debuggers such as GDB [7,19] offer limited single-stepping support for virtual machines even on *AMD*. They accomplish this by using the non-virtualized single-stepping feature of the processor. Therefore, they incur severe drawbacks: the single-stepping is trivially detectable from within the guest, malicious actors can easily disable the mechanism, and the guest cannot do any single-stepping on its own. All of these reasons make this approach unsuitable for VMI, where solutions are bound to be isolated from the guest and stealthy. The challenges to achieving these properties in an adverse environment are the topic of this paper.

Finally, Sato et al. discuss ensuring stealthiness and correctness of retrofitted virtual machine introspection mechanisms [12]. Their work mainly focuses on retrofitted hardware breakpoints. However, we believe we can draw relevant lessons for a much broader range of mechanisms, including hyper-single-stepping, see Section 4.2.

4 Introspection on the AMD64 Architecture

As the realisation of the solutions presented by Tanda [13] rests on the availability of hyper-single-stepping, and one can implement the outlined approaches at the level of the introspection application with existing APIs, we focus solely on the hyper-single-stepping functionality for the remainder of this paper. In our work, the trust model assumes the hypervisor and the host system, which runs the VMI application, to be trusted. Furthermore, we consider the hardware and its firmware to be untampered. All other entities are untrusted. Finally, we assume that the attacker does not directly attack the hypervisor or other trusted entities, for example, through VM escapes.

4.1 Design

A naive approach to address the lack of the monitor trap flag on *AMD* machines is to emulate instead of virtualizing the instructions in question. The approach of falling back to emulation when hardware-assisted virtualization is unsuitable is a common technique for similar problems. Regrettably, it comes with two significant shortcomings: First, the emulation is often prone to errors due to the high complexity and heterogeneity of processors. Second, it may sacrifice the performance gains of efficient virtualization and can therefore slow down the execution of the virtual machine significantly.

[5] `https://xenbits.xen.org/gitweb/?p=xen.git;a=blob;f=xen/arch/x86/hvm/vmx/intr.c;h=80bfbb478782446cb17b53004435e41206f993b8;hb=556c2e817c9cf23b675eb4eaa2dc091f7bb3039f#l1250`

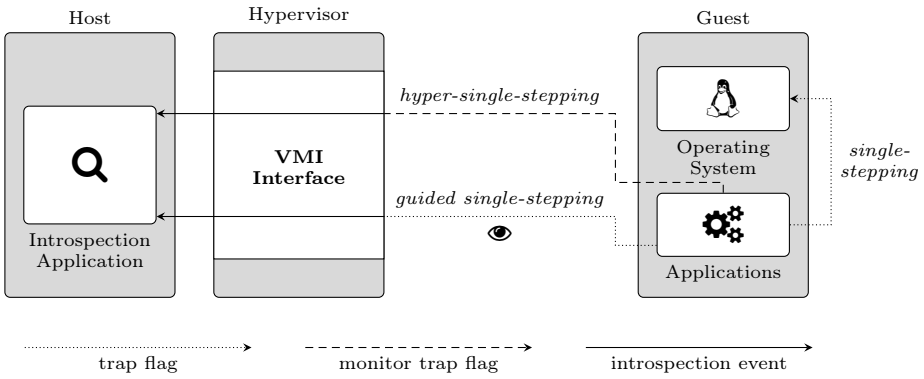

Fig. 1. Concept of *guided single-stepping* contextualized with other approaches

Another potential solution is to disassemble the current instruction and insert a hyper-breakpoint directly following this instruction on-the-fly [11]. This approach reduces the susceptibility to errors compared to the emulation since the only requirement to execute the strategy is to determine the correct length of any instruction. However, it still can be considered significantly slower than virtualization with the monitor trap flag because every single-step now requires mapping, reading, and length-disassembling the next instruction through VMI.

Instead of these slow and error-prone approaches, we rely on the regular trap flag in the guest to generate the trap. Then, we elevate this trap to the hypervisor by configuring the processor to perform exception intercepting on the respective exception vector [1]. As we will see later, this retrofitting of single-stepping for introspection applications comes at practically no cost and is resistant to both errors and malicious insiders.

Figure 1 shows our approach called *guided single-stepping* alongside single-stepping and hyper-single-stepping. We refer to it as *guided* since we guide the guest's execution from the outside to avoid manipulation and detection of the monitoring. Our approach offers a way to realise the benefits of virtual machine introspection, namely isolation, and interposition, for a single-stepping mechanism targeting virtual machines. Hence, it provides the same guarantees as hyper-single-stepping, even when not explicitly supported by the hardware.

However, an important design decision remains. Like many other introspection mechanisms, our approach can be implemented at different levels, i.e., in the hypervisor, an introspection library, or the guest itself. We decided to place our solution in the hypervisor and the introspection library *LibVMI*.

Operating at multiple levels in the introspection stack enables a particularly small footprint: Since our implementation is implemented transparently at the hypervisor level, existing introspection applications, in general, do not need to be modified for operation on *AMD* systems when using *guided single-stepping*. As single-stepping in a VMI context is often used only selectively, e.g., to step

over a single instruction, counterintuitively, an implementation at the level of the introspection library reduces the overhead. Hence, the introspection application can determine partial emulation functions ahead of time and potentially even cache them.

4.2 Implementation

As mentioned in the beginning, our proof-of-concept implementation[6] builds upon the open-source introspection API *KVMi* based on the *KVM* hypervisor. Since it operates at multiple layers in the introspection stack, we will discuss the challenges and concrete realisation separately for each affected layer.

Hypervisor Layer The core responsibility of the part of our solution that resides in the hypervisor is to configure the virtual machine such that regular hardware single-stepping is enabled and the generated traps end up in the hypervisor. In particular, this requires the following three steps that must occur transactionally and are started by the `KVMI_VCPU_CONTROL_SINGLESTEP` command:

1. Exception intercepting must be enabled for the #**DB** (Debug Exception) vector. We implement this using the `set_exception_intercept` helper function[7] during `update_bp_intercept`.

2. We must save off the original `RFLAGS` of the vCPU to determine whether to reinject the exception and to set the trap flag correctly upon exit. Our solution stores the guest's `RFLAGS` within the `vcpu_svm` struct using the `svm_get_rflags` function.

3. Finally, we must force the trap flag on within the guest for the duration of the *guided single-stepping*. Our solution performs this manipulation within `__kvm_set_rflags`[8].

To disable the *guided single-stepping* after reaching the target instruction, we have to perform the inverse of these three steps in reverse order. In this case, we can omit step 2. Enabling and disabling the *guided single-stepping* is only possible when the vCPU is currently not running, i.e., it is halted by either an active mechanism or paused.

After setting up the virtual machine in this way, single-step operations in the guest trap to the hypervisor. In *KVM*, they eventually reach the exit handler called `db_interception`[9]. In this handler, we can convert the exception to a VMI event and send it to the introspection application for further processing.

[6] Available at: `https://github.com/smartvmi/VMI-on-AMD`

[7] `https://elixir.bootlin.com/linux/v5.4.217/source/arch/x86/kvm/svm.c#L591`

[8] `https://elixir.bootlin.com/linux/v5.4.217/source/arch/x86/kvm/x86.c#L10104`

[9] `https://elixir.bootlin.com/linux/v5.4.217/source/arch/x86/kvm/svm.c#L2783`

Since our retrofitted approach utilizes the regular trap flag, we need to consider the edge case that the virtual machine itself is already single-stepping, e.g., for debugging purposes. To account for this issue, we need to reinject the interrupt into the virtual machine if the trap flag was already set by the guest. Hence, we have saved the guest's original value of `RFLAGS` in step 2. Reinjecting the debug exception is as simple as calling `kvm_queue_exception` with `DB_VECTOR` after delivering the event to the introspection application. Thereby, the operating system in the guest can correctly handle the single-step on its own.

Our implementation currently does not handle a change to the trap flag in the guest during single-stepping. This situation can, for example, occur when a self-debugging program in the guest sets the trap flag on itself to obfuscate its control flow. Addressing this problem would require sophisticated emulation at the level of the hypervisor. However, we consider this a niche technique that remains outside the scope of our current work.

Finally, the single-step may clobber the debug-status register (`DR6`). As the processor sets bit 14 of `DR6` when a **#DB** exception occurs due to single-stepping, the guest could detect this value and conclude that it is being monitored. To avoid this situation, we clear this bit if and only if we have not reinjected the single-step into the guest.

Application Layer At the level of the introspection library or the application layer, we will exclusively deal with implementation details that are cheaper to implement at this level than in the hypervisor. In particular, we apply partial emulation on top of some critical instructions. As the application, in many cases, e.g., when using hyper-single-stepping together with hyper-breakpoints, knows the instruction ahead of time, the resulting overhead can be limited. Not only can we avoid mapping and reading the page of the currently executed instruction, but we can also eliminate all superfluous emulation for the statistically dominant instructions that do not require intervention.

As our retrofitted approach uses the guest's trap flag to generate the interrupts, malicious insiders could potentially interfere with the monitoring by manipulating the `RFLAGS` in the guest. To address this, we guide the execution in the guest. The instruction we have to worry about the most is the `POPF` instruction that loads new flags from the stack. We manipulate the execution of this instruction by dynamically rewriting the stack contents upon execution. Before the guest executes this instruction, we force the trap flag onto the top-most value on the stack:

```
1  ACCESS_CONTEXT(ctx,
2          .translate_mechanism = VMI_TM_PROCESS_DTB,
3          .addr = event->x86_regs->rsp + 8,
4          .pt = event->x86_regs->cr3 & ~0x1000ull);
5
6  uint64 eflags, eflags_new;
7  if (VMI_SUCCESS == vmi_read_64(vmi, &ctx, &eflags))
8  {
9          eflags_new = eflags | X86_EFLAGS_TF;
```

```
10          vmi_write_64(vmi, &ctx, &eflags_new);
11  }
```

As the value remains on the stack after the execution, we have to write back the original value after the instruction executes to hide the presence of the monitoring. To this end, we must account for the fact that this value lies beyond the stack pointer after the execution of the instruction.

The counterpart to the `POPF` instruction is the `PUSHF` instruction. This instruction places the flags on top of the stack. As it is also available in user mode, it is an ideal candidate to detect our monitoring. Again, we can avoid detection by rewriting the stack after its execution:

```
1   ACCESS_CONTEXT(ctx,
2           .translate_mechanism = VMI_TM_PROCESS_DTB,
3           .addr = event->x86_regs->rsp + 8,
4           .pt = event->x86_regs->cr3 & ~0x1000ull);
5
6   uint64 eflags;
7   if (VMI_SUCCESS == vmi_read_64(vmi, &ctx, &eflags))
8   {
9           eflags &= ~X86_EFLAGS_TF;
10          vmi_write_64(vmi, &ctx, &eflags);
11  }
```

Finally, we have to deal with the *CLI* instruction that clears the interrupt flag. With interrupts disabled, the single-stepping mechanisms will no longer work. Luckily, most modern kernels nowadays are fully or mostly preemptible. Therefore, we should not encounter this instruction for the most part when using *guided single-stepping*. Assuming the execution reaches this instruction, we propose two different approaches based on where it is located: In case it occurs in a trusted location, i.e., one of the few places in the kernel that are not preemptible, we ignore it. If we encounter the instruction in an untrusted position, i.e., a driver or code not belonging to the kernel image, we propose to halt the virtual machine for manual inspection.

5 Evaluation

In the following, we assess our *guided single-stepping* proof-of-concept implementation regarding its correctness, stealthiness, and performance.

5.1 Correctness

Our work relies on the correctness of the trap flag in the guest and the interception of the #**DB** exception. Since all existing debuggers for the *x86* architecture use the trap flag, we can assume the feature to be working correctly.

However, there are critical differences to the hyper-single-stepping facilitated by the monitor trap flag. In particular, the behavior between the two can differ when delivering interrupts. First, our solution does not yet consider that the trap flag can be reset from an *Interrupt Service Routine* (ISR), e.g., through the `IRET` instruction. We could address this weakness by applying the flag to the stack when encountering such an instruction, much like for the `PUSHF` instruction. Since most introspection applications use single-stepping selectively, e.g., for stepping over a single instruction, we currently do not regard this as a problem.

Second, it is theoretically possible to use hardware multitasking to turn off the trap flag from within the guest. The necessary procedure requires stripping the flag from the `EFLAGS` field in the active *Task State Segment* (TSS). We must note that this is not possible from long mode since hardware multitasking is not available in this mode and the TSS only holds the stack pointers and the *Interrupt Stack Table* (IST). Therefore, exploiting this weakness would require the attacker to switch the processor back to protected mode. For this reason, we consider this possibility very unlikely. However, we could solve this issue by trapping writes to the `EFER` (Extended Feature Enable Register) MSR (Model-specific Register). By checking against a write of 0 to bit 8 (Long Mode Enable), we can detect this behavior and halt the machine for manual inspection.

What we should note for both cases, however, is that this only disables single-stepping until the next VM exit since our implementation in the hypervisor makes sure to reapply the flag before entering the guest.

Finally, all instructions covered in our proof-of-concept implementation are single-byte instructions (`POPF`, `PUSHF`, and `CLI`). Therefore, we generally do not require sophisticated disassemblers to identify them accurately from the introspection application. However, a malicious guest could append prefixes such as the REX prefix to the instruction that does not have any effect, and thus no sensible assembler would generate [1]. Hence, we caution against simply checking the opcode for security-critical applications.

5.2 Stealthiness

As is the case with many VMI-based systems, our solution is not entirely invisible to the guest. Active introspection can be detected from within the guest in numerous ways. The most primitive approach that is often present in malware is the detection of the hypervisor. This detection is possible due to the enforced isolation, which requires privileged instructions to be emulated [10]. By comparing the execution time of these instructions with their native unvirtualized counterparts, it is easy to determine if the system is currently executing under virtualization.

However, there are also much more sophisticated methods that not only can determine the presence of the hypervisor but also ongoing active introspection [15]. These can include timing attacks on various exit conditions, such as our interception of the **#DB** exception. Therefore, our *guided single-stepping approach* is detectable from within the guest. However, we should consider that the same is true for hyper-single-stepping.

While discussing our implementation, we addressed other detection methods, such as reading out the trap flag. To verify the effectiveness of these measures, we attempted to detect the presence of our monitoring with the application shown below. As expected, our solution can successfully hide the presence of the trap flag from the guest.

```
1 uint64_t eflags = __builtin_ia32_readeflags_u64();
2 fprintf(stdout, "X86_EFLAGS_TF: %lu\n", !!(eflags &
    ↪ X86_EFLAGS_TF));
```

5.3 Performance

To evaluate the performance of our solution, we use an *ASUS PN51* fitted with an *AMD Ryzen 5 5500U* Hexa-core CPU, 32 *GiB* of *DDR4-2666* main memory, and 1 *TiB* of non-volatile memory on an *Kingston A2000* NVMe SSD. We use Debian 11 for both the host and guest operating systems in our evaluation. The virtual machine used in the following experiments has 2 *GiB* of main memory assigned to it. We take comparative measurements for *Intel* processors on a machine with similar characteristics. This machine is an *ASUS PN62* equipped with a *Intel Core i7-10510U* quad-core CPU, 32 *GiB* of *DDR4-3200* main memory, and 1 *TiB* of non-volatile memory on an *Kingston A2000* NVMe SSD.

Microbenchmarks. To assess the performance of our retrofitted software solution, we measure breakpoint and single-stepping performance by placing a breakpoint on the `getpid` system call. Upon execution of this breakpoint, we replace it with the original instruction, single step over it, and restore the breakpoint. We chose this way of evaluating the performance of our solution because it resembles how VMI is usually used in real-world applications. We include the source code of this benchmark in the repository. Our setup measures 1,000 system call invocations from user mode. We present the results of this measurement with a sample size of 10 in Figure 2.

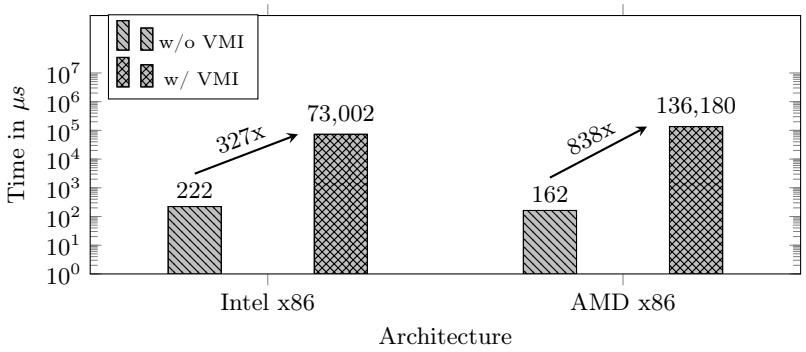

Fig. 2. Execution time with breakpoint on `__x64_sys_getpid` (less is better)

As expected, the native solution implemented in hardware and microcode on *Intel* outperforms our retrofitted software solution for *AMD*. However, both in terms of overhead and absolute execution time, the approaches are within one order of magnitude of each other. The overhead of this mechanism is around 2.6 times higher on *AMD* than on *Intel*. In absolute terms, the difference amounts to a factor of 1.9. The error of our measurement was below 10% in both cases.

A limitation of this measurement is the jointness of the breakpoint and single-stepping mechanism. Hence, it is not evident if the difference we observe can be attributed entirely to the single-stepping or if the breakpoint implementation exhibits some overhead caused by architectural differences. However, we still argue for this way of determining the performance as it is closest to real-world applications of the proposed mechanism.

UnixBench. The initial microbenchmark focused on assessing the worst-case scenario for VMI. It involved a loop executing a lightweight system call, `getpid`, which merely returns the process's PID and has minimal execution time. VMI introduced a trap to the monitoring application in each iteration. To provide a more realistic understanding of VMI's impact on the performance of a target VM, we conducted additional tests using selected system calls traced during the execution of various *UnixBench* [3].

Table 1 lists the results obtained from these tests. The performance figures indicate the number of iterations completed within a fixed time interval. The *spawn* test repeatedly invoked the `clone` system call, and the *execl* test invoked the `execl` function, which translates to the `execve` system call. Despite tracing each system call invocation in these tests, due to the higher execution time of the system calls itself, the relative overhead imposed by VMI was significantly lower. The *syscall* (system call overhead) test resembled our microbenchmark and produced similar results. The *pipe* throughput test focuses on communication via a pipe, and we traced all `write` system calls during the test, with significant overhead. Lastly, the pipe-based context switching test *context1* is "more like a real-world application" [3] and measures switches between two processes engaged in a bidirectional pipe conversation. When tracing the `pipe` system call, which is called only once at the beginning, no noticeable overhead was observed. We omitted the *DhryStone* and *WhetStone*, as they do not use system calls, and thus their runtime performance is not influenced by VMI-based system call tracing.

Table 1. Consolidated *UnixBench* scores on *AMD64* (higher is better)

Test	**Monitored Syscall**	**w/o VMI**	**w/ VMI**	**Unit**
spawn	`sys_clone`	18,050 (±475)	3,548 (±817)	processes/s
execl	`sys_execve`	5,168 (±101)	1,880 (±553)	calls/s
syscall	`sys_getpid`	17,758,288 (±339,741)	7,435 (±911)	calls/s
pipe	`sys_write`	2,639,602 (±49,713)	7,472 (±1,057)	calls/s
context1	`sys_pipe`	259,947 (±1,272)	261,195 (±3,612)	calls/s

6 Summary

In this paper, we have identified the causes of the limited availability of active introspection mechanisms on *AMD x86* processors impacting many introspection applications and remedied some of the more pressing concerns. Thus, we have improved the state of the art of VMI-based approaches and enabled their use on previously inaccessible systems.

First, we have highlighted two architectural differences that affect the implementation of introspection tools and the applicability of previous research. Active mechanisms such as hyper-breakpoints often use SLAT-based controls and events to realize code hiding. However, the specific implementation of SLAT with *AMD RVI* does not allow to set read and execute permissions independently. Hence, we cannot use the usual code-hiding technique. Previous research has proposed alternative approaches, which rely on the availability of hyper-single-stepping. Yet, due to the missing support of the monitor trap flag on *AMD* processors, these approaches have not been realized for introspection-based solutions.

Second, we have focused on remedying this lack of hyper-single-stepping with our *guided single-stepping* approach. Instead of relying on hardware support through the *Monitor Trap Flag*, we retrofit the capabilities using software and interception intercepting. We guide the execution of the guest from the hypervisor and the introspection application to ensure the correctness and stealthiness of the monitoring.

Third, we have implemented the approach of *guided single-stepping* in the *KVM* hypervisor and the *LibVMI* introspection library. The evaluation of this novel software-based approach demonstrates that its performance is in the same order of magnitude as comparable hardware implementations on *Intel* processors. Hence, we claim that our solution increases the portability of introspection applications for *AMD* processors.

Finally, we release our proof-of-concept implementation as free software and work towards integrating it into the relevant open-source projects.

Acknowledgement

This work has been funded by the Bundesministerium für Bildung und Forschung (BMBF, German Federal Ministry of Education and Research) – project 01IS21063A-C (SmartVMI).

References

1. Advanced Micro Devices: AMD64 Architecture Programmer's Manual (2019). Volume 2
2. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP, p. 164–177. Association for Computing Machinery, Bolton Landing, NY, USA (2003). https://doi.org/10.1145/945445.945462

3. Byte Magazine: byte-unixbench (1983). URL https://github.com/kdlucas/byte-unixbench. Accessed: 2023-04-20
4. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: NDSS, vol. 3, pp. 191–206 (2003)
5. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual (2009). Volume 2A
6. Jain, B., Baig, M.B., Zhang, D., Porter, D.E., Sion, R.: SoK: Introspections on Trust and the Semantic Gap. In: IEEE Symposium on Security and Privacy, pp. 605–620 (2014). https://doi.org/10.1109/SP.2014.45
7. Kiszka, J.: Debugging kernel and modules via gdb. https://www.kernel.org/doc/Documentation/dev-tools/gdb-kernel-debugging.rst (2023). Accessed: 2023-03-31
8. Lazăr, A.: KVMi subsystem v7 for KVM. KVM mailing list https://lore.kernel.org/kvm/20200207181636.1065-1-alazar@bitdefender.com/ (2021). Accessed: 2023-03-24
9. Lengyel, T.K.: Stealthy monitoring with Xen altp2m. https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/ (2016). Accessed: 2023-03-24
10. Pék, G., Buttyán, L., Bencsáth, B.: A survey of security issues in hardware virtualization. ACM Computing Surveys **45**(3) (2013)
11. Proskurin, S., Lengyel, T., Momeu, M., Eckert, C., Zarras, A.: Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC, p. 407–417. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3274694.3274698
12. Sato, M., Nakamura, R., Yamauchi, T., Taniguchi, H.: Improving transparency of hardware breakpoints with virtual machine introspection. In: 12th International Congress on Advanced Applied Informatics (IIAI-AAI), pp. 113–117 (2022). https://doi.org/10.1109/IIAIAAI55812.2022.00031
13. Tanda, S.: 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 (2019). Accessed: 2023-03-24
14. Taubmann, B.: Improving digital forensics and incident analysis in production environments by using virtual machine introspection. Ph.D. thesis, Faculty of Computer Science and Mathematics, University of Passau (2019)
15. Tuzel, T., Bridgman, M., Zepf, J., Lengyel, T.K., Temkin, K.J.: Who watches the watcher? Detecting hypervisor introspection from unprivileged guests. Digital Investigation **26**, S98–S106 (2018)
16. Uhlig, R., Neiger, G., Rodgers, D., Santoni, A.L., Martins, F.C., Anderson, A.V., Bennett, S.M., Kagi, A., Leung, F.H., Smith, L.: Intel virtualization technology. Computer **38**(5), 48–56 (2005)
17. Van Doorn, L.: Hardware virtualization trends. In: ACM/Usenix International Conference On Virtual Execution Environments, vol. 14, pp. 45–45 (2006)
18. VMWare Inc.: Performance Evaluation of AMD RVI Hardware Assist. https://www.cse.iitd.ernet.in/~sbansal/csl862-virt/2010/readings/RVI_performance.pdf (2008). Accessed: 2023-03-24
19. Wessel, J.: Using kgdb, kdb and the kernel debugger internals. https://www.kernel.org/doc/Documentation/dev-tools/kgdb.rst (2022). Accessed: 2023-03-31
20. Zhang, M., Zonouz, S.: How to hide a hook: A hypervisor for Rootkits. Phrack Magazine **15**(69) (2016)