The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture

Robert Buhren(✉), Julian Vetter, and Jan Nordholz

Technical University Berlin, Berlin, Germany
{robert,julian,jnordholz}@sec.t-labs.tu-berlin.de

Abstract. The virtualization capabilities of today's systems offer rootkits excellent hideouts, where they are fairly immune to countermeasures. In this paper, we evaluate the vulnerability to hypervisor-based rootkits of ARM-based platforms, considering both ARMv7 and ARMv8. We implement a proof-of-concept rootkit to prove the validity of our findings. We then detail the anatomy of an attack wherein a hypervisor rootkit and a userspace process collaborate to undermine the isolation properties enforced by the Linux kernel. Based on our discoveries, we explore the possibilities of mitigating each attack vector. Finally, we discuss methods to detect such highly privileged rootkits so as to conceive more effective countermeasures.

Keywords: Rootkit · Hypervisor · ARM · Virtualization

1 Introduction

In the ongoing malware arms race, adversaries try not only to take over systems but retain control over the system for as long as possible. As higher privileged levels implement the abstractions and define the boundaries that all lower privileged layers have to adhere to, sophisticated attacks always try to infect the highest privileged layer of a system, ideally a higher privileged layer than the one used by the defender's detection mechanism.

Soon after virtualization extensions by Intel and AMD publicly appeared in 2005 King et al. [16] proposed the first VM-based rootkit in 2006. The technique then became famous under the name *Bluepill* [19,20], whereby an adversary installs a malicious hypervisor during normal execution of the OS to take control over all system resources. On the ARM architecture, just like on the x86 architecture before, the arms race is well underway. A considerable number of rootkits exist (e.g. [8,22,23]) that infiltrate the OS kernel and maintain control over the system. However, the question whether the technique of VM-based rootkits is applicable on the ARM architecture remains open.

In this paper, we want to address the open research question as to whether the construction of a hypervisor rootkit is feasible on the ARM architecture. First we are going to answer whether it is possible to install a rootkit into the

K.-Y. Lam et al. (Eds.): ICICS 2016, LNCS 9977, pp. 376–391, 2016.
DOI: 10.1007/978-3-319-50011-9_29

hypervisor mode to subvert a running OS kernel. Moreover, we answer the question concerning the detectability of the rootkit. Previous work has thoroughly discussed the detectability of hypervisors on the x86 architecture [13,15,21,26]. However, because of the fundamental differences in the design of the virtualization extensions between ARM and x86, the earlier findings cannot be simply extrapolated.

Contribution. We make the following contributions: (1) We determined three attack vectors on currently deployed Linux systems that allow us to install a rootkit into the hypervisor mode and subvert the running OS. Each vector is applicable in a different scenario, which proves the versatility of the attack, and allows us to attack a broad range of devices. (2) We built a very small rootkit that is installed into hypervisor mode to gain full system control and optionally provide malicious services. The rootkit image is a mere 16 kB in size, with 95 % attributed to page table data and alignment. (3) We evaluated our rootkit and discuss potential detection mechanisms. We identify a new and reliable way to detect it by exploiting characteristics of the ARM TLB. (4) We present a number of mitigation techniques to seal the hypervisor mode and prevent our attack.

2 ARM Virtualization

This section introduces the ARM virtualization extensions as far as needed to understand the remainder of the paper. Well-experienced readers may skip ahead but should note our preference for ARMv8 terminology (see below).

Version 7 and earlier versions of the ARM architecture define seven execution modes. One of these is unprivileged and operates at *privilege level 0* (PL0), whereas the other six are privileged and collectively referred to as *privilege level 1* (PL1). ARMv8 combines the privileged execution modes into a single one, thus allowing for simpler exception and interrupt handler code. It also slightly changes nomenclature and coins the new term *exception level*, but leaves the numbering and their meaning basically unmodified, thus renaming PL0 to EL0 and PL1 to EL1. In addition systems with virtualization extensions have an additional execution mode. This mode is located in the new privilege level EL2 (PL2 for ARMv7), placed above EL0 and EL1.

The ARM architecture provides an additional separation concept that is orthogonal to privilege levels. TrustZone [3,6] introduces the notion of a "secure world", which mirrors the privilege levels of the classical "non-secure world". In addition, a new execution mode, monitor mode (*mon*), facilitates the switch between the two worlds. There is one notable difference though between ARMv7 and ARMv8 with respect to TrustZone. As the world switch component was classically provided by the Secure OS as well, *mon* mode was added to the ARMv7 PL1 modes, and switching between secure *svc* and *mon* was seamlessly possible. ARMv8 moved the monitor mode into a level of its own at the top of the hierarchy, EL3, so that it can no longer be entered freely from the secure world. This has implications for one of our attack vectors.

As both terms PL and EL mean virtually the same, we have chosen to stick with "EL" for the remainder of this paper, i.e. we prefer ARMv8 terminology. Unless stated otherwise, all statements apply to both ARMv7 (with PL substituted in place of EL) and ARMv8.

3 Attack Model

In the following section, we discuss the assumptions and requirements for our attack as well as the scope and focus of the attacks.

In the considered attack scenario, depicted in Fig. 1, an adversary first gains control of a user-level process (Fig. 1 ①) and then manages to exploit a kernel vulnerability (Fig. 1 ②). Vulnerabilities in the Linux kernel appear frequently enough [9] to make our assumptions sound. Once having kernel access, the adversary is able to manipulate the OS at will, but he is still visible to the OS and exposed to scanners executing directly in kernel mode or as a highly privileged process. Therefore, the adversary hides by moving to the more privileged hypervisor mode ③. From there, he can put away the OS into a virtual machine, eliminating the risks of being detected from an EL1 scanner (Fig. 1 ④). During the infection phase, the rootkit is briefly exposed to a scanner running in EL1; however, as we show later in the paper (Sect. 8), the time frame is small.

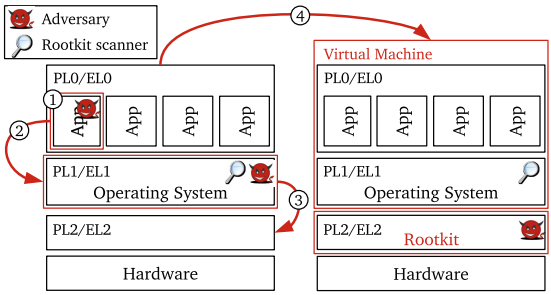

Fig. 1. From a compromised application (①), an adversary compromises the OS kernel (②). Then he gains hypervisor privileges (③) before the victim OS is put away into a VM (④).

4 Entering EL2

In the following, we describe several ways to plant code into EL2. The key observation is that being able to overwrite the exception vector table address for EL2 is sufficient for that end. After the adversary has placed his own base address, he can easily trigger an exception from EL1 that traps into EL2, executing one of his planted handler vectors. So each of the following attack vectors manages to overwrite the value contained in `VBAR_EL2`, thus enabling the adversary to gain control on the next EL2 intercept.

We want to note that except for attack vector 3, all described attack vectors were tested on both an ARMv7 and ARMv8 processor[1].

Attack Vector 1 - Linux Hypervisor Stub. Current versions of the Linux kernel check the EL they are booted into. If they find themselves in EL2, they install a stub vector table address before dropping down to EL1. The purpose of this stub is to allow a type-II hypervisor implementation (e.g. KVM) to install its own vector table, thus gaining the world switch capabilities required for its hypervisor duties. This stub only consists of four lines of assembly, which provide support for querying and writing the vector table base address. KVM uses this facility in the following way. It stores its EL2 vector table at some memory location and loads the base address into register `r0`. Then it executes the `hvc` instruction. The stub simply copies `r0` to `VBAR_EL2` and returns. All subsequent calls after this installation procedure are then handled by KVM's vector table. Thus KVM has acquired EL2 privileges and can use these to control and switch between virtual machines.

The installation of the stub vector table depends only on the bootup EL, Linux provides no way to turn it off. If no KVM module is available or the adversary can mount his attack before KVM is loaded, this provides easy control over EL2.

Attack Vector 2 - KVM Hyp Call Function. The KVM hypervisor on ARM uses "split-mode" virtualization [10,11], i.e., parts of the hypervisor code run in EL1. Only code that explicitly needs access to functionality that is only present in the hypervisor mode runs in EL2. The EL1 part is called "high-visor" and the EL2 part is called "low-visor". The "host" Linux is still running in EL1. When KVM is loaded, it installs its own `VBAR_EL2`, using the previously mentioned hypervisor stub. This prevents an adversary from planting his own code. However, KVM also offers the possibility for the Linux kernel to provide a function pointer to the "low-visor" running in EL2[2], i.e. there is no well defined API between low- and high-visor, but arbitrary code execution in EL2 is always possible. This mechanism can be used to trivially replace the exception vector table for EL2.

Attack Vector 3 - Migrate Linux to Non-secure (ARMv7 only). Some systems run their normal world OS completely in the secure world. This simplifies the system deployment because the bootloader does not have to configure the secure world and then switch to the normal (non-secure) world. When the system does not need the secure world, this seems like a valid scenario. In the secure world all, registers are named exactly the same as their non-secure counterparts. Therefore, an OS can either run in secure or non-secure EL1 without any changes.

However, as the secure EL1 has control over the non-secure EL2, an adversary running in secure EL1 can manipulate registers belonging to the non-secure EL2.

[1] Attack vector 3 only works on the ARMv7 architecture due to changes in the design of the processor modes (for details refer to Sect. 2).

[2] This function is defined as `kvm_call_hyp` on both ARMv7 and ARMv8.

But after installing itself into non-secure EL2, the adversary does not have any control over the OS yet, because it is still running in secure EL1. So first the OS has to be migrated from secure EL1 to non-secure EL1. Apart from copying register values from their secure counterpart, the adversary has to configure the interrupt controller so that all interrupts arrive in the normal world instead of in the secure world. After duplicating the processor state and installing his malicious code, the adversary can resume the execution in the non-secure EL1. The state duplication is the critical step for this attack vector. Installing code into EL2 afterwards is trivial because all EL2 registers are writable from secure EL1.

5 EL2 Rootkit Requirements

In order to design a hypervisor-based rootkit (a rootkit that runs in EL2), we identified three crucial aspects. These are: *Resilience*, *Evading detection* and *Availability*. Each point is addressed in the following section.

5.1 Resilience

Even though the rootkit executes in EL2, the code pages of the rootkit are memory pages managed by the *victim OS*. To prevent the *victim OS* from modifying or removing these pages, the rootkit must use a stage 2 page table. This stage 2 page table then contains the entire physical address space, except for the pages occupied by the rootkit. However, as the *victim OS* is unaware that these pages have been repurposed, it might still try to use them. The rootkit must therefore handle these accesses appropriately. The rootkit can back virtual pages with identical contents with only one physical page, freeing the duplicates for the rootkit. This is similar to the well-established *Kernel Samepage Merging* [7]. Accesses to these pages do not trap and thus perform at native speed; however, the unexpected side-effects of the duplicity of the page may lead to confusion or a crash of the *victim OS*.

Another alternative is to leave its own pages unmapped in the stage 2 page table. This would lead to a stage 2 data abort, which transfers control to the rootkit. The rootkit could now return fake data to the *victim OS* on a read operation and ignore write operations to these pages. Accesses to these pages are vastly reduced in performance, and a write test would reveal the fake. However, timing effects can be hidden and this method could be implemented with minimum complexity.

5.2 Evading Detection

A sufficiently sophisticated rootkit scanner running in EL1 could detect a rootkit in EL2 in a number of ways. In this section, we discuss the approaches we could employ to obfuscate the rootkit and hide it from a scanner.

Performance Counters. The ARM performance counters [4,5] can be programmed to specifically count instructions executed in EL2 which would reveal the presence of a rootkit. However, the ARM architecture allows EL2 to trap all coprocessor instructions, among them the performance counters. To hide its presence, the rootkit simply has to trap and emulate the sensitive performance monitor registers and provide unsuspicious response values. Thus the *victim OS* would still be able to use the performance monitor infrastructure, but the presence of the rootkit would not be revealed.

DMA. Some peripherals have the ability to access memory directly (DMA). A suspecting *victim OS* could reprogram hardware peripherals to directly write to any physical address, effectively bypassing the stage 2 translation. Such a mechanism threatens the rootkit. On hardware platforms that contain an ARM System Memory Management Unit (SMMU [18]), the rootkit could easily prevent DMA access to its own pages. It would do so by preventing the *victim OS* to manage the SMMU, emulating SMMU accesses by producing fake responses, and then programming the SMMU to restrict DMA access to those pages still available to the *victim OS*. On hardware platforms without an SMMU, the rootkit would have to emulate every DMA-capable device – third-party DMA controllers as well as first-party DMA devices, e.g. SD/MMC controllers – to prevent its memory from being disclosed or overwritten.

System Emulation. Many system control interfaces on ARM platforms are memory mapped. For example, the interrupt controller interface exposes the current interrupt configuration state. The *victim OS* could use this to look for discrepancies to its own expected interrupt state. It could thus discover the EL2 timer, which the rootkit might employ for its periodic execution. In order to hide these activities, accesses to the interrupt controller have to be emulated.

Time. As described before, some system control interfaces and peripherals have to be emulated by the rootkit. However, the increased access latencies due to emulation can be measured by a scanner in EL1. To prevent this, the rootkit has to present a virtualized timer to the *victim OS*. Newer versions of the Linux kernel already use the ARM EL1 virtual timer interface. This allows the rootkit to transparently warp the time for the *victim OS*.

In case the *victim OS* uses the EL1 physical timer, the rootkit can trap all accesses to these timer registers and emulate the "time warp" by reporting lower values. If auxiliary timers (like additional ARM SP804 [2] peripherals) exist on the system, the rootkit has to emulate accesses to those as well. Since the *victim OS* has no access to an independent clock source on the system, it cannot reliably determine how much wall time has passed since its last measurement. The only chance for a scanner to detect the rootkit is with the reference of an external time source. To reveal the presence of a rootkit in EL2 using an external time source, a sophisticated scanner could induce a large amount of traps into EL2. This could be done by accessing coprocessor registers which are emulated by the rootkit. Also, the *victim OS* could trigger large amounts of interrupts that have

to be handled by the rootkit. Due to the time warping the discrepancy between the local time and the external time grows with each entry in EL2. Section 8 discusses the effectiveness of such a scanner in more detail.

Cache and TLB. ARM allows SoC designers to use several levels of caches, but most common are just two levels, a dedicated L1 cache for each core and a L2 cache shared among all cores. Since the cache is shared between all privilege levels, a scanner could notice a performance slow down because not all cache lines are available to the *victim OS*.

As described in Sect. 5.1, the rootkit would want to use a stage 2 page table to prevent the *victim OS* from accessing the memory pages of the rootkit. The stage 2 page table translations are cached in a dedicated part of the TLB (Translation Lookaside Buffer), the IPA (Intermediate Physical Address) cache. The IPA cache is transparent and fetches translation just like the normal TLB (for stage 1 translations), but only for stage 2 page table translations. Thus, a scanner could exploit this fact and try to measure artifacts originating from IPA cache hits or misses.

5.3 Availability

To perform its malicious tasks, a rootkit must gain control. A rootkit can run in two different modes of operation, which we have termed *proactive* and *reactive*. Whether a rootkit operates in *reactive* or *proactive* mode has implications on detectability, runtime and implementation complexity.

Proactive execution requires a time source to periodically gain control. A periodic timer interrupt that is routed to EL2 can be configured in such a way that the rootkit can perform its malicious operation. ARM's interrupt controller, however, does not provide a mechanism to selectively route interrupts to EL1 or EL2. Therefore, in the *proactive* model, all interrupts have to be intercepted by the rootkit. The rootkit then has to filter out its EL2 timer events and deliver all other interrupts to the *victim OS*. This approach is more complex to implement and increases interrupt latency. However, it is perfectly suited for data exfiltration attacks where keystrokes or other user actions are monitored during phases of platform activity and later transmitted to an external command-and-control entity when the platform would otherwise be idle.

Reactive execution is a less invasive approach because the rootkit would only react to certain stimuli from within the *victim OS*. Inside the *victim OS*, the adversary would want to run an unsuspiciously looking program in EL0 (without any specific user permissions) that communicates directly with the rootkit in EL2. However, most traps that can be configured to target EL2 can only originate in EL1 (and not EL0), e.g. the `hvc` instruction. Execution of such an instruction in EL0 is considered undefined and would simply be reported to EL1. This makes it difficult for a program running in EL0 to communicate directly with the rootkit in EL2, without notifying the *victim OS* in EL1.

One of the few exceptions are the deprecated Jazelle[3] instructions. These instructions can be executed in EL0 and directly trap into EL2. ARMv7 mandates that any system implementing the ARM virtualization extensions must provide an empty Jazelle implementation. This implementation only includes a few Jazelle control registers and the `bxj` instruction. The specification also mandates that this `bxj` instruction must behave exactly like a `bx` instruction. Even though the Jazelle implementation is not fully implemented anymore, the *HSTR* register still provides the option to trap accesses to Jazelle functionality to EL2. Thus, in the *reactive* execution mode, the rootkit would enable trapping of the Jazelle instructions into EL2. Now an EL0 application is able to trigger EL2 traps by executing a `bxj` instruction without notifying the *victim OS* in EL1.

The *reactive* approach is much easier to implement than the *proactive* model, and it has almost zero overhead during regular system activity. However, it is more suited for externally triggered attacks. For example, an unsuspicious application with network connectivity could allow an adversary to invade the platform, quickly elevate his privileges by activating the rookit, steal sensitive pieces of information, and deprivilege itself again all by signalling the rootkit with the previously described Jazelle functionality.

6 EL2 Rootkit Implementation

Based on the previously defined requirements on *resilience*, *detectability* and *availability*, we implemented a rootkit, which we called *rHV*. Of course, a real attack would comprise the transition from EL0 to EL1 first, which would rely on a real vulnerability in the Linux kernel. For simplicity, we implemented a kernel module to load our *rHV* code directly into EL1. The kernel module provides a device node where we supply our *rHV* binary, along with a number to signal the kernel module which attack vector to use. The kernel module then exploits the specified attack vector to deploy *rHV*.

Once *rHV* is deployed, its execution is split into two parts. The *initialization phase* starts immediately when *rHV* is loaded. Depending on the attack vector, the *initialization phase* starts in secure EL1 (Attack vector 3) or directly in EL2 (Attack vectors 1 and 2). After the initialization phase, *rHV* enters *runtime phase*, where *rHV* provides its malicious service.

Initialization Phase. In the initialization phase, *rHV* checks whether the processor's current security state is secure. If this is the case, *Attack vector 3 - Migrate Linux to non-secure* (see Sect. 4) is used. To do this, *rHV* copies a number of registers from the secure to their non-secure counterpart. Additionally, the interrupt controller is configured in such a way that all interrupts are routed to the non-secure world.

[3] Jazelle is a special processor instruction set for native execution of Java bytecode found in earlier ARM cores.

As discussed in Sect. 5.1, for a rootkit to be *resilient*, it has to control accesses to main memory. For *rHV*, we decided to stick with the solution to trap accesses to pages which are occupied by *rHV* itself and emulate read/write accesses to these pages. So the second step of the initialization code, after the migration is finished, is to setup a stage 2 page table. In our experiments, we tested different stage 2 page table layouts (from a single 1 GByte mapping entry for the entire address space up to 4 KBytes entries) to verify the impact through the stage 2 page table and the IPA cache on the overall system performance. Results with the different stage 2 page table layouts are provided in Sect. 8. Once the stage 2 page table is constructed and activated, *rHV* jumps to the third step of the initialization phase.

As already described in Sect. 5.2, certain performance monitoring registers can be configured to reveal the presence of *rHV*. Therefore, we configured the hardware so that all accesses to these registers trap into EL2. We implemented emulation code to reflect the real values of the registers but ignored write accesses to them.

We implemented two versions of *rHV*, one for the *reactive* mode and one for the *proactive* mode, because the mode has influence on the layout of the stage 2 page table. In *reactive* mode, *rHV* does not need access to the interrupt controller, so it can just forward the interfaces to the *victim OS*. No additional entries in the stage 2 page table are necessary. In *proactive* mode, however, *rHV* has to handle timer interrupts. Thus, accesses from the *victim OS* to the interrupt controller must be prevented. Instead, the stage 2 page table gets an entry for a virtual interrupt controller interface, which is presented to the *victim OS*. Finally, *rHV* also enables the EL2 timer to gain periodic control.

Runtime Phase. As discussed in Sect. 5.3, we implemented both modes of operation *reactive* and *proactive*. The implications on the overall system performance based on the execution mode are provided in Sect. 8.

Independent from the fact whether *rHV* runs in *proactive* or *reactive* mode, a number of operations need to be done. First, the cycles the CPU spends in EL2 mode must not be visible to the *victim OS*. Recent versions of the Linux kernel already use the virtual timer infrastructure, which makes it easy to warp the time for the *victim OS*. *rHV* warps the guest timer in the following manner: upon each entry into EL2, the current time value is saved. Upon exiting EL2, the *rHV* again reads the current time value. The difference of these values is then stored in the appropriate offset register (`CNTVOFF`). The ARM virtualized timer infrastructure automatically subtracts the value of this offset register whenever the *victim OS* reads its ("virtual") time. Thus, the time spent in EL2 mode is no longer detectable from EL1.

In addition to the time warping, which is necessary in both modes of operation, in *proactive* mode, *rHV* also has to handle interrupts. In order to use a dedicated timer for EL2, all interrupts must be trapped into EL2. Upon each interrupt, *rHV* checks whether the interrupt originated from the EL2 timer or not. In the latter case, the interrupt is simply forwarded to the *victim OS*; otherwise *rHV* handles the interrupt itself and performs its malicious operation. Afterwards, execution is resumed in the *victim OS*.

7 Malicious Service

Once *rHV* has taken control of EL2, it can provide services to adversary-controllable EL0 services. We implemented a simple privilege escalation service as a proof-of-concept to show the feasibility of the approach. However, more sophisticated services are conceivable as *rHV*'s power over the system is unconfined.

Our malicious service consists of a combination of code that executes in EL2 (residing in *rHV*) and a malicious app running in EL0. The idea is that the malicious app, which executes with normal user privileges, requests elevation to root, performs modifications to the installed OS or extracts sensitive information, and is deprivileged again. In the *proactive* case, *rHV* uses the dedicated EL2 timer to gain control and uses this timeslot to check whether the malicious EL0 process is currently running. We use the kernel structure reconstruction method to search for the `task_struct` of our malicious app. When its presence is detected by comparing certain identifiers, e.g., the process name, the *rHV* replaces the UID of the task with zero, thus granting root privileges.

A process that suddenly executes with root privileges might raise suspicion. A component in EL1 might recognize this change; therefore, the *rHV* resets the UID to its original value when the next interrupt occurs. With this mechanism, we make sure that no reschedule happens while the malicious process is executing.

In the *reactive model*, *rHV* is invoked with the `bxj` instruction. If the general purpose registers contain the correct magic values, *rHV* uses the same method as mentioned before to find the `task_struct` and then sets the UID to zero. Otherwise, the instruction is simply emulated as `bx`.

8 Evaluation and Countermeasures

The effectiveness of any rootkit heavily depends on the stealthiness. As described in Sect. 5, some transitions from EL1 into EL2 are inevitable. Thus, in this section, we evaluate how long certain operations take and discuss the effectiveness of scanners trying to detect the presence of *rHV*. We also analyze the overhead of the two-stage MMU translation process. All tests were conducted on a Cubieboard 2 [1].

A rootkit scanner could try to uncover *rHV* through the induced performance overhead (e.g., when *rHV* runs *proactive* all interrupts cause the CPU to trap into EL2). Also the 2 stage page table translations introduce overhead that a scanner could try to measure. To estimate the effectiveness of such a scanner, we performed a number of standard system benchmarks. With the lmbench [17] benchmarking suite, we measured *rHV*'s impact on these low level operations. Table 1 shows the results. Column 1 describes the performed benchmark, the other columns show the results in the respective setups. We performed each benchmark 50 times and calculated the mean values and their respective standard deviation. The mean values show a slight, but noticeable performance overhead in the *rHV* setups. However, the high standard deviation values render the mean value difference almost undetectable. In order to further analyze the

Table 1. lmbench benchmarks (results are in microseconds).

Benchmark	Linux		*rHV (proactive)*		*rHV (reactive)*	
	mean	std. dev	mean	std. dev	mean	std. dev
`lat_ctx 2`	58.1050	4.8200	59.2100	5.6957	58.8400	4.6556
`lat_ctx 4`	64.3100	4.3080	65.8950	5.3935	65.8300	4.3352
`lat_ctx 8`	66.0458	4.4091	68.2240	5.1715	67.6644	4.3407
`lat syscall`	0.2785	0.0018	0.2787	0.0007	0.2785	0.0014
`lat read`	0.6623	0.0015	0.6628	0.0015	0.6625	0.0015
`lat write`	0.4779	0.0009	0.4788	0.0010	0.4781	0.0009
`lat_pipe`	12.5509	0.6583	12.6093	0.7291	12.8524	0.8827
`lat_select`	15.7479	0.0061	15.7526	0.0074	15.7502	0.0076

detectability, we verified that the measurement results indeed follow a Gaussian distribution by running 500 iterations of the lmbench context switch benchmark. We then extrapolated Gaussian curves from the mean values and standard deviations from Table 1 and compared native against *rHV*-controlled execution (see Fig. 2).

From these graphs, it is evident that a scanner would require a vast number of measurements and exact knowledge of the distribution parameters of both clean and infected systems to reliably discern whether *rHV* is indeed running on a platform.

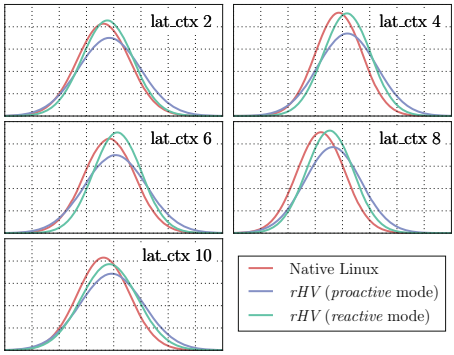

Fig. 2. Comparison of the distributions of context switch benchmarks.

Clock Drift. Another approach is to measure the clock drift that is induced by *rHV*. As described in Sect. 5.2, *rHV* hides the clock cycles that the CPU spends in EL2. In combination with an external time source, a scanner could exploit this effect to reveal the discrepancy between the local clock and the external clock source. Since the scanner can not know when *rHV* actually executes,

it would enforce traps into EL2 to reveal the clock drift. This could be done by e.g. multiple executions of a `bxj` instruction in the *reactive* setup or by utilizing a peripheral to trigger a large number of interrupts in the *proactive* setup. Figure 3 depicts the drift of the local clock compared to an external clock, e.g. NTP. Assuming an NTP accuracy of ∼5 ms over an internet connection the clock drift introduced by *rHV* becomes visible after 60.000 traps into EL2, which could be either an execution of `bxj` or an interrupt handled by *rHV*.

In both cases, a huge number of events is necessary in order to build a scanner that could reliably discern between a native and an *rHV*-infected system. Although not implemented by us, we argue that *rHV* could be retrofitted with an "alarm mechanism" that detects unusually large numbers of EL2 traps and activates appropriate countermeasures to evade detection (e.g. switching from *proactive* to *reactive* execution).

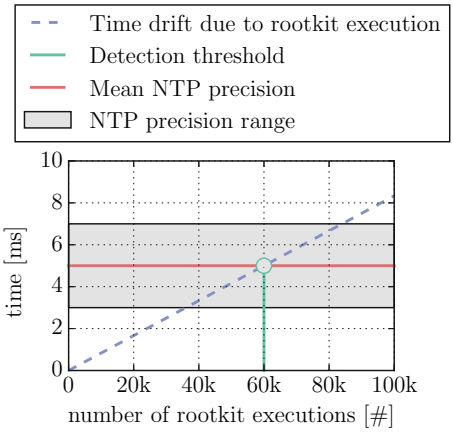

Fig. 3. Detectability of *rHV* in *reactive* execution based on time drift.

Stage 2 MMU. As mentioned in Sect. 5.2, *rHV* might employ a stage 2 page table to hide itself. This two-step translation process costs additional time, which *rHV* cannot hide, as there is (quite intentionally) no trap. In order to illustrate and quantify this effect, we built a separate bare-metal setup. We chose two sets of sixteen memory locations each, one of them pathological to the TLB due to its limited associativity on the Cortex-A7. These sixteen locations are then accessed in a tight loop and the total time is measured, flushing the TLB before the first iteration. The resulting graph is shown in Fig. 4.

We can observe that for low loop iteration counts, the major contributing factor is the stage 2 page table walk caused by the initial TLB flush. However we will have to assume that the IPA cache is warm when the scanner operates, so we cannot assume that this effect will be directly visible. The figure shows that

for higher loop iteration counts, the access times average out, even for a stage 2 pagetable consisting of 2 MBytes entries. If, on the other hand, the scanner knows and exerts an access pattern that requires continuous walks, the effect is indeed detectable, regardless of the current TLB state, as the "s2_2m_pat" curve shows. This pattern is however highly dependent on the stage 2 entries and requires e.g. for 2 MBytes mappings access to locations that are at least 512 MBytes apart.

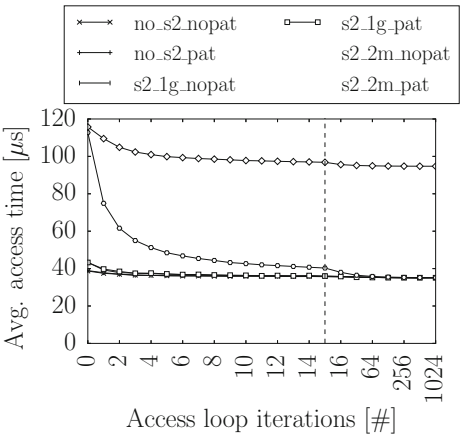

Fig. 4. Memory access times with different stage 2 parameters (s2 and no_s2), different mapping sizes (1g and 2 m mappings) and pathological and non-pathological access patterns (pat and nopat).

As this is an intrinsic effect, *rHV* could only evade this by resorting to a different hiding strategy or by employing more complex paging schemes, e.g. adaptive mechanisms as discussed by Wang et al. in [24].

9 Attack Vector Prevention

In this section, we describe several ways to prevent *rHV* or any other malware from occupying EL2. In general, the countermeasures are different depending on whether our attack should be prevented in an already deployed system or if it is possible to replace components with recompiled versions.

If the user has full control over the platform, and the system firmware boots into secure EL1, the `SCR.HCE` could be set to zero, disabling the `hvc` instruction. The bootloader could also directly switch to EL1 mode in the non-secure world, giving Linux no chance to install its hyp stub vectors into EL2. This way, EL2 is sealed and cannot be entered directly anymore. However, these fixes require changes to the boot chain, which is usually under vendor control. Additionally, boot software would have to know whether virtualization extension lockdown is

desired, which requires development of an appropriate mechanism, e.g., a flag in the EL1 OS image, or a runtime EL3 service which irrevocably disables EL2 until reset.

If EL2 is still unsealed when EL3 (or secure EL1 on ARMv7) has been left and the general purpose OS starts up, it is complicated to get EL2 sealed. It is remarkable that EL2 itself offers no way to disable `hvc` functionality. In these cases where disabling EL2 is impossible, the best bet is to make it unusable.

One approach might be to set `VBAR_EL2` to an invalid location. As `hvc` itself cannot be enabled, this still leaves an opportunity for a denial of service attack, as execution of that instruction would then lead to an endless exception loop. An improved attempt might thus install a "nop" vector table, which just executes `eret` (exception return) for every EL2 trap it receives. This strategy suffers from the problem that the vector table has to be located in physical memory, and all of physical memory is accessible to EL1. Thus an adversary could again find the location of this vector table, overwrite its entries, and gain EL2 control again.

Finally, the defender could create a stage 2 page table of his own to protect his lockdown EL2 vector table from being manipulated from EL1. Accesses to this range would then either result in stage 2 page faults, which the lockdown hypervisor could reflect to EL1, or could be backed by invalid physical addresses or emulated so that EL1 just sees garbage data.

The Linux hyp stub was added to the kernel soon after Linux kernel release v3.6. Many Android devices still run kernel versions lower then v3.6 (e.g. v3.0 or v3.4). These devices then have a completely uninitialized EL2 mode. To prevent an adversary from exploiting this entry (Sect. 4), an administrator or a user can seal EL2 as described for attack vector 1.

10 Related Work

Soon after Intel and AMD released their respective virtualization extensions King et al. [16] proposed a new form of malware that resides in a virtual machine. The suggested malware, dubbed VMBR (Virtual Machine Based Rootkit), runs in a VM on top of an existing hypervisor, such as Virtual PC or VMWare Workstation. With the help of the underlying hypervisor, they implemented a number of malicious services to spy on a victim OS. In the same year, Rutkowska [19] set out the *Bluepill* concept. Her attack leverages AMD's virtualization extension to move the operating system into a virtual machine *on-the-fly*. In 2008, Wojtczuk and Rutkowska [20,25] also showed how to attack Xen using different DMA capable devices (e.g. network card), which are controlled by the privileged domain Dom0. They use these devices from Dom0 to overwrite parts of the Xen hypervisor, installing a backdoor.

A number of mitigation techniques have been proposed to detect and prevent the subversion of system software (OS or hypervisor) on x86 based processors. Garfinkel et al. [15] discuss the detectability of hypervisors from within a guest. Based on a number of discrepancies (CPU interface, timing, resources, etc.), they argue that the prevention of detecting a hypervisor from within a guest

OS is infeasible and fundamentally in conflict with the technical limitations of virtualized platforms. In the same year, Franklin et al. [14] proposed a technique to detect the presence of a hypervisor on a target system. Their approach exploits hypervisor timing dependencies to elicit measurable hypervisor overhead.

On the ARM architecture, David et al. [12] and Zhang et al. [27] proposed hardware-assisted rootkits. Both leverage architectural features to hide their rootkits. The Cloaker rootkit [12] uses an alternative `VBAR` address to regularly gain control. On every exception that traps into a privileged mode, the rootkit gains control. As soon as the rootkit has performed its malicious task, it jumps to the original exception vector to stay stealthy. The CacheKit rootkit, as described in [27], uses the ARM cache lockdown feature to solely stay in the L2 cache. The authors claim that rootkit scanners that only scan the main memory are unable to detect the rootkit.

11 Conclusion

In this paper, we showed that an adversary can gain control over the EL2 processor mode on ARM to install a highly privileged rootkit. This EL2 rootkit is very hard to detect and to remove because it has full control over all system resources and can easily spy on the OS kernel as well as user applications. In a proof-of-concept implementation, we showed what a malicious service utilizing the capabilities of *rHV* could look like. We evaluated our rootkit and showed that most of the obvious detection mechanisms (e.g. time drift, memory access times, etc.) would not work on such a EL2 rootkit. However, with the IPA cache we identified a unique and reliable a way to detect it nevertheless. We also discussed a number of mechanisms to seal the EL2 mode and to prevent malicious software from installing itself into EL2 entirely.

We believe that EL2 should be sealed if an operating system does not intend to make use of the virtualization capabilities of the device. This ensures that no malware can gain higher privileges than the OS kernel itself and thus escape detection.

References

1. Cubieboard 2. http://cubieboard.org/model/cb2/. Accessed 06 May 2015
2. ARM Dual-Timer Module (SP804). Technical Reference Manual, January 2004
3. ARM Security Technology - Building a Secure System using TrustZone Technology. Whitepaper, April 2009
4. ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition. Whitepaper, July 2012
5. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. Whitepaper, July 2014
6. Alves, T., Felton, D.: Trustzone: integrated hardware and software security-enabling trusted computing in embedded systems. White paper, arm, July 2004
7. Arcangeli, A., Eidus, I., Wright, C.: Increasing memory density by using KSM. In: Proceedings of the Linux Symposium, pp. 19–28. Citeseer (2009)

8. Coppola, M.: Suterusu rootkit: inline kernel function hooking on x86 and arm, January 2013. http://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/. Accessed 08 Mar 2016
9. CVE, Details: The ultimate security vulnerabilty datasource: Linux Kernel: Vulnerability Statistics, March 2016. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. Accessed 29 Mar 2016
10. Dall, C., Nieh, J.: KVM/ARM: Experiences Building the Linux ARM Hypervisor (2013)
11. Dall, C., Nieh, J.: KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor, pp. 333–347 (2014)
12. David, F.M., Chan, E.M., Carlyle, J.C., Campbell, R.H.: Cloaker: hardware supported rootkit concealment. In: IEEE Symposium on Security and Privacy, SP 2008, pp. 296–310. IEEE (2008)
13. Franklin, J., Luk, M., McCune, J.M.: Detecting the presence of a vmm through side-effect analysis 15–712 project final report. Selected Project Reports, Fall 2005 Advanced OS & Distributed Systems (15–712), p. 7 (2005)
14. Franklin, J., Luk, M., McCune, J.M., Seshadri, A., Perrig, A., van Doorn, L.: Towards sound detection of virtual machines. In: Botnet Detection, pp. 89–116. Springer, Heidelberg (2008)
15. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: Vmm detection myths and realities. In: HotOS (2007)
16. King, S.T., Chen, P.M.: SubVirt: implementing malware with virtual machines. In: 2006 IEEE Symposium on Security and Privacy, p. 14. IEEE (2006)
17. McVoy, L.W., Staelin, C., et al.: lmbench: portable tools for performance analysis. In: USENIX Annual Technical Conference, San Diego, CA, USA, pp. 279–294 (1996)
18. Mijat, R., Nightingale, A.: Virtualization is coming to a platform near you. ARM White Paper (2011)
19. Rutkowska, J.: Introducing blue pill. The official blog of the invisiblethings. org 22 (2006)
20. Rutkowska, J., Tereshkin, A.: Bluepilling the xen hypervisor. Black Hat USA (2008)
21. Sharifi, M., Salimi, H., Saberi, A., Gharibshah, J.: VMM detection using privilege rings and benchmark execution times. Int. J. Commun. Netw. Distrib. Syst. **11**(3), 310–326 (2013)
22. trimpsyw,: adore-ng - linux rootkit adapted for 2.6 and 3.x, October 2014. https://github.com/trimpsyw/adore-ng. Accessed 08 Mar 2016
23. unixfreaxjp,: MMD-0028-2014 - Fuzzy reversing a new China ELF Linux/XOR. DDoS, September 2014. http://blog.malwaremustdie.org/2014/09/mmd-0028-2014-fuzzy-reversing-new-china.html. Accessed 08 Mar 2016
24. Wang, X., Zang, J., Wang, Z., Luo, Y., Li, X.: Selective hardware/software memory virtualization. ACM SIGPLAN Not. **46**(7), 217–226 (2011)
25. Wojtczuk, R.: Subverting the xen hypervisor. Citeseer (2008)
26. Xiao, J., Lu, L., Huang, H., Wang, H.: Hyperprobe: towards virtual machine extrospection. In: 29th LISA, pp. 1–12. USENIX Association (2015)
27. Zhang, N., Sun, H., Sun, K., Lou, W., Hou, Y.T.: Cachekit: evading memory introspection using cache incoherence. In: 2016 IEEE European Symposium on Security and Privacy. IEEE (2016)