Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection

Sergej Proskurin
Technical University of Munich
proskurin@sec.in.tum.de

Tamas Lengyel
The Honeynet Project
tamas@tklengyel.com

Marius Momeu
Technical University of Munich
momeu@sec.in.tum.de

Claudia Eckert
Technical University of Munich
eckert@sec.in.tum.de

Apostolis Zarras
Maastricht University
apostolis.zarras@maastrichtuniversity.nl

ABSTRACT

ARM has become the leading processor architecture for mobile and IoT devices, while it has recently started claiming a bigger slice of the server market pie as well. As such, it will not be long before malware more regularly target the ARM architecture. Therefore, the *stealthy* operation of *Virtual Machine Introspection* (VMI) is an obligation to successfully analyze and proactively mitigate this growing threat. Stealthy VMI has proven itself perfectly suitable for malware analysis on Intel's architecture, yet, it often lacks the foundation required to be equally effective on ARM.

In this paper, we closely examine both ARMv7 and ARMv8 architectures to identify shortcomings and develop novel techniques necessary for effective virtualization-based dynamic malware analysis. We implement and open-source a prototype, named *altp2m*, for the open source Xen Project hypervisor on ARM. Compared to traditional VMI approaches, our solution enables hypervisors to dynamically allocate and switch among multiple guest memory views by utilizing the *Second Level Address Translation* (SLAT). Further, we implement an alternative single-stepping mechanism and leverage the *execute-only* capability of the SLAT mechanism on ARMv8 to enable stealthy in-guest instrumentation. To target also ARMv7-based systems, we manipulate the TLB organization through altp2m to coordinate the guest kernel execution flow. To demonstrate the effectiveness of our system, we combine all building blocks of our work to form the foundation for the dynamic malware analysis system DRAKVUF on ARM. Overall, our experiments reveal that our novel dynamic analysis system is stealthy, efficient, and is perfectly suited to assist malware analysts to quickly comprehend the behavior and reduce the mitigation time of malware targeting ARM.

ACM Reference Format:
Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. 2018. Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3274694.3274698

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

1 INTRODUCTION

State-of-the-art malware utilizes techniques that facilitate its execution with same privileges as the sensitive and security relevant parts of the *Operating System* (OS). This allows malware to operate under the radar and evade detection. To address this ever prevailing threat, security applications progressively take advantage of virtualization technology. More precisely, Chen et al. [5] suggest to isolate applications from the host OS to enhance security and mobility; virtualization technology encapsulates, and thus protects, security mechanisms from the OS by means of a hypervisor. This method has proven effective, which led to an increased adoption of virtualization for malware detection and analysis frameworks in both commercial [3, 10, 32] and open source applications [17, 18, 24, 33].

Virtualization has helped defenders gain the upper hand in the arms race against malicious actors. As hypervisors expose a narrow, virtual hardware interface toward guests, they bare a limited attack vector while maintaining a complete and untainted view of the guest's state. To benefit from this constellation, *Virtual Machine Introspection* (VMI) must be applied [12]. VMI constitutes techniques that allow hypervisors through hardware virtualization extensions to observe, analyze, and control the state of guest *Virtual Machine*s (VMs). In particular, the stealthy nature of VMI is of high relevance as it can assist the analysis of split-personality malware that behaves differently if it believes it is being analyzed [6].

Previously, x86 was the dominant player in the server world and an exceedingly attractive target for exploitation; [1] this is where VMI was most needed. The same applies to malware. Yet, IoT, mobile devices, and the growing demand for ARM in the server market created renewed emphasis on developing VMI systems for ARM.

Virtualization-based analysis frameworks often resort to techniques allowing to *intercept* and *single-step* guest OSes [7, 13, 17]. To intercept the guest's execution, one can apply both *invasive* [7, 13, 17] and *non-invasive* [8, 20, 24] approaches. To assist malware analysis, both approaches must be invisible to the guest. Although non-invasive approaches are inherently stealthy, in-guest memory or register artifacts used by invasive approaches must be explicitly hidden. For instance, an adversary can use the finite number of hardware breakpoint registers to reveal the analysis framework. Also, while Intel as well as the AArch64 execution state of ARMv8 CPUs allow to hide memory-artifacts by marking memory pages as *execute-only*, second level translation tables of both the AArch32

[1] We refer to both x86 and x86-64 as the x86 architecture.

execution state of ARMv8 and the ARMv7 architecture prohibit *execute-only* memory and thus impede stealthy VMI.

To transparently single-step guest OSes on Intel CPUs, the *Monitor Trap Flag* (MTF) can be used. As a matter of fact, MTF is part of Intel's virtualization extensions and inaccessible to the guest. Sadly, this feature is not supported by ARM. Literally, it is unfeasible to single-step the guest in a stealthy way by relying solely on the hardware capabilities. While previous efforts employ VMI on ARM [13, 29, 37], none of them achieves a stealthy solution against attackers with root privileges. Emulation presents a potential workaround, but is known for being imperfect [9, 35, 36].

In this paper, we explore new directions of VMI primitives, which empower stealthy monitoring of guest OSes with multiple *virtual CPUs* (vCPUs) on both AArch32 and AArch64 without resorting to emulation. First, we introduce an alternative method on placing breakpoints; instead of using hardware or software breakpoints that either leak information about the analysis framework or require logic that distinguishes between breakpoints set by the analysis system and the guest—thus increasing the performance overhead—we expand the idea that was first presented in SPROBES [13]. As such, we place *Secure Monitor Call* (SMC) instructions into the guest kernel to intercept the VM and redirect the control to the hypervisor. By injecting only two SMC instructions, we enable single-stepping without using the hardware-intended approach, which can reveal the analysis system. In parallel, we implement a system, which facilitates an external monitor using second level translation tables to define and dynamically switch among different guest physical memory views. In this context, we introduce our extension to the Xen Project hypervisor [19] on ARM, called *alternate p2m* (altp2m).

These methodologies suffice for stealthy analysis on AArch64. Yet, to hide from malware on AArch32, which lacks *execute-only* memory, we consolidate the aforementioned techniques along with the *Translation Lookaside Buffer* (TLB). In detail, we take control of the TLB organization by leveraging altp2m to establish a stealthy guest monitoring approach. This approach employs altp2m to carefully de-synchronize the TLB organization to effectively hide code pages in guest memory. This allows us to maintain different mappings of the same guest physical frame in the *instruction* and *data* TLB and thus to effectively hide code pages in guest memory [31].

We extend the well-known dynamic malware analysis framework DRAKVUF [17] with the capabilities of the above primitives to empower ARM for stealthy VMI. In fact, we leverage DRAKVUF to evaluate the performance and effectiveness of our VMI primitives that are tailored for AArch32 and AArch64. We believe our work constitutes an important building block, able to introduce stealthy VMI to the ARM architecture while remaining efficient and robust.

In summary, we make the following main contributions:

- We empower *stealthy* multi-vCPU guest kernel monitoring on ARM by extending the Xen hypervisor to dynamically switch among different guest physical memory views.
- We utilize Xen and altp2m to de-synchronize the TLB organization on AArch32 to tackle the architectural deficit of *execute-only* memory in the second level translation tables.
- We develop and open source the foundation for the binary analysis framework DRAKVUF and establish stealthy dynamic malware analysis on AArch32 and AArch64.

2 BACKGROUND

Split-personality malware frequently employs anti-virtualization techniques [2, 6] to reveal and evade introspection. This can be addressed by trying to make the VM indistinguishable from real hardware—a system that achieves perfect VM *transparency* is still infeasible in practice [11]. Recently, however, we observe an increasing trend toward system consolidation through virtualization which renders the goal of VM transparency obsolete. A virtualized system does not necessarily indicate that its sole purpose is malware analysis. Therefore, it makes no sense for attackers to exclude virtual environments. Nevertheless, malware can still detect VM-based analysis systems, through artifacts, ranging from guest-accessible memory to register contents. Consequently, cloaking remains an open question and emphasizes the need for stealthy monitoring.

Contrary to x86, ARM does not foresee hardware capabilities that are essential for *stealthy* malware analysis. In fact, this is one of the reasons why existing VMI approaches for x86 cannot be applied to ARM. In particular, ARM is not capable of hiding artifacts involved in single-stepping guest VMs. Additionally, ARMv7 complicates hiding in-guest code instrumentation, as it lacks the capability of granting *execute-only* permissions to code pages. While both points can be addressed through emulation techniques, we choose to avoid emulation, as it is known for being imperfect [9, 35, 36]. In this section, we introduce the ARM architecture and depict its limitations in regard to stealthy VMI. We particularly amplify upon the ARM virtualization extensions discussing chosen architectural components that are relevant for this paper.

2.1 ARM Exception Levels

ARM distributes software execution across different *processor modes*, each with distinct *privilege levels* on ARMv7, also referred to as *exception levels* on ARMv8. Disregarding the Thumb/T32 instruction set (i.e., a 16-bit and 32-bit encoded subset of the ARM *Instruction Set Architecture* (ISA)), while ARMv7 supports only 32-bit, ARMv8 facilitates 32-bit and 64-bit execution. Thus, ARMv8 differentiates between the AArch32 (binary compatible to ARMv7) and AArch64 execution state. Both have only moderate differences, which are of no relevance to our work. To ensure a consistent terminology, we prefer the term exception levels (ELs) over privilege levels and refer to AArch32 and AArch64.

Different exception levels restrict access to privileged resources. Similar to x86, the execution of OSes is mainly distributed across two processor modes in different exception levels (i.e., *User* and *Supervisor*). The notion of processor modes is limited to AArch32, yet, it can be equally applied to AArch64 as the distribution of software does not change. Figure 1 illustrates the hierarchy of execution levels and the associated processor modes on both AArch32 and AArch64. User applications execute in the less privileged EL0, whereas the OS kernel runs in the higher privileged EL1. Systems with hardware virtualization introduce EL2, into which the *Hyp* mode—dedicated for hypervisors or *Virtual Machine Monitor*s (VMMs)—is placed claiming the highest privileges.[2]

Similar to system-calls, guest VMs executing in EL0 and EL1 trap into the higher privileged VMM in EL2 (e.g., on privileged or sensitive instructions fetches, or on access to hardware devices managed

[2]In the following, we use the terms hypervisor and VMM interchangeably.

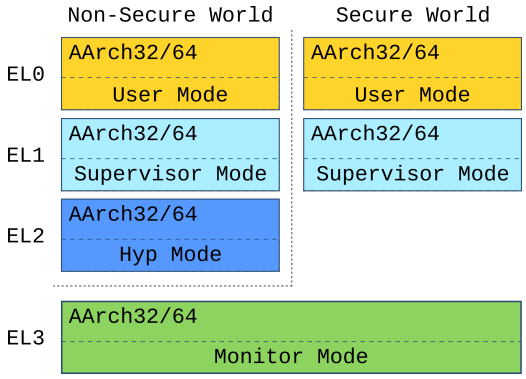


Figure 1: Exception Level hierarchy and mode distribution on AArch32 and AArch64 until ARMv8.3. For detailed architectural differences, please refer to [1].

by the VMM). This allows the guests to keep their original exception levels, while isolating the execution of the VMM. ARM also distinguishes between the *normal* and *secure world*, which is part of ARM's security extensions and referred to as *TrustZone*. While the normal world maintains all three exception levels, the secure world does not contain EL2 and hence does not support virtualization. Therefore, the secure world will not be further considered.

2.2 Guest Physical Memory Architecture

Hypervisors provide the illusion to guest VMs of having control of their physical memory. While guests maintain page tables for translating guest virtual to guest physical addresses, the VMM is responsible for translating guest physical into machine physical addresses. This lends VMMs a memory isolation property that ensures that even compromised guest's cannot manipulate other VMs or the VMM itself. On systems without virtualization support, the VMM maintains an additional set of page tables (*shadow page tables*) for managing the guest's physical memory in software. By write-protecting the guest's page tables, the VMM intercepts their modifications and redirects translations to a dedicated memory location. The downside is that the complex software management of shadow page tables entails a significant performance overhead.

To approach the poor performance of shadow page tables on hardware with virtualization support, the *Memory Management Unit* (MMU) features a supplementary level of indirection through the *Second Level Address Translation* (SLAT). Similar to shadow page tables, the hardware requires an additional set of page tables that is maintained solely by the VMM and cannot be accessed by the guest. These second stage translation tables complement the translation of guest virtual to guest physical addresses, by mapping the guest physical to machine physical addresses. Accesses to memory that is not mapped or lack access permissions in these tables result in traps allowing the VMM to isolate and control the guest's view on the physical memory. For instance, Intel's *Extended Page Tables* (EPTs) allow to define *execute-only* memory, which lends a VMM the ability to hide code instrumentation [7, 17]. In contrast to Intel and AArch64, SLAT on AArch32 does not support this functionality.

2.3 Debug Exceptions

Another feature that is imperative for further understanding concerns the implementation of breakpoints and watchpoints as well as single-stepping. ARM features a set of debug registers that can be configured to generate debug events. These events in turn generate debug exceptions which must be handled in dedicated exception handler routines that are typically set up by debuggers.

Both AArch32 and AArch64 support up to 16 configurable breakpoints. Every breakpoint can be set by means of the Breakpoint Control Register `DBGBCR` in conjunction with one of the Breakpoint Value Registers `DBGBVR`. Respectively, ARM supports up to 16 watchpoints, which function in a similar way. In the simplest case, a set breakpoint or watchpoint holds an instruction address, that generates an associated debug event on every instruction or data fetch. On top of that, ARM features software breakpoint instructions. The CPU generates debug events on execution of these instructions.

To single-step hit breakpoints, a monitor can configure `DBGBCR` to mismatch the breakpoint address in one of the `DBGBVR` registers: as addresses of instructions following the hit breakpoint do not match the address in `DBGBVR`, this will cause the CPU to generate a debug event on execution of every following instruction. Alternatively, AArch64 allows to generate Software Step exceptions by setting the `SS` bit of the Monitor Debug System Control `MDSCR_EL1` and Saved Program Status Register `SPSR` of the target exception level. For instance, to single-step a hit breakpoint in EL1 the monitor must set the `MDSCR_EL1.SS` and `SPSR_EL1.SS` bits. After returning to the trapped instruction, the `SPSR` will be written to the process state `PSTATE` register in EL1. Consequently, the CPU executes the next instruction and generates a Software Step exception.

To prevent disclosure of the analysis system, the VMM can intercept (and emulate) guest-access to debug registers and hence cover, e.g., set breakpoints controlled by the VMM. Yet, we highlight that adversaries can use the finite number of breakpoint and watchpoint registers as side channel information to reveal the analysis framework. Also, in-guest debugging cannot be perfectly emulated. For example, the KVM hypervisor implements VMM-based debugging on ARM with the restriction that the guest will be unable to use these features concurrently. Thus, hardware breakpoints and watchpoints, as well as single-stepping through breakpoint mismatching—the only way to single-step guest's on AArch32—are not suited for stealthy VMI.

Software Step exceptions on AArch64 are also visible to guest VMs. The VMM can intercept accesses to `MDSCR_EL1` and hide the `SS` bit. Also, ARM forbids direct access to the `PSTATE.SS` bit in all exception levels, complicating the discovery of analysis systems. Still, an adversary with control over the guest's exception handlers in EL1 can reveal the analysis by provoking an interrupt from EL1 that traps as well to EL1: in the exception handler, the `PSTATE` holding the set `SS` bit will be written to `SPSR_EL1` which in turn is accessible. We have validated this behavior as part of our evaluation (Section 4.4). Since accesses to `SPSR_EL1` cannot be intercepted, the VMM would need to trap and emulate every instruction in the exception handlers to cloak the analysis. This, however, is not a good alternative as the overhead of handling exceptions would rise enormously. As such, we are in need for *stealthy* single-stepping alternatives, upon which we place great emphasis in this paper.

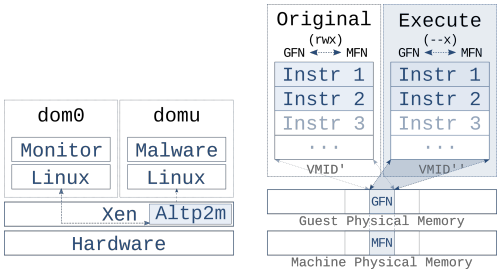


Figure 2: Xen altp2m enables a monitor in the privileged domain *Dom0* to manage different memory views of *DomU*. While the *execute-view* maps the target guest frame *execute-only*, the permissions of the *original-view* are unchanged.

2.4 Translation Lookaside Buffer

To counter the lack of *execute-only* memory on AArch32, we shift our focus toward the TLB organization. Virtual memory address translation entails high performance overhead. The reason for this is that the associated page tables reside in main memory. To increase performance, the TLB buffers recent guest virtual to guest physical address (and guest physical to machine physical) translation results. The TLB organization on x86 and ARM evolved to a split TLB architecture. A split TLB separates the TLB into two disjoint sets comprising the *instruction TLB* (iTLB) and *data TLB* (dTLB); the iTLB caches translated instruction fetches, whereas the dTLB holds translated data fetches. To further speed up memory translation, the TLB organization adds a superior caching level serving as a victim cache for the instruction and data TLB. On ARM, the introduced cache is called *unified TLB* (uniTLB) and is comparable to the *shared TLB* (sTLB) on x86. The uniTLB holds evicted entries from the iTLB and dTLB and is first consulted before walking the page tables.

To minimize TLB maintenance, the TLBs are associated or tagged with an identifier, which organizes the TLB entries based on a specific context. This means TLB entries with the same Address Space Identifier tag refer to a specific process. Similarly, entries tagged with the same *Virtual Machine Identifier* (VMID) refer to a specific VM or rather to a specific second level translation table (the VMID is part of the Virtualization Translation Table Base Register `VTTBR`). As such, the CPU does not need to flush the TLBs on context switches, thus significantly increasing the overall system performance.

2.5 Threat Model

We assume an adversary with root privileges, who possesses full control of the guest VM; she can access all security relevant parts of the OS, including the guest kernel and exception handlers. We also assume the attacker is indifferent to virtualized systems. Yet, she will abort her operation in case of disclosure of an analysis framework. Additionally, she can employ anti-virtualization techniques, such as carving the guest's memory for artifacts that may reveal the presence of virtualization-based analysis frameworks. These artifacts comprise instructions, such as software breakpoints and hypercalls, capable of redirecting the guest's execution to the VMM. Further, she can inspect the OS for agents in form of processes or kernel modules. Moreover, we assume she cannot modify the OS before her malicious code has been injected, thus she cannot manipulate any security critical vectors without us noticing it.

In this paper, we consider that the attacker can analyze in-guest register and thus reveal analysis frameworks that utilize debug registers, e.g., to hardware breakpoints, watchpoints, and the single-stepping mechanism. The attacker understands that although access to these registers can be intercepted and falsified by a VMM-based analysis framework, the VMM will not be able to cloak the resulting side effects. For instance, if the attacker is being emulated, she has the necessary means to discover it, due to imperfect emulation.

As part of our evaluation, we apply this threat model to the *adore-ng* rootkit to simulate a realistic attacker (Section 4.4).

3 GUEST KERNEL MONITORING PRIMITIVES

Malware can reveal analysis systems that use standard debugging techniques and change its behavior [2, 6]. Thus, non-stealthy analysis can cause false observations. Stealthy monitoring requires:

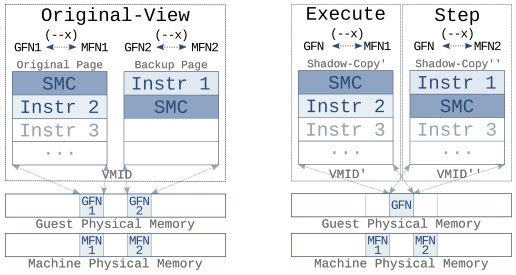
- **(R1)** a mechanism to intercept the guest in EL1 (guest kernel),
- **(R2)** a single-stepping mechanism that cannot be discovered by in-guest artifacts, and
- **(R3)** execute-only memory

Sadly, ARM does not support stealthy single-stepping (Section 2). On AArch32 the finite number of hardware breakpoints helps the attacker to infer the presence of an analysis framework; on AArch64, the attacker can spill the set `PSTATE.SS` bit into the `SPSR_EL1` register to uncover the analysis framework. Besides, AArch32 lacks execute-only memory (Section 2). As such, both architectures do not meet `(R2)` and AArch32 additionally fails to comply with `(R3)`.

To tackle these shortcomings, we present VMI primitives that enable monitoring of VMs, without relying on the intended hardware mechanisms. Instead, we employ virtualization extensions to intercept the guest kernel at arbitrary locations (aka. *tap points* [23]) and leverage such tap points to single-step trapped instructions in a stealthy way. Next, we extend Xen to control the guest's physical memory, which presents the foundation for stealthy monitoring on ARM. Finally, we combine these primitives to set the ground for stealthy VMI on ARM. Figure 2 illustrates an overall architecture of our system whose components are described in this section. In the following, we present primitives that, when combined, form stealthy monitoring systems on both AArch64 and AArch32.

3.1 Implementing Kernel Tap Points

A monitor can leverage SLAT to intercept the guest's execution at arbitrary locations. By withdrawing the *execute* permission from code pages containing functions of interest, the monitor can redirect the guest's execution in EL1 and EL0 to the VMM. Yet, this coarse-grained method incurs a high overhead: execution of code irrelevant for tracing on the page holding the target instruction would trap into the VMM. Instead, it is more desirable to monitor only the fact that the guest has executed a specific function. This demand can be met by software breakpoints. These instructions lend themselves as the instruction of choice to implement *tap points* `(R1)`. Yet, there exist other instructions that can be similarly configured to trap to the VMM and have additional properties that make them more desirable. The SMC instruction is the ideal candidate as it can be



(a) Without Xen altp2m. (b) With Xen altp2m.

Figure 3: On execution of the first SMC in the original page, the VMM redirects control-flow either to the backup page (a) or switches to the *step-view* (b) to single-step `Instr 1`. In both cases, the pages are marked *execute-only*.

configured to trap to the VMM and thus employed as a trigger to switch the execution-flow of the guest OS to the VMM.

The benefit of using the SMC instruction is that the guest is architecturally unable to subscribe to SMC traps. In contrast to software breakpoints, SMC traps can only be directed to TrustZone or to the VMM. This property reduces complexity of the monitor, as the execution of an SMC instruction never has to be re-injected into the guest. A limitation of the SMC in place of a software breakpoint is that it can only be executed in EL1, that is the guest kernel.

While the SMC instruction can thus be used for implementing tap points in the guest kernel (`R1`), achieving *stealthy* single-stepping without architectural support presents a great challenge. Without stealthy single-stepping we seem to reach an impasse on how to resume the execution of the guest kernel without losing control over the execution-flow. If we remove the tap point to allow the trapped vCPU to continue execution, an additional event needs to be triggered to place the tap point back. This event will normally be generated by a single-step exception. Further, in multi-vCPU domains, removing the tap point from memory is critical, as it may introduce a race condition: while removing the tap point, other vCPUs must not fetch the instruction from the same location.

3.2 Novel Single-Stepping Mechanism

The ARM architecture has an advantage over its x86 counterparts that we can leverage for a novel *single-stepping* scheme, without using the single-stepping feature of the CPU: ARM implements a fixed-width ISA. On x86, software breakpoints cannot be placed at arbitrary locations, as you may end up overwriting a part of a large instruction. On ARM, we can determine the position of the next instruction; depending on the execution mode, the width of instructions is known beforehand. Thus, we can locate instruction boundaries in memory without having to rely on a disassembler.

To illustrate how to utilize the fixed-width ISA for single-stepping, let us consider a scenario where we run a guest VM with a single vCPU. An external monitor with access to debug information of the target kernel, such as the *System.map* file on Linux, can determine the location of system-call handling kernel functions. By reading the first two instructions from the prolog of the target kernel function into a backup buffer and then overwriting the function's first instruction with an SMC, the monitor will intercept the guest kernel (`R1`) on execution of the marked kernel function. Upon execution of the SMC, the monitor can place the original first instruction back into memory while writing a second SMC in place of the immediately following instruction. When execution is resumed, the original instruction is executed followed by the execution of the second SMC. Now, the monitor can restore the original SMC without losing control over the guest kernel and hence conclude single-stepping of the first instruction in the system-call handler. On AArch64, we achieve stealthy single-stepping of the guest (`R2`) by configuring the instrumented page, holding the system-call handler, as execute-only (`R3`); reads and writes will trap into the VMM.

To achieve multi-vCPU safety, we store the first instruction at a location already mapped as executable, but unused at run-time. Here, we can safely place the second SMC after the stored instruction. The exact location of this memory is flexible as we only need space for two instructions per monitored location. For this, we can leverage known memory holes in the Linux kernel, such as the memory immediately following the kernel. For simplicity, we dedicate an entire page, *backup page*, for these two instructions (Figure 3(a)). In the figure, the *original-view* represents the physical memory that is made visible to the guest through SLAT. On execution of the first SMC in the system-call handler, we point the trapped vCPU's *Program Counter* (PC) to the *backup page* holding the original first instruction without performing any further modifications. Once the second SMC in the *backup page* is executed, we point the PC back to the instruction following the first SMC in the *original page*. While the above focuses on single-stepping the first instruction of system-call handlers, we can apply the same approach for arbitrary regions in the guest kernel. This however, must foresee corner-cases, such as function returns and branches, and thus requires the monitor to compute the target address.

3.3 Xen altp2m on ARM

Due to architectural differences between Intel and ARM, existing VMI solutions cannot be applied to the ARM architecture. To shift existing malware analysis tools that rely on the requirements (`R1`), (`R2`), and (`R3`) to the ARM architecture, we mimic the behavior of an effective approach for Intel, namely Xen *alternate p2m* subsystem (short *altp2m*). The original Xen altp2m has been exclusively used on Intel. In this architecture, a VM's memory view can be directly associated with an EPT represented by the *EPT pointer* (EPTP) in the hardware defined data structure *Virtual Machine Control Structure* (VMCS). The VMCS holds the host's and the guest's state and VM control information. It has capacity for up to 512 EPTPs—memory views—that can be dynamically switched. Xen altp2m is the first public implementation making use of this CPU feature, which makes it a unique tool for Virtual Machine Introspection [14].

We implement *altp2m* for ARM upon the Xen *p2m* subsystem. Xen p2m stands for *physical to machine* and leverages SLAT to manage and isolate memory between guest domains (Xen's notion for VMs) and the VMM. On ARM the *Virtualization Translation Table Base Register* (VTTBR) holds the base address of SLAT tables, similarly to the EPTP on Intel. Xen p2m maintains only one VTTBR. As such, the p2m subsystem maintains a *single* view of the guest's physical memory, even for VMs with multiple vCPUs. On the other

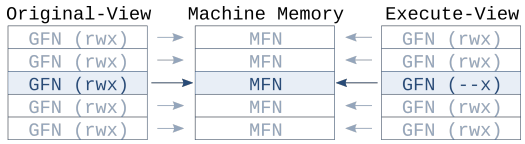


Figure 4: The *execute-view* maps the target guest frame as *execute-only*; the *original-view* maps it as *read-write-execute*.

hand, Xen altp2m on ARM allows to dynamically define and switch among *different* VTTBRs per domain and vCPU. The interaction with the altp2m interface takes place through dedicated hypercalls, called *HVMOPs*. These facilitate the privileged domain *Dom0* to create, switch, and destroy individual memory views that are then applied to unprivileged domains *DomU* (Figure 2). In addition, the altp2m interface allows to define memory access permissions of individual guest physical page frames per view and also remap individual guest frames to different machine frames.

VMI tools leverage SLAT to control access permissions of the guest's physical memory [7, 17]. When the guest traps into the VMM due to a memory access violation, the access permissions of the associated entries must be temporarily relaxed; the VMM must grant the required permission so the guest can continue. Yet, relaxing permissions in this ubiquitous view may allow one of the remaining vCPUs to access the targeted memory without notifying the VMM. One solution is to pause remaining vCPUs while single-stepping the trapped vCPU. This, however, imposes severe performance degradation. Also, the lack of stealthy single-stepping on ARM makes VMI-tools susceptible to disclosure. Xen altp2m solves such race conditions by maintaining *different* views of the guest's physical memory (Figure 2). Instead of changing permissions of a single memory view at run-time, altp2m allows to allocate a set of views beforehand. This way, a monitor can individually assign a specific memory view to each vCPU of *DomU*. As such, for instance on memory access violations, VMI-tools can switch the view of the affected vCPU to a less restrictive view, instead of explicitly relaxing permissions of the view that led to the trap; switching views is as simple as switching the domain's VTTBR.

Let us depict a scenario in which we monitor writes to critical regions (e.g., exception vectors) in a multi-vCPU domain. This scenario assembles the architecture in Figure 2. A monitor in *Dom0* allocates two distinct guest memory views that can be applied to each vCPU in *DomU*. The monitor grants all guest frames of the critical region *execute-only* permissions in the *execute-view*. The same guest frames keep their original permissions in the *original-view* (Figure 4); write attempts to the critical page by vCPUs with an active *execute-view* generate memory access violations. We configure this behavior as default on all vCPUs. On a potentially malicious write attempt, instead of relaxing permissions, the monitor switches the view of the trapped vCPU to the less restrictive *original-view*. This allows us to record the event, satisfy the write request, and avoid the target application to become suspicious. We do not relax permissions of other vCPUs and thus avoid race conditions. To ensure the monitor regains control immediately after the write, we single-step the trapped vCPU and switch back to the *execute-view*.

To further highlight the potential of Xen altp2m, we combine the single-stepping scheme in Section 3.2 with Xen altp2m. For every instruction to be single-stepped, an external monitor has to increase the guest's physical memory by two additional pages. This allows it to create two shadow-copies of the page holding the original instruction (we need two additional copies if we would like to satisfy code integrity checks that can be redirected to a view pointing to the original page). That is, similar to the above scenario, we allocate two additional guest memory views: the *execute-view* holds the first duplicate, *shadow-copy'*, while the *step-view* maps the *shadow-copy"* (Figure 3(b)). We replace the target instruction in the *execute-view* with a privileged SMC instruction. Then, instead of allocating a backup page in the same memory view, we replace the second instruction of the same function in the *shadow-copy"*. As such, on execution of the first SMC in the *execute-view*, the monitor can switch to the *step-view* without further adjustment. Finally, upon the execution of the target instruction, the execution of the adjacent SMC instruction in *step-view* traps again into the VMM, where we return to the *execute-view* and complete single-stepping.

These scenarios demonstrate the potential of Xen altp2m, enforcing memory restrictions through SLAT. The guest has no access to SLAT tables, as they reside in VMM's memory. Thus, such memory restrictions are stealthy. In fact, we meet the requirements (R1–R3) and hence establish a base for stealthy VMI on AArch64.

3.4 Splitting the TLBs

Sadly, AArch32 is not capable of enforcing *execute-only* pages; every code page has to be both *readable* and *executable*, or instruction fetches will fail. Thus, while (R1) and (R2) (partially) apply to AArch32, (R3) is not covered. Therefore, we cannot assure stealthy operation of Xen altp2m without further actions. To overcome this limitation we explore the TLB organization on ARM and implement a system we refer to as *split-TLB* that satisfies our requirements. [3]

ARM implements TLB-tagging to isolate translations tagged with different VMIDs. We made a joint decision with Xen maintainers that unique VMIDs will be assigned to each altp2m view. Contrary to x86, on ARM a TLB tag corresponds to a specific *memory view* instead of a *vCPU*; by switching altp2m views we activate the associated VMID, without having to flush differently tagged mappings of same guest physical memory. On the other hand, if different altp2m views shared a VMID, the guest would be susceptible to using stalled translations in the TLB, even though the active view contained the most recent mappings in memory. We choose to employ this architectural feature to hide modified code pages from data fetches and thus mimic *execute-only* memory on AArch32. As such, we extend the altp2m interface to pair the VMIDs of altp2m views to de-synchronize the physically separated *iTLB* and dTLB.

To cause an inconsistent state in the TLBs that we require for hiding code pages, we prime the iTLB so that it holds guest frame mappings that translate to different machine frames than those cached in the dTLB. That is, we require a mechanism that allows us to translate one guest frame to two physically different machine frames; only one of both mappings will be exclusively cached either in the iTLB or dTLB. To achieve this effect, first, we duplicate the

[3] The term *split-TLB* first appeared in the context of processor architectures. In this paper, we use this term also as a substitute for the de-synchronized TLB organization.

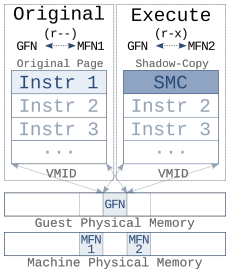


Figure 5: The *original-view* translates the guest frame to `MFN1`, while the *execute-view* translates the same guest frame to `MFN2`. Both views are tagged with the same VMID.

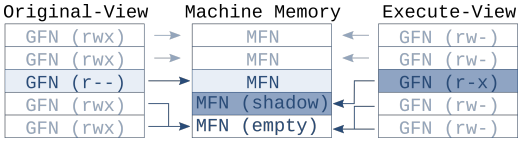


Figure 6: The *original-view* maps the target to the original machine frame; the *execute-view* to the shadow-copy.

page with the instruction to be monitored and replace it with an SMC instruction in the *shadow-copy* without modifying the *original page*. Then, we prepare two altp2m views and map both pages according to Figure 5. It is essential that both memory views are tagged with the same VMID; the system will ignore the primed iTLB entry, if it switches to a memory view with a different VMID. We grant the *original page read-only* permissions. We withdraw write permissions from the *original page* in the *original-view* to intercept write attempts. This allows us to monitor any change to the original page and propagate the modifications to the shadow-copy as required. Since AArch32 does not support execute-only mappings, we grant the *shadow-copy read-execute* permissions. Also, we withdraw the *execute* right from all other mappings in the *execute-view*, as to limit the execution in this view to the page of interest (Figure 6).

We configure the *original-view* to be active by default. On the first execution of the function to be monitored, the guest hands over control to the VMM: the instruction fetch violates the permissions of the *read-only* mapping. As such, the translation result does not get cached in the TLBs. The monitor leverages this architectural property to intercept the guest upon permission violation and switch to the *execute-view*, which grants execution access of the requested guest frame. This time, upon the successful SLAT table walk using the *execute-view*, the translation mechanism populates the iTLB with the machine frame that is associated with the *execute-view* (`MFN2` in Figure 5). Consequently, further instruction fetches from the page in question will directly consult the primed iTLB entry until it gets evicted. When the primed iTLB entry gets evicted it will need to be primed again. Upon execution of the SMC in the *execute-view*, the monitor can single-step the original instruction as described in Sections 3.2 and 3.3. After single-stepping the monitored instruction, the monitor switches back to the *original-view*.

The setup configures two views that map one guest frame to two machine physical frames with different access permissions. By priming the iTLB the target instruction is fetched from the *execute-view*, while reads use the *original-view*. As the iTLB and dTLB hold mappings from two different views, the primed system does not need the VMM to switch the views. This setup satisfies reads initiated, for example, by integrity checkers. At the same time, it transparently causes the guest to execute the SMC instruction in the *shadow-copy* of the *original page* (`R3`). Thus, split-TLB incurs minimal overhead, as it traps to the VMM only for the purpose of priming the TLBs or on execution of SMC instructions. Nevertheless, this setup entails a limitation that we discuss in detail in Section 5.2.

4 EVALUATION

To evaluate our work, we have implemented the discussed Xen altp2m subsystem and ported LibVMI [18] and the dynamic malware analysis framework DRAKVUF [17] to the ARM architecture. [4] We have equipped DRAKVUF with the presented VMI-primitives to assemble the foundation for stealthy guest kernel monitoring on AArch32 and AArch64. This allowed us to assess both the effectiveness and performance penalty of the introduced primitives.

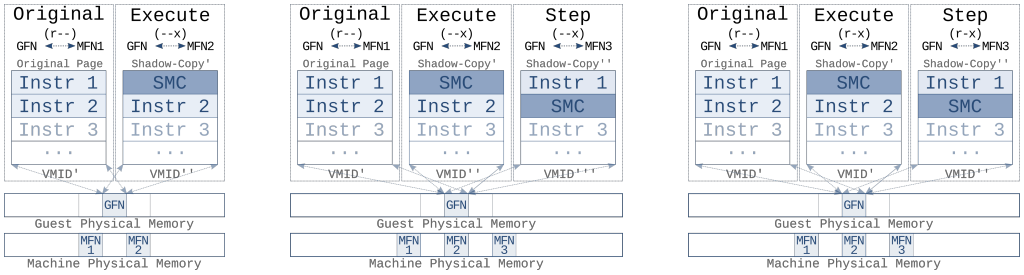
4.1 System Setup

Our system setup comprised an external monitor (i.e., DRAKVUF) running on top of the Xen hypervisor v4.11. DRAKVUF was executed in the privileged domain *Dom0* and it traced system-calls that were executed in the unprivileged domain *DomU*. The domains executed the Linux kernel v4.15. Generally, DRAKVUF uses OS-profiles statically generated by Rekall [26] to locate system-calls and set tap points in the prologue of each system-call handling function in the guest's kernel memory. This way, it establishes the means to intercept and monitor the guest's kernel behavior. Unfortunately, Rekall lacks the ability to generate profiles for Linux kernels compiled for AArch64. Consequently, we have implemented and open-sourced a custom Rekall profile generator to gather relevant AArch64 system-call and kernel data structure information. [5]

As part of our evaluation, DRAKVUF leveraged our Xen altp2m implementation [6] to dynamically create and switch among different guest memory views on ARM; it used the altp2m interface to communicate with the introduced subsystem from Dom0. We employed altp2m in combination with the discussed VMI-primitives that meet our requirements (`R1 – R3`) to stealthy monitor every system-call on AArch64 (Section 3). In total, our system-setup monitored 340 different system-calls which were distributed across 111 different 4 KB memory pages. In addition, we armed altp2m with the ability to take control over the TLB organization to assess the primitives for stealthy monitoring on AArch32 (Section 3.4).

We have set up our system on a HiKey LeMaker AArch64 development board running an ARM Cortex-A53 CPU with 1.2 GHz. Further, we have lent 1GB of RAM to the privileged domain *Dom0* and the target domain *DomU*. Although we performed all measurements on AArch64, the setup can be equally applied to AArch32.

[4] https://github.com/drakvuf-on-arm/drakvuf-on-arm
[5] https://github.com/drakvuf-on-arm/rekall-profile-generator
[6] https://github.com/drakvuf-on-arm/xen/tree/arm-altp2m-drakvuf



(a) Hardware-SS (b) Double-SMC-SS (c) Split-TLB-SS

Figure 7: Xen altp2m guest-physical memory view configuration leveraging three single-stepping alternatives.

4.2 DRAKVUF on ARM

To assess the presented VMI-primitives for the ARM architecture, we have combined these with DRAKVUF. The established foundation for DRAKVUF enabled thereby stealthy dynamic malware analysis on ARM. In addition, to demonstrate that our proposed single-stepping primitive (by using both synchronized and de-synchronized TLB configuration) can keep up with the performance of the hardware-supported approach, we have implemented a conventional, non-stealthy single-stepping mechanism for Xen in the way it is specified by the AArch64 specification [1]; this single-stepping alternative leverages dedicated hardware capabilities controlled through the `PSTATE.SS` bit on AArch64 (Section 2).

That is, within the scope of our evaluation, we have extended Xen, LibVMI, and DRAKVUF in such a way that we can employ the VMI-primitives for setting arbitrary *tap points* in the guest's kernel memory and protecting these through our *Xen altp2m* subsystem implementation. At the same time, we can combine these primitives with one of the three different single-stepping approaches:

- *Hardware-SS*: a non-stealthy single-stepping implementation that leverages hardware capabilities of AArch64 (Section 2).
- *Double-SMC-SS*: our stealthy method that leverages two SMC instructions protected by Xen altp2m (Section 3.3).
- *Split-TLB-SS*: an approach that additionally de-synchronizes the TLB organization to make up for the lack of execute-only memory (`R3`) on AArch32 (Section 3.4).

Regardless of the applied single-stepping method, DRAKVUF leverages Xen altp2m to set and protect tap points in the prologue of each system-call handling function. More precisely, we leveraged DRAKVUF to duplicate all pages holding system-calls to be monitored (Figure 7). Then, we allocated two guest memory views, whereas the first view (*original-view*) mapped the original page, the second view (*execute-view*) mapped the shadow-copy. We granted the original page in the *original-view read-only* permissions. The permissions of the shadow-copy in the *execute-view* were either *execute-only* on AArch64 or *read-execute* on AArch32. In both *Hardware-SS* and *Double-SMC-SS* setups, it was the *execute-view* that was active by default; *Split-TLB-SS* used the *original-view* as default. Finally, we replaced the first instruction of every system-call handling function in the shadow-copy with an SMC instruction. Further processing depends on the employed single-stepping method. The following exemplifies the individual steps taken by DRAKVUF to trace every invocation of a system-call inside the guest domain in accordance with one of the single-stepping alternatives.

Hardware-SS: To prevent the guest from discovering the instrumented SMC instruction in the shadow-copy, we mark the memory page mapped in the *execute-view* as *execute-only* (Figure 7(a)). This way, the execution of the first instruction of the system-call handler in the *execute-view* interrupts the guest kernel execution and hands control over to Xen, which in turn notifies DRAKVUF about the trap. Consequently, DRAKVUF switches back to the *original-view*, single-steps only one instruction (i.e., the original first instruction of the system-call handling function) via the dedicated hardware mechanism, and resumes execution in the *execute-view*. All read-requests to code pages that are marked *execute-only* in the *execute-view* can be satisfied by switching to the *original-view*. Also, we intercept write-requests to propagate potential changes to all views.

Double-SMC-SS: In this context, we use the same *original-view* and *execute-view* configurations as in *Hardware-SS*. In addition, we create a third view, *step-view*, which we use to single-step instructions without the intended hardware capabilities. The *step-view* maps a second copy of the original page, in which we replace the *second* instruction of the system-call handler function with a second SMC (Figure 7(b)). This way, on interception of the first SMC in the *execute-view*, DRAKVUF switches to the *step-view* to execute and trap immediately after the first instruction. The second SMC instruction in the *step-view* facilitates DRAKVUF to switch back to the *execute-view* and continue execution right after the first SMC.

Split-TLB-SS: As AArch32 lacks *execute-only* memory, we granted *read-execute* permissions to the shadow-copies in the *execute-view* and *step-view* Figure 7(c)). To de-synchronize the entries in the iTLB from the dTLB, we used the same VMID for the *original-* and *execute-view*. We primed the iTLB such that instruction fetches from the target page accessed the *execute-view* in the iTLB; data access consulted the *original-view* in the dTLB. Right after executing the first SMC, DRAKVUF dynamically switched to the *step-view* to single-step the original first instruction similarly to *Double-SMC-SS*.

In all configurations, every time a system-call trapped into Xen, it notified DRAKVUF about the event, which monitored the system-call for further processing; on every system-call, DRAKVUF intercepted the guest, monitored the event, single-stepped the trapped instruction according to the applied single-stepping alternative, and resumed the guest. As such, during benchmarking, our monitor was overwhelmed with a persistent shower of system-calls.

Table 1: Monitoring overhead (OHD) of DRAKVUF utilizing Hardware-SS, Double-SMC-SS, and Split-TLB-SS primitives measured by Lmbench 3.0, in *msec*.

Benchmark	w/o	Hardware	(OHD)	Double-SMC				Split-TLB			
				Step-View	(OHD)	Backup Page	(OHD)	Step-View	(OHD)	Backup Page	(OHD)
fork+execve	1383.33	6053.67	4.38 ×	5567.33	4.02 ×	6033.00	4.36 ×	26690.66	19.29 ×	17057.00	12.33 ×
fork+exit	377.43	835.52	2.21 ×	787.14	2.09 ×	924.83	2.45 ×	5910.83	15.66 ×	4225.83	11.20 ×
fork+/bin/sh	3249.17	12542.00	3.86 ×	11672.67	3.59 ×	12737.33	3.92 ×	53134.66	16.35 ×	34231.33	10.54 ×
fstat	0.62	94.94	152.57 ×	78.65	126.40 ×	84.20	135.81 ×	103.52	166.97 ×	75.33	121.06 ×
mem read	1745.00	1692.33	0.97 ×	1692.33	0.97 ×	1738.00	1.00 ×	1730.33	0.99 ×	1735.33	0.99 ×
mem write	4687.67	4310.00	0.92 ×	4308.33	0.92 ×	4715.00	1.00 ×	4575.33	0.98 ×	4602.00	0.98 ×
open/close	5.44	202.67	37.25 ×	158.33	29.11 ×	179.26	35.95 ×	269.67	49.57 ×	184.65	33.94 ×
page fault	1.49	1.72	1.15 ×	1.74	1.16 ×	1.62	1.09 ×	1.90	1.28 ×	1.91	1.28 ×
pipe lat	12.26	371.92	30.34 ×	344.83	28.13 ×	425.28	34.69 ×	955.53	77.94 ×	482.60	39.36 ×
read	0.67	95.21	141.14 ×	79.10	117.27 ×	84.06	125.46 ×	99.34	148.27 ×	75.39	111.77 ×
select 500 fd	28.33	124.62	4.40 ×	110.23	3.89 ×	114.51	4.04 ×	124.47	4.39 ×	113.85	4.02 ×
signal handle	4.34	189.67	43.70 ×	150.33	34.64 ×	154.13	35.51 ×	178.00	41.01 ×	158.33	36.48 ×
signal install	0.51	95.00	186.27 ×	72.00	141.18 ×	75.13	147.31 ×	89.07	174.65 ×	73.73	144.58 ×
stat	2.63	99.97	38.06 ×	80.73	30.74 ×	85.30	32.43 ×	105.58	40.14 ×	83.57	31.82 ×
syscall	0.31	94.21	299.05 ×	75.15	238.55 ×	83.49	269.32 ×	98.48	317.68 ×	78.84	250.26 ×
write	0.47	95.34	203.32 ×	76.82	163.81 ×	83.86	178.43 ×	103.22	219.62 ×	73.77	157.31 ×

4.3 Performance

Automated VMI-based malware analysis strongly affects the overall system performance. As such, the VMI-induced performance overhead must be kept to a minimum. To evaluate the performance overhead of the introduced VMI-primitives, we have conducted two experiments comprising a set of CPU-intensive macro- and micro-benchmarks, during which we used DRAKVUF to monitor every system-call that was set off by the benchmarking tools in *DomU*. To solely focus on the monitoring overhead, we have deactivated the output to the console. All results are means over three runs.

In the first experiment, we used DRAKVUF to trace all system-calls that were set off by a set of *lmbench 3.0* micro-benchmarks. This allowed us to analyze the induced performance cost on system-software level (Table 1). To be more precise, first, we have executed DRAKVUF in combination with the *Hardware-SS* approach that leveraged the AArch64 hardware architecture to single-step protected tap points and determined the overall execution overhead. In this way, we have established a baseline which we then have compared with DRAKVUF's performance using both of our proposed single-stepping variants, namely *Double-SMC-SS* and *Split-TLB-SS*. Besides, as both variants can leverage a *backup page* in the *execute-view* instead of using an additional *step-view* (Figure 3(a)), we examined the performance of both configurations for each single-stepping primitive. Overall, the results show that the performance of *Double-SMC-SS* and *Split-TLB-SS* are close to our baseline implementation for single-stepping and thus present suitable alternatives.

Surprisingly, we have observed that in most cases the *Double-SMC-SS* implementation outperforms *Hardware-SS*. One can argue that our implementation responsible for managing the hardware-supported single-stepping in Xen requires additional overhead that might result in the observed behavior. However, we believe that the hardware logic behind stepping one instruction requires more time than simply intercepting the execution of an SMC.

The performance of the *Split-TLB-SS* approach strongly depends on the number of memory pages are involved in the system-call. That is, since we have only a very limited number of iTLB-entries (10 iTLB entries on ARM Cortex-A53), as soon as the system-call accesses memory pages that are not yet part of the iTLB, primed iTLB entries might get evicted according to the TLB eviction strategy.

Table 2: Virtualization overhead (OHD) induced by DRAKVUF, utilizing Hardware-SS, Double-SMC-SS, and Split-TLB-SS primitives by the Phoronix Test Suite v7.6.0.

Benchmark (*unit*)	w/o	Hardware	(OHD)	Double-SMC	(OHD)	Split-TLB	(OHD)
Gzip (*s*)	416.81	466.78	11.99 %	568.10	36.30 %	1209.24	190.12 %
N-queens (*s*)	779.78	779.79	0.00 %	779.78	0.00 %	779.76	0.00 %
7-Zip (*MIPS*)	612.67	616.00	-0.54 %	612.00	0.11 %	550.33	10.17 %

In our second experiment, we have conducted a set of CPU-intensive macro-benchmarks of the *Phoronix Test Suite v7.6.0* and summarized the results in Table 2. Please note that the units of our measurements vary. The collected results suggest that DRAKVUF incurs only limited overhead on the overall system performance and thus is very well suited for efficient malware analysis on ARM.

4.4 Effectiveness

To demonstrate the effectiveness of DRAKVUF on ARM, we have set up our system to analyze the *adore-ng* rootkit on ARM. By setting a tap point to trace `kallsyms_lookup_name`, we identiefied that the rootkit determined the location of the kernel function for kernel hot patching (`aarch64_insn_patch_text`). In particular, it created kernel hooks required, among others, to hide files, processes, and logs and to communicate with the rootkit. As such, we set another tap point to `aarch64_insn_patch_text` and hence observed all malicious writes to kernel regions holding sensitive function hooks.

We have further lend the *adore-ng* a split-personality property that was looking for artifacts that could reveal our monitor. When the rootkit had detected our analysis framework, it terminated its operation. With this property, *adore-ng* was able to successfully uncover our system when it leveraged *Hardware-SS* to trace and single-step the rootkit (Section 2.3). Also, the rookit was able to reveal SMC instructions by synchronizing the TLBs, when we applied our *Split-TLB-SS* scheme on AArch32. The exact steps to disclose *Split-TLB-SS* are described in Section 5.2. Yet, the *Double-SMC-SS* method on AArch64 remained undisclosed. Consequently, as we can use the AArch64 architecture to also trace AArch32 guests, we deem this analysis method stealthy for both architectures.

5 DISCUSSION

This section discusses alternative tracing methods and reviews the limitations of the proposed VMI primitives.

5.1 Alternative Tracing Methods

Our approach necessitates two guest context switches to trap on and single-step one instruction. If we limit ourselves to tracing the Linux kernel, we can adapt the functionality of *ftrace* [30], a tracing framework for Linux kernel analysis. Assuming a Linux kernel compiled with the `CONFIG_DYNAMIC_FTRACE` parameter, the prolog of white-listed kernel functions holds a call to a dedicated stub, calls to which allow ftrace to record the function call. When tracing is disabled, this stub is filled with `NOP` instructions. We can reuse the call to the function stub in every kernel function by placing an SMC instruction to this position and protecting it through Xen altp2m. This approach eliminates the need for the second SMC (Figure 3), as we do not need to replace and single-step any instructions.

This approach reduces the single-stepping overhead. Yet, it limits itself to monitoring only Linux guests with ftrace support. Also, tracing has to be deactivated; an adversary can use this knowledge to reveal the monitor. On the other hand, the monitor can fall back to our default approach if tracing is activated. In contrast, our single-stepping method (Sections 3.2 and 3.3) is not limited to tracing only Linux kernels and can single-step functions at arbitrary locations by considering corner-cases affecting the control-flow. This renders our design capable of monitoring all guest kernel locations.

Besides, our single-stepping schemes create up to three altp2m views, whereas each maps an own variant of the original page (Figure 7). I.e., we consume up to two additional pages per page holding the target function. Instead of creating an additional *step-view*, we can use a *backup page* holding the original instruction and a second SMC per tap point (Figure 3(a)). Thus, we can reduce the pages as one *backup page* has capacity for up to 512 tap points.

5.2 Limitations

The discussed techniques present novel approaches for stealthy monitoring the kernel space in multi-vCPU guest domains. Our implementation of Xen altp2m on AArch64 in connection with *de-synchronizing* split-TLBs on AArch32 allows us to hide arbitrary code from the guest. Yet, our prototype entails following limitations.
Applicability: By employing SMC instructions as a trigger to switch the control flow to the VMM, we limit ourselves to only intercepting the execution of EL1, which is the guest's kernel space. While providing the means to hide arbitrary code in the user space, we chose the SMC instruction because guests cannot subscribe to SMC events. This concept reduces the complexity of the monitor, as it eliminates the need for any SMC event injections into the guest. A great alternative to the SMC instruction is the *Branch Exchange Jazelle* (BXJ) instruction [4]. This instruction can be executed in EL0 and can be configured to trap into EL2. Yet, while BXJ can be executed by AArch32 guests, AArch64 guests do not support this instruction. Another alternative can be implemented by instructions accessing memory that is known to be protected by the monitor.
Robustness: Our *Split-TLB-SS* solution cannot guarantee ongoing stealthy operation on AArch32 CPUs that implement a uniTLB. On such systems, attackers can re-synchronize the TLBs and reveal SMC instructions from inside the guest. Therefore, she must synchronize the TLBs to detect pages hidden in the iTLB. This can be achieved by forcing the system to evict the primed iTLB entries to the uniTLB and fetch them to the dTLB. The attacker must ensure that the dTLB does not contain valid mappings of the affected pages, as they would satisfy the guest's read and write requests. Also, if the primed entries in the iTLB get evicted, while having valid mappings in the dTLB, the VMM will be able to re-prime the iTLB. As such, the guest must only evict the dTLB. Explicit evictions result from flush operations. Implicit evictions result when the iTLB or dTLB buffer gets full and needs to store a new entry; the hardware evicts one of the entries from either the iTLB or dTLB to the uniTLB. The adversary knows that ARM holds a finite number (in our case 10) of fully associative entries in the iTLB and dTLB. Upon allocation of pages, she can read or write to them forcing the system to implicitly flush the dTLB. Hence, the next data access to the target page will consult the uniTLB. Next, the attacker flushes the iTLB to the uniTLB in a similar way to ensure the primed entries are available for future access through the dTLB. Subsequent reads from the target address will consult the uniTLB and reveal the hidden SMC.
Scope: On AArch32, we further restrict our scope to tracing the execution of code pages in kernel space, which do not perform integrity checks on themselves. This applies to the majority of the kernel, including system-call handlers. This limitation emerges as we cannot hide injected SMC instructions at arbitrary positions in the kernel space; AArch32 specifies executable pages to be marked with *read-execute* access privileges. Thus, for instance, we distance our mechanisms from monitoring dynamically loaded kernel modules that might perform integrity checks of routines located on the same page as the checking mechanisms themselves.

Besides, the analyst must be aware of system-calls that are mapped into the *Virtual Dynamic Shared Object* (VDSO), a shared library mapped into the address space of user space applications. It is used to increase performance of frequently called system-calls, by eliminating the context-switch overhead. This way, frequently called system-calls are mapped to user space and therefore cannot be monitored through SMC instructions. On ARM, however, symbols that are exported through the VDSO into user space are limited (i.e., four symbols on AArch64 and two symbols on AArch32).
Stealth: Some of the problematic anti-virtualization categories deal with behavioral discrepancies between physical and virtual environments. These comprise timing overhead induced by emulation or analysis. Side effects of certain instructions can differ as they are not sufficiently documented. This category can only be partially addressed. The hardware behavioral knowledge can be gathered through massive testing [27] and simulated by a VMM. Yet, if an attacker has access to external time sources, such as NTP, she will be able to detect discrepancies caused by the virtualization overhead.

6 RELATED WORK

An analysis framework must be stealthy to avoid perturbing malware. SPIDER [7] is a stealthy debugging and instrumentation framework based on Linux KVM. In addition, DRAKVUF [17] is a VMI-based, automated dynamic malware analysis framework built on top of LibVMI [18] and Xen. SPECTRE [39] facilitates a stealthy analysis framework by operating on a level below the

hypervisor. In a similar fashion, MALT [38] provides debugging capabilities that can be employed from remote. SPROBES [13] utilizes ARM TrustZone to enforce kernel integrity. Similarly, Ninja [21] leverages TrustZone and also ARM's performance monitor unit to transparently analyze malware. Also, Lengyel et al. [15, 16] explore concepts that may be leveraged for VMI with Xen on ARM. Finally, the WhiteRabbit [25] VMI framework combines on-the-fly virtualization with VMI on x86 and ARM. Our primitives combined with WhiteRabbit would establish a stealthy monitor that could be deployed on systems that were not explicitly set up for VMI.

In the opposite direction, Shadow Walker rootkit [28], abuses the split organization of TLBs for stealth purposes. Similarly, Wurster et al. [34] defeat integrity checks. Additionally, the MoRE Shadow Walker [31] demonstrates that modern, hybrid TLB organizations with an additional shared TLB level are prone to de-synchronization techniques. Grsecurity, on the other hand, de-synchronizes the split-TLB architecture in PAGEEXEC [22] to overcome the lack of hardware supported execute-only pages. These mechanisms mainly focus on the x86 architecture. Also, the presented approaches require invasive kernel changes or a dedicated hypervisor. In contrast, our approach employs capabilities of the open source Xen hypervisor to de-synchronize TLBs on both ARMv7 and ARMv8 architectures facilitating stealthy monitoring of guest domains.

7 CONCLUSION

In this paper, we proposed novel techniques that facilitate stealthy monitoring of guest OSes on ARM. We overcame ARM's lack of hardware support for stealthy VMI by presenting primitives which empower monitoring of guest OSes. These primitives establish an alternative way of setting and single-stepping software breakpoints without using the intended hardware mechanisms. We extended the Xen Project hypervisor to leverage SLAT to define and dynamically switch among different guest physical memory views. To this end, we introduced the first system on ARM capable of holding multiple guest memory views in parallel which presents a stealthy solution on AArch64. We further combined the above techniques with peculiarities of the TLBs to overcome the lack of *execute-only* memory on AArch32 by de-synchronizing the TLB organization. We have examined this approach and identified its inherent advantages and limitations. In conclusion, we believe that our methodologies can establish powerful covert VMI analysis systems on ARM.

REFERENCES

[1] ARM. 2017. *ARM Architecture Reference Manual, ARMv8 for ARMv8-A Architecture Profile (DDI 0487C.a).*

[2] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient Detection of Split Personalities in Malware. In *ISOC Network and Distributed System Security Symposium (NDSS).*

[3] Bitdefender. 2018. Bitdefender. http://www.bitdefender.com/.

[4] Robert Buhren, Julian Vetter, and Jan Nordholz. 2016. The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture.

[5] Peter M. Chen and Brian D. Noble. 2001. When Virtual Is Better Than Real. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS).*

[6] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*

[7] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization. In *Annual Computer Security Applications Conference (ACSAC).*

[8] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *ACM Conference on Computer and Communications Security (CCS).*

[9] Ferrie, Peter. 2007. Attacks on More Virtual Machine Emulators. *Symantec Technology Exchange* (2007).

[10] FireEye. 2018. FireEye. https://www.fireeye.com/.

[11] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility Is Not Transparency: VMM Detection Myths and Realities. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS).*

[12] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *ISOC Network and Distributed System Security Symposium (NDSS).*

[13] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. In *IEEE Mobile Security Technologies Workshop (MoST).*

[14] Tamas K Lengyel. 2016. Stealthy Monitoring With Xen Altp2m. https://blog.xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m.

[15] Tamas K. Lengyel, Thomas Kittel, and Claudia Eckert. 2015. Virtual Machine Introspection With Xen on ARM. In *Workshop on Security in highly connected IT systems (SHCIS).*

[16] Tamas K Lengyel, Thomas Kittel, Jonas Pfoh, and Claudia Eckert. 2014. Multi-Tiered Security Architecture for ARM via the Virtualization and Security Extensions. In *International Workshop on Database and Expert Systems Applications (DEXA).*

[17] 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 *Annual Computer Security Applications Conference (ACSAC).*

[18] LibVMI. 2018. LibVMI Virtual Machine Introspection. http://libvmi.com.

[19] Linux Foundation. 2018. Xen Project. https://www.xenproject.org/.

[20] Litty, Lionel and Lagar-Cavilla, H. Andrés and Lie, David. 2008. Hypervisor Support for Identifying Covertly Executing Binaries. In *USENIX Security Symposium.*

[21] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards transparent tracing and debugging on arm. In *USENIX Security Symposium.*

[22] PaX Project. 2018. Pageexec. http://pax.grsecurity.net/docs/pageexec.txt.

[23] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *IEEE Symposium on Security and Privacy (S&P).*

[24] Jonas Pfoh, Christian Schneider, and Claudia Eckert. 2011. Nitro: Hardware-Based System Call Tracing for Virtual Machines. In *International Workshop on Advances in Information and Computer Security (IWSEC).*

[25] Sergej Proskurin, Julian Kirsch, and Apostolis Zarras. 2018. Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection. In *IFIP International Conference on ICT Systems Security and Privacy Protection (IFIP SEC).*

[26] Rekall Forensics. 2018. Advanced Forensic and Incident Response Framework. http://www.rekall-forensic.com/.

[27] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. 2014. Cardinal Pill Testing of System Virtual Machines. In *USENIX Security Symposium.*

[28] Sherri Sparks and Jamie Butler. 2005. Shadow Walker: Raising the Bar for Rootkit Detection. *Black Hat, Japan* (2005).

[29] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *ISOC Network and Distributed System Security Symposium (NDSS).*

[30] The Linux Kernel. 2018. Ftrace – Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[31] Jacob Torrey. 2014. MoRE Shadow Walker: TLB-splitting on Modern X86. *Black Hat, USA* (2014).

[32] VMRay. 2018. VMRay GmbH. https://www.vmray.com.

[33] Sebastian Vogl, Fatih Kilic, Christian Schneider, and Claudia Eckert. 2013. X-Tier: Kernel Module Injection. In *International Conference on Network and System Security (NSS).*

[34] Glenn Wurster, Paul C van Oorschot, and Anil Somayaji. 2005. A Generic Attack on Checksumming-Based Software Tamper Resistance. In *IEEE Symposium on Security and Privacy (S&P).*

[35] Xen Project. 2018. Xen Security Advisory 203. https://xenbits.xen.org/xsa/advisory-203.html.

[36] Xen Project. 2018. Xen Security Advisory 204. https://xenbits.xen.org/xsa/advisory-204.html.

[37] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security Symposium.*

[38] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *IEEE Symposium on Security and Privacy (S&P).*

[39] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. 2013. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).*