SelMon : Reinforcing Mobile Device Security with Self-protected Trust Anchor

Jinsoo Jang
jisjang@cnu.ac.kr
Chungnam National University
Daejeon, Republic of Korea

Brent Byunghoon Kang
brentkang@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

ABSTRACT

Higher privileged trust anchors such as thin hypervisors and TrustZone have been adopted to protect mobile OSs. For instance, the Samsung Knox security platform implements a kernel integrity monitor based on a hardware-assisted virtualization technique for 64-bit devices. Although it protects the OS kernel integrity, the monitoring platform itself can be a target of attackers if it encompasses exploitable bugs. In this paper, we propose SelMon, a portable way of self-protecting kernel integrity monitors without introducing another higher privileged trust anchor. To this end, we first logically separate the regions of the integrity monitor into two parts: privileged and non-privileged regions. Then, we ensure that only the privileged region code can access the critical data objects that can be exploited to compromise the monitor integrity (e.g., the hypervisor page table). The non-critical operations in terms of preserving the monitor integrity are conducted in the non-privileged region. In addition to the privilege separation, we also illustrate how to utilize the general hardware features, watchpoint and data execution prevention (DEP), to ensure the robustness of the separation. In the evaluation, it was found that our approach imposes a negligible overhead of 2% in the worst case with SPEC CPU2006.

CCS CONCEPTS

• **Security and privacy → Mobile platform security**; **Trusted computing**; *Virtualization and security*.

KEYWORDS

Mobile device security, Privilege separation, Debug facility

ACM Reference Format:
Jinsoo Jang and Brent Byunghoon Kang. 2020. SelMon : Reinforcing Mobile Device Security with Self-protected Trust Anchor. In *The 18th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '20), June 15–19, 2020, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3386901.3389023

MobiSys '20, June 15–19, 2020, Toronto, ON, Canada

ACM ISBN 978-1-4503-7954-0/20/06…$15.00
https://doi.org/10.1145/3386901.3389023

1 INTRODUCTION

Advancements in the mobile processor technology have led to introducing various options for the development of security applications. Specifically, supporting privileged hardware features such as hardware-assisted hypervisors and TrustZone enables building trust anchor-based security facilities. For instance, the industry and academia utilized TrustZone to protect the mobile OS kernel integrity [2, 16, 30] on 32-bit mobile devices. On the 64-bit architecture, the Samsung Knox platform leverages the hardware-assisted hypervisor as a trust anchor to ensure kernel integrity [12]. Hundreds of millions of manufactured mobile devices are running with these security artifacts.

Unfortunately, researchers have demonstrated that the trust anchor itself is not free from attacks [3, 4, 12]. Regardless of whether the hypervisor or TrustZone is used, the kernel integrity protection is conducted in a similar way in which the OS is deprivileged and security critical operations of the OS (e.g., update of system register and page table) are trapped, verified, and emulated by the trust anchors. This delegation of OS kernel operations creates some channels between the OS and the trust anchor for contiguous interaction between them, which are abused by attackers to find and exploit vulnerabilities of the trust anchors.

To address this problem, we propose a trust anchor self-protection mechanism, called SelMon, which does not need another higher privileged components for trust anchor protection. A thin hypervisor-based kernel integrity monitor is enhanced with SelMon to show the feasibility of our approach. The virtual separation of monitor privilege and the exploitation of general hardware features lay at the heart of SelMon. The monitor is logically separated into privileged and non-privileged regions. The non-privileged region performs non-critical and primary operations for monitoring the OS kernel, such as updating kernel system registers. On the other hand, the privileged region manages predefined critical operations closely related to the monitor security. For example, page tables can be abused to manipulate the monitor; thus, a page table update is defined as one of the security-critical operations restrictively conducted in the privileged region.

For proper protection, the privileged region should be neither accessible nor executable while the non-privileged region is activated. For the non-accessibility enforcement, we utilize the hardware debugging feature, watchpoint. Specifically, at the entry to the non-privileged region, we enable watchpoint monitoring to the entire privileged region so that any non-privileged code cannot maliciously modify it. To prevent an arbitrary execution of the privileged region code, we exploit the hardware support for data execution prevention (DEP). In particular, we map the privileged code to the writable pages and enable the DEP when a switch to

the non-privileged region occurs. Owing to the non-accessibility guaranteed by the watchpoint configuration, this region remains immutable during the non-privileged code execution. Finally, switches between the regions should be managed in a deterministic way. To this end, we designed entry gates to each region, which handle the aforementioned configuration based on region switches.

Previous works such as SKEE [17] and the nested kernel [27] have also introduced ways for partitioning regions with the same privilege (i.e., intra-region privilege separation). The nested kernel realizes the separation by using the write protect (WP) bit in CR0 register on x86. The WP-bit is disabled only when the privileged part is executed so that it can update the write-protected objects. SKEE utilizes the system registers for the virtualization configuration. For example, on the 64-bit ARM architecture, the translation table base register (TTBR) is updated with a new table that does not map the privileged region when the non-privileged region is activated. Unfortunately, those approaches are not effective for protecting the privileged trust anchors running on a 64-bit ARM architecture. The system-wide configuration that turns off the write-protection (CR0.WP) is not supported on ARM. In addition, SKEE requires a secondary paging support, which is not available in privileged modes such as the hypervisor (EL2 in ARM terminology) and secure kernel (TrustZone).

We implemented SelMon on an ARM Juno development board furnishing Cortex-A57 and Cortex-A53 multicore processors and 8GB DRAM. The thin hypervisor that conducts OS kernel integrity protection is hardened with SelMon to show the feasibility of our approach. The hardware-supported DEP and one of four debug watchpoints are used to isolate the privileged from the non-privileged region. SelMon degrades the hypervisor performance due to the privilege separation of the hypervisor and the hardware feature configuration operations that are conducted whenever region switches occur. However, our Phoronix benchmark results, which measure and compare the performance of Linux running on the original and SelMon-hardened hypervisors, indicate that the performance impact of SelMon is limited; a maximum overhead of 5% was observed.

Our contribution can be summarized as follows:

- We introduce SelMon, which aims to protect the trust anchor (privileged software) running on high-end mobile devices. Our approach does not require higher privileged software or hardware components other than the general hardware features such as the watchpoint and DEP.
- SelMon is scalable. Because the watchpoint and DEP are supported in kernel mode, SelMon can be adapted to self-protect the OS when the trust anchor is not available. In addition, any other architecture that supports similar hardware features can benefit from the approach of SelMon.

2 BACKGROUND AND RELATED WORK

2.1 ARM processor security state

The ARM security extension [6] enables dividing the processor security state into secure and non-secure states. In the secure state, the user, kernel, and monitor modes are available. In the non-secure state, user, kernel, and hypervisor modes are supported. The security state is configured by using the non-secure (NS) flag in the secure configuration register (SCR_EL3). For example, if the NS flag is set, the current security state is in non-secure. The configuration is performed in the monitor mode, which is introduced to handle the switches between the secure and non-secure states. In general, this processor security state separation is leveraged together with TrustZone technologies [7] that isolate critical memory region and peripherals, which enables to create the trusted execution environment (TEE).

Table 1: Example configuration for watchpoint exception generation in non-secure and secure states.

SSC	PAC	Security state	Watchpoint for
01	10	Non-secure	User
01	01	Non-secure	Kernel
11	00	Non-secure	Hypervisor
10	10	Secure	User
10	01	Secure	Kernel

Table 2: Control flag setting for debug exception routing.

	Privilege level where debug exception is generated		
	User	Kernel	Hypervisor
Settings	**Privilege level that handles the exception**		
NS=1, TDE=0	Kernel	Kernel	Hypervisor*
NS=1, TDE=1	Hypervisor	Hypervisor	Hypervisor
NS=0, TDE=×	Kernel	Kernel	N/A

* Breakpoint instruction exception only

2.2 ARM hardware-assisted virtualization

From ARMv7 [1] onward, hardware-assisted virtualization is supported. The key features of the virtualization in terms of building the security application can be classified into two: (1) secondary paging [5, 8, 14] and (2) hypervisor trap. The secondary paging essentially aims to translate the intermediate physical address (i.e., physical address in the view of the virtual machine) into a real machine address. This feature is mainly used to isolate or protect a certain memory region from other regions by crafting secondary page tables. For example, the hypervisor-based kernel integrity monitors [12, 36, 39] protect the kernel text region by setting the read-only permission of the secondary pages that map the text. The hypervisor trap configuration facilitates monitoring security-critical operations that need to be verified or emulated by the hypervisor [40, 44]. The virtual memory configuration in the OS kernel is a typical example operation that is trapped and verified by the security hypervisor.

2.3 Hardware debugging support

In addition to debugging through the external hardware (e.g., JTAG debugger), commercial processors such as ARM and x86 support hardware debugging features that enable the privileged software to trap and handle the debugging exceptions. For instance, by setting a hardware breakpoint on a particular address, a breakpoint exception can be generated when the instruction on the address is executed. The exception is trapped by the privileged software such as an OS kernel. On the other hand, the watchpoints can be

used in a similar way to monitor the data access. Specifically, when any data access (read or write) to a watchpoint-monitored region occurs, a watchpoint exception is generated, and is trapped and handled by the privileged software. In our work, we use the watchpoint to monitor access to the security-critical memory region. The following properties of watchpoints are leveraged.

Watchpoint exception generation. We can set the watchpoint exception to be generated in a certain processor mode by configuring watchpoint control flags. For example, as can be observed in Table 1, we can generate the exception in hypervisor mode by setting the security state control (SSC) and the privileged access control (PAC) flags in the debug watchpoint control register (DBGWCR) to 0b11 and 0b00, respectively. Depending on the SSC value, the watchpoint exception can be generated in the TEE (secure state) as well; setting the SSC value to 0b10 enables generating the exceptions in user mode or kernel mode in the TEE.

Watchpoint exception routing. As presented in Table 2, when the processor mode is in a non-secure state, the watchpoint exception can be handled by the kernel or hypervisor, depending on the trap debug exception (TDE) flag in the monitor debug configuration register (MDCR_EL2). For example, a TDE flag should be set to trap the watchpoint exception in the hypervisor mode. On the other hand, when the processor is in the TEE (secure state, i.e., NS=0), the exception is always trapped and handled by the kernel mode (secure OS).

Watchpoint configuration requirement. On the ARM 64-bit architecture (ARMv8), the maximum size of watchpoint monitoring is 2 GB. The size must be aligned to a power of 2. We can configure the monitoring size by using the mask flag in DBGWCR. Moreover, the monitoring start address should be aligned to the monitoring size. For example, if the monitoring size is 4 KB, the starting address of the monitoring should be aligned to 4 KB. The starting address can be configured in the debug watchpoint value register (DBGWVR).

In addition to the size and address configuration, several control flags need to be set to activate the watchpoint monitoring. First, the enable flag in DBGWCR needs to be set. Second, the kernel debug enable (KDE) flag should be set to generate the watchpoint exception in the same mode that handles the exception. For instance, for the self-management of the watchpoint exception in the hypervisor mode, the KDE and TDE flags should be set. Finally, the debug exception mask flag (D) in the process state (PSTATE) must be cleared. *This flag is automatically set when any exception occurs, disabling the debug exception generation.*

2.4 Kernel integrity monitor

Trust anchors such as TrustZone and hypervisor have been used to implement kernel integrity monitors. TZ-RKP [16] and Sprobes [30] locate the integrity monitor in the TrustZone-based TEE on 32-bit devices. In state-of-the-art mobile devices, which are based on the 64-bit ARM architecture, the hardware-assisted virtualization technique is also leveraged to enable monitoring. Samsung utilizes a thin hypervisor to protect the kernel text and data [12].

Meanwhile, regardless of where the integrity monitor is deployed, the monitors commonly enforce the following security properties. First, the page tables are not allowed to be updated by the kernel; only the integrity monitor can emulate the update after security verification. This ensures memory protection. Second, the kernel cannot configure security critical system registers. Similar to the memory protection, critical system registers are updated only by the monitor. This hinders maliciously reconfiguring the system settings (e.g., disabling the memory management unit (MMU) or remapping a page table). The monitor is synchronously invoked whenever such update needs to be fulfilled. To this end, original kernel operations for the update are replaced with a trust anchor invocation such as a hypercall.

2.5 Privilege separation

Various approaches have been explored to separate the privilege of software and to safely execute critical operations. SKEE [17] and Hilps [21] utilize memory translation-related features to create an isolated execution environment in the OS kernel. The system-wide memory WP was leveraged by the nested kernel [27], Nexen [41], and HyperSafe [43] to isolate critical operations of the OS and hypervisor. Those approaches enable intra-region privilege separation but are limitedly adoptable to the trust anchor protection on the ARM architecture as explained in Section 2.5.

On the other hand, an additional layer for monitoring the critical operations is also introduced. Xen's paravirtualization [18] verifies and emulates the OS kernel's critical operations such as page table update. The Secure Virtual Architecture (SVA) [26] provides virtual instruction-based abstraction layer to monitor the OS behavior. KCoFI [24] and Virtual Ghost [25] adopt SVA to protect the control flow of OSs and to shield applications. Apparition [28] enhances Virtual Ghost to protect the shielded applications from side-channel attacks by malicious OSs. HypSec [34] redesigns a monolithic hypervisor (e.g., KVM) to isolate a small trusted corevisor from a large untrusted hostvisor. The isolation is enforced by using secondary paging and running the compartments (corevisor and hostvisor) in different privileged CPU modes, i.e., the hypervisor and kernel, respectively. We propose a different approach that effectively isolates security artifacts without adopting an additional trusted layer and thus provide a further option when the layer is not available.

2.6 Mobile device security

Various hardware features have been leveraged to implement security applications for mobile devices. To prevent a cold boot attack, Sentry [23] stores sensitive data and code in an ARM SoC instead of DRAM. ARMlock [47] utilizes the memory domain and domain access control register (DACR) to enable hardware-based fault isolation. Norax [20] enables the execute only memory (XOM) benefitting from its hardware support from ARMv8. As a security extension to ARM processors, TrustZone [7] was broadly explored to enhance the device security. For example, TLR [37] and ObCs [33] use TrustZone to provide a TEE to 3rd party developers. TrustOTP [42] protects one-time password tokens by using TrustZone. Komodo [29], TrustShadow [31], and CaSE [46] shield a part of the application based on the isolation provided by TrustZone. Finally, vTZ [32] virtualizes TrustZone to provide a TEE to individual virtual machines.

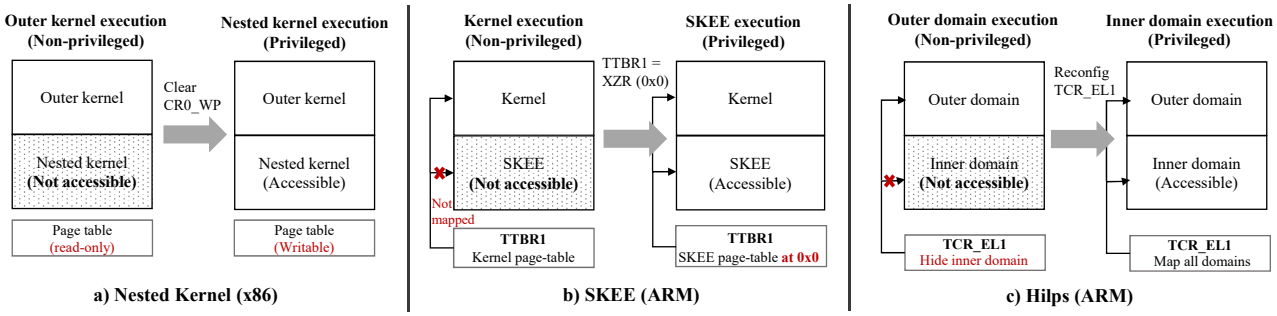

Figure 1: Nested Kernel, SKEE, and Hilps aim to isolate the critical region without using a higher-privileged trust anchor. In each approach, architecture specific features such as CR0.WP, TTBR, and TCR_EL1, respectively, are leveraged to realize the goal.

3 ATTACK MODEL

We assume the presence of a trust anchor that monitors kernel integrity and is built as a thin hypervisor. The monitor protects the OS kernel as described in Section 2.4. That is, the OS kernel is deprived of the authority of performing security-critical operations such as page table and system register updates. Instead, the monitor is in charge of verification and emulation of such operations.

However, as shown by Google researchers [12], the monitor (hypervisor) itself can be compromised by an attacker. In particular, the monitor itself could encompass vulnerabilities that accord the attacker with the ability for arbitrary memory access and control flow hijacking. With the escalated privilege, the attacker might attempt to manipulate the monitor to perpetuate the attack. To this end, critical hypervisor components such as page tables and system configuration registers will be abused. Our approach, SelMon, aims to harden the hypervisor in such a way that the integrity of hypervisor's critical components is protected even in the presence of vulnerabilities. More specific analysis and classification of the security-critical components and the design of SelMon are provided in Section 5.

4 MOTIVATION

As illustrated in Section 3, the integrity monitor itself could be vulnerable. As a primary step for enhancing the monitoring platform security, we propose SelMon, which aims to self-protect the trust anchors running in hypervisor and secure (and non-secure) kernel modes. The key technique of SelMon is realizing the privilege separation of the software that otherwise runs monolithically in the same privilege. Notably, benefiting from the hardware watchpoint and DEP support, SelMon does not depend on higher privileged hardware and software components.

Previous self-protection approaches. Before introducing SelMon, we analyze the existing approaches, namely the nested kernel [27], SKEE [17], and Hilps [21], that separate the software privilege without depending on the higher privileged components. Then, we illustrate why these approaches are not sufficient for protecting the trust anchors (privileged softwares) on modern ARM architectures.

The three approaches first separate the system into two compartments, i.e., privileged and non-privileged regions, based on whether it performs security-critical operations such as page table update. In addition, to protect the privileged region, they design specific gate code that handles the entry to and exit from the privileged region. The gates particularly manage the system configuration, which dominates the robustness of the proposed approaches. Hence, we analyze the feasibility of each approach by verifying if the gate design is generally applicable to protect the privileged software on ARM.

```
1  entry:
2  ......
3  mov %cr0,%rax              //Get current CR0 value
4  and ~CR0_WP,%rax           //Clear WP bit in copy
5  mov %rax,%cr0              //Write back to CR0
6  cli                        //Disable interrupts
7  ......
```

Listing 1: Part of the gate code for the entry to the nested kernel, which configures the WP bit in CR0.

4.1 Nested kernel

To protect critical kernel objects such as page tables, the nested kernel enforces the read-only permission for all objects. Then, to enable the privileged part to update the objects, the nested kernel utilizes the WP bit in CR0, which turns off the system-wide read-only setting. As can be seen in lines 3-5 in Listing 1, it disables the write protection at the entry to the privileged region (and vice versa at the exit from the privileged region). Unfortunately, the ARM architecture does not provide such a control bit configurability (i.e., WP). The page table attributes always take effect as long as the MMU is enabled.

```
1  ......
2  msr DAIFset,0x3            //Mask all interrupts
3  mrs x0,ttbr1_el1           //Read existing TTBR1 value
4  str x0,[sp, #-8]!          //Save existing TTBR1 value
5  msr ttbr1_el1,xzr          // Load Zero to TTBR1
6  isb
7  tlbi vmalle1               //Invalidate the TLB
8  isb
9  ......
```

Listing 2: Part of SKEE gate code that configures the TTBR value by using the zero register (xzr).

4.2 SKEE

Due to the lack of WP configuration support on ARM, SKEE uses a different approach. It prepares two different page tables for the privileged and non-privileged regions. Only the page table for the privileged region has a mapping to that region, which is activated at the entry to the privileged region. This approach, which switches the page table during region switches, has a downside in that the gate can be abused to map the malicious page table by executing the instruction that updates the TTBR. To address this problem by deterministically switching the TTBR, SKEE locates the page table for the privileged region in the constant address (0x0) and uses the zero register (XZR) in the TTBR update, as can be seen in Listing 2. If the address (0x0) is not available, the authors recommend to use the virtualization technique (i.e., the secondary paging) to remap a certain memory region to 0x0 in the view of kernel. Depending on device implementation, the address 0x0 can be reserved for hardware peripherals. For instance, the Juno development board locates Boot ROM at 0x0 [10], so it is necessary to remap the memory to activate SKEE in the Juno board. Unfortunately, the additional translation layer (i.e., the secondary paging) is not supported in the hypervisor mode and kernel mode in the secure state (i.e., the TEE kernel). As a result, the SKEE approach is difficult to adopt for the protection of software running in such privileged modes.

```
1  ......
2  1:
3  mrs x5,tcr_el1                //Read the current TCR
4  and x5,x5,#0xffffffffffdfffff //Set T1SZ flags
5  orr x5,x5,#0x400000
6  msr tcr_el1,x5                //Configure TCR
7  isb
8
9  mov x6,#0xc03f               //Check the TCR setting
10 mov x7,#0x1b                 //(1)virtual address range
11 movk x6,#0xc07f,lsl #16     //and (2)trans. granule size
12 movk x7,#0x8059,lsl #16
13 and x5,x5,x6
14 cmp x5,x7
15 b.ne 1b              //If invalid, configure TCR again
16 ......
```

Listing 3: Part of Hilps gate code that configures the translation configuration register (TCR). The flags for the virtual address range and the page size are validated.

4.3 Hilps

As illustrated in Listing 3, Hilps leverages the TxSZ field in the TCR. The range of the virtual address can be adjusted by configuring TxSZ; by dynamically changing the virtual address range, the particular virtual address region can be hidden on the fly. By using this, Hilps hides the privileged region during the non-privileged region execution. This approach could be potentially vulnerable when the attacker abuses the TCR update instruction in line 6 in Listing 3. This is because, not only the virtual address range, but also the translation granule size can be adjusted by configuring the translation granule (TGx) field in the TCR. Currently, 4 KB, 16 KB, and 64 KB translation granule sizes are configurable on the ARM 64-bit architecture. The granule size determines the index size for walking page tables as well as the minimum page size. For instance, 4 KB granule uses 9 bits slice of the virtual address as the index whereas 16 KB granule uses 11 bits; 4 KB granule allows 1 GB, 2 MB, and 4 KB page sizes but 32 MB and 16 KB pages are available with 16 KB granule depending on the level of page tables. Therefore, the page table entries and the number of the entries for walking a certain range of virtual addresses are determined by the translation granule size. That is, changing the granule size can cause undeterministic behavior because a certain virtual address can be translated by different page table entries. The attacker can exploit this to execute arbitrary and unintended instructions. For example, in Listing 3, if the validation logic (line 9-14) is located in the boundary of different 4 KB-sized page from the predecessor code (line 1-7), changing the granule size configuration from 16 KB to 4KB (or vice versa) could lead to undeterministically bypassing the validation depending on system status (e.g., cache condition and sparse physical memory allocation).

5 SYSTEM DESIGN

5.1 Overview of SelMon

SelMon splits the thin hypervisor (integrity monitor) into privileged and non-privileged regions depending on its operation criticality in terms of the monitoring platform self-protection. As described in Section 5.2, we define the hypervisor exception vector, page tables, and system control registers as the security-critical objects, which are isolated in the privileged region. On the other hand, the non-privileged region performs primary operations for the OS kernel integrity protection. We assume that this region can be vulnerable and thus can be exploited by an attacker. However, an attack triggered from the non-privileged region is effectively isolated in the non-privileged region and thus, the integrity of the hypervisor is preserved. This is because the following conditions are satisfied by SelMon. First, the non-privileged region cannot access the hypervisor-managed page tables; therefore, direct memory manipulation attempts are hampered. Second, the security-critical instructions do not exist in the non-privileged region; hence, the attacker cannot change the system configuration such as the exception vector address. Lastly, the attacker cannot reuse the critical instruction in the privileged region because the entire privileged region is enforced to be non-executable during the non-privileged region activation. The switches between regions are managed through predefined ways. We designed the secure gates that manage the entries to each region in a way that ensures security of the privileged region.

5.2 Security-critical objects

Critical objects that can be exploited by the attacker need to be properly verified and isolated. The hypervisor exception vector, page tables, and the hypervisor-related system registers are critical components that need to be protected. Note that, in our work, we consider that a thin hypervisor is deployed in the production mobile device, the operation of which is lightweight and dedicated to security. However, in general, the classified components are common and prerequisite for building privileged software including the OS kernel. Therefore, we expect that our analysis can be extended to protect more complex softwares such as the OS kernel as well as bundled hypervisors (e.g, KVM).

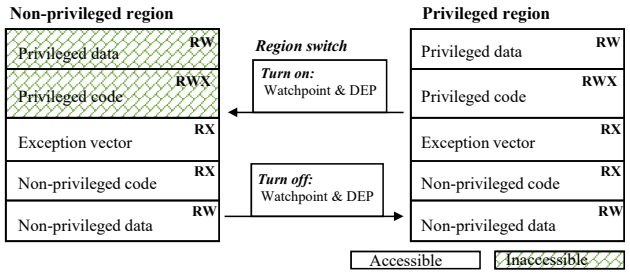

Figure 2: SelMon enables intra-region privilege separation using general hardware features such as watchpoint and DEP. For strict isolation and protection of the privileged region, switches between regions are conducted by predefined and narrow interfaces that timely configure the hardware features.

5.2.1 Page tables. There are two types of page tables that are managed by the hypervisor: hypervisor and secondary page-tables. Regardless of the types, we consider the tables as critical objects that need to be protected.

Page table for hypervisor mode. The virtualization extension supports virtual addresses in the hypervisor mode. To enable the MMU, the hypervisor needs to configure page tables and the page table base register. By configuring the page tables, the hypervisor code and data can be set as read-only. Security features such as DEP can also be enabled. However, once the attacker escalates his or her privilege to the hypervisor, the settings for protection are no longer valid. For instance, the attacker can make the hypervisor text pages writable and freely modify the code [12].

Secondary page table. As mentioned in Section 2.4, the monitor (thin hypervisor) manages the secondary page table to ensure that the OS kernel text and code are immutable (read-only). However, this page table can also be abused to break the hypervisor integrity. The attacker can manipulate the secondary page table entries such that the hypervisor memory area is mapped to the OS kernel. Doing so causes the hypervisor to be maliciously modified even from the kernel with lower privilege.

5.2.2 Hypervisor exception vector. The exception vector is a code that dispatches a handler routine corresponding to the current exception. Any exception occurrence changes the program counter to the predefined location of the exception vector. The vector has different branches depending on the exception occurrence privileges (current or lower). For example, a hypercall that is triggered from the OS kernel is trapped by the branch for the exception from a lower privilege. However, any exceptions that occur in the hypervisor mode execution are trapped by the branches for the current (hypervisor) privilege. Because the exception vector plays a role as a code dispatcher, the integrity of the vector should be protected to prevent the attacker from redirecting the control flow to the malicious code.

5.2.3 System control registers. System control registers can be targets of attacks. For example, the TTBR can be modified to map malicious page tables. The vector base address register (VBAR) can be abused to replace the current exception vector with a malicious one. Previous works [16, 30] remove all the security-critical instructions from the code region of the OS to prevent the attacker from abusing the system registers. SelMon applies a similar approach. The critical instructions exclusively exist in the privileged region. Moreover, the attacker cannot redirect the control flow to the arbitrary location in the privileged region.

5.3 Privilege separation

In this section, we discuss the logical separation of the hypervisor privilege. The separation is conducted based on the accessibility to the critical objects classified in Section 5.2.

5.3.1 Privileged region. The critical objects are isolated in the privileged region. The writable objects such as page tables are protected as privileged data. In addition, the exception vector and the privileged region code are separated based on the granularity of page, which is 4 KB. We set different page permissions to them, as shown in Figure 2. Note that although the exception vector and the privileged code are all executable, the pages for the privileged code have a writable permission whereas the exception vector page is read-only. The writable permission is given to dynamically adjust the executability of the privileged code at the entry to the non-privileged region. We discuss the access control mechanism of the privileged region in Section 5.4.

5.3.2 Non-privileged region. The non-privileged region performs general operations required for monitoring the OS kernel. As discussed in Section 2.4, the monitor needs to emulate the update of page tables and system registers for the monitored OS kernel. Similar to previous works [12, 16, 30], we implemented our integrity monitor so that system registers such as the kernel TTBR are updated by the hypervisor. Specifically, we enforce that the kernel page tables are managed by the non-privileged region of the monitor. Note that SelMon is applied to the hypervisor to highlight its feasibility and superiority in protecting the trust anchor compared to previous approaches. Thus, implementing the kernel integrity monitor is not our main contribution. However, minimal functionalities of the monitoring kernel were implemented for the performance evaluation of SelMon.

5.4 Privileged region protection

The critical operations that can be abused to compromise the hypervisor integrity are exclusively performed in the privileged region. In addition, critical data objects such as page tables are isolated in the privileged region. Thus, the privileged region should not be executable and accessible when the non-privileged region is activated.

Non-executability. To ensure that the privileged region is not executable, we exploit a hardware feature for DEP enforcement. When the writable execute never (WXN) flag in the hypervisor system control register (SCTLR_EL2) is set, all writable pages in the hypervisor regime are non-executable. To benefit from this feature, we configure the page permission of the privileged code as writable (Section 5.3.1). Hence, by setting the WXN flag at the entry to the non-privileged region, we can dynamically enforce the no-executability of the privileged region.

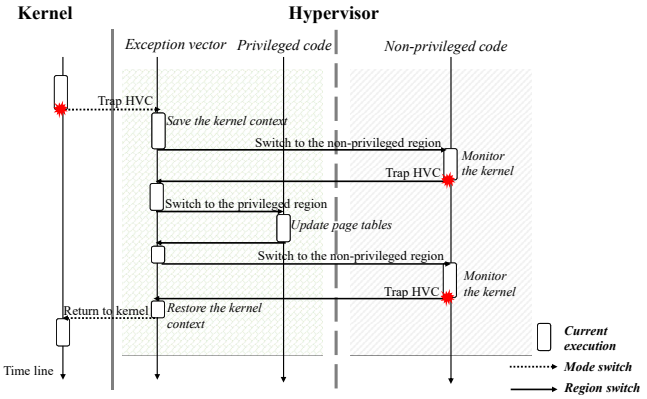

Figure 3: Mode and region switches with SelMon.

Non-accessibility. Besides the non-executability feature, we need to guarantee that the privileged region should not be accessible during the non-privileged region is activated. *By doing so, we can protect the privileged code that is mapped with a writable permission as well as critical data objects such as page tables.* To this end, we utilize the general hardware feature, the watchpoint, to monitor any read or write access to the privileged region. Specifically, we configure a watchpoint at the entry to the hypervisor so that it monitors the entire privileged region. Then, the watchpoint is activated when the region switch to the non-privileged region occurs. Note that the start address of the watchpoint monitoring should be aligned to the size of the monitoring, which must be aligned to the power of 2 (Section 2.3). To satisfy this requirement, we assign 8 MB for the privileged region and manage the start address of the region to be aligned to the 8 MB granularity.

5.5 Region switch

Figure 3 describes the flow of region switches between the privileged and non-privileged regions. Once a hypercall, which is implanted for the OS kernel monitoring as described in Section 2.4, is invoked from the OS kernel, the hypercall exception is trapped by the exception vector in the privileged region. The vector verifies the type of OS request (e.g., TTBR configuration or page table update) and dispatches a proper handler in the non-privileged region. At this point, a region switch from the privileged to the non-privileged region occurs. To isolate the privileged region as discussed in Section 5.4, we switch the region through the secure gate that configures the watchpoint and DEP flag.

On the other hand, during the non-privileged region execution, there can be a request for the security-critical operations such as the hypervisor page-table update. For example, hypervisor needs to create a mapping to the OS kernel region to update the page tables of the OS. To handle this request, a switch to the privileged region should occur. We manage this switch by exploiting the hypercall that is trapped by the exception vector in the privileged region. Details regarding each entry to the regions are described in the following sections.

```
//...(Omitted: save kernel registers)...
//Setup hypervisor trap for debug exception

L1:
mov x4,#MDCR_TDE          //Get the MDCR value
msr MDCR_EL2,x4           //Set the MDCR
isb                       //Synchronization barrier
mov x5,#MDCR_TDE          //Get the MDCR value
cmp x4,x5                 //Validate the set value
b.ne L1                   //If invalid, loop back

L2:
mov x4,#MDSCR_KDE_MDE     //Get the MDSCR value
msr MDSCR_EL1,x4          //Set the MDSCR
isb                       //Synchronization barrier
mov x5,#MDSCR_KDE_MDE     //Get the MDSCR value
cmp x4,x5                 //Validate the set value
b.ne L2                   //If invalid, loop back

L3:
mov x4,#WPVALUE           //Get the monitoring addr.
msr DBGWVR0_EL1,x4        //Set the watchpoint addr.
mov x5,#WPVALUE           //Get the monitoring addr.
cmp x4,x5                 //Validate the set value
b.ne L3                   //If invalid, loop back

L4:
mov x4,#WPCTLR_LO         //Get the control flag(low)
mov x5,#WPCTLR_HI         //Get the control flag(high)
add x4,x4,x5              //Get the full control value
msr DBGWCR0_EL1,x4        //Configure the watchpoint
mov x5,#WPCTLR_LO         //Get the control flag(low)
mov x6,#WPCTLR_HI         //Get the control flag(high)
add x5,x5,x6              //Get the full control value
cmp x4,x5                 //Validate the configured value
b.ne L4                   //If invalid, loop back
```

Listing 4: Entry to the hypervisor that configures the watchpoint-related registers.

5.5.1 Entry to the hypervisor. In addition to saving the kernel context, SelMon configures the watchpoint-related registers when the CPU mode enters the hypervisor (Listing 4). The TDE flag in MDCR_EL2 is set to route the watchpoint exception to the hypervisor. The KDE and monitor debug event (MDE) flags are set to allow the watchpoint exception generation in the hypervisor mode. Then, we set the DBGWCR and DBGWVR to monitor the entire privileged region. Because the watchpoint configuration is static, we load predefined constant values for the registers. In addition, after setting up the watchpoint registers, we check if the currently configured values are predefined values to prevent the attacker from abusing the configuration instructions to neutralize the watchpoint-based protection. *Note that even after setting those registers, the watchpoint remains deactivated because the debug (D) flag in the PSTATE is masked.* As mentioned in Section 2.3, PSTATE.D is always masked (cleared) whenever any exception occurs.

```
L5:
mov x4,#SCTLR_WXN_LO      //Get the SCTLR value
add x4,x4,#SCTLR_WXN_HI
msr SCTLR_EL2,x4          //Set the SCTLR
isb                       //Synchronization barrier

mov x5,#SCTLR_WXN_LO      //Get the SCTLR value
add x5,x5,#SCTLR_WXN_HI
cmp x4,x5                 //Validate the set value
b.ne L5                   //If invalid, loop back

tlbi ALLE2                //Invalidate the TLB
```

```
13  isb                         //Synchronization barrier
14
15  adr x4,savedPrivStack
16  str sp,[x4]                 //Save the priv stack
17  adr x4,savedNonPrivStack
18  ldr x4,[x4]
19  mov sp,x4               //Switch to the non-priv stack
20
21  msr DAIFclr,#8          //Enable the debug exception
22  b  non_priv_reg         //Jump to the non-priv region
```

Listing 5: Entry to the non-privileged region that enables the DEP and watchpoint.

5.5.2 Entry to the non-privileged region. At the entry to the non-privileged region, the privileged region is hidden. This is realized by the gate that handles the region switch to the non-privileged region (Listing 5). To enforce the no-executability of the privileged region, we set the WXN flag in SCTLR_EL2; thus, the privileged region code that is set as writable is enforced to be non-executable. After setting up this system register, we check if the configured value is equal to the predefined value, to hinder abusing the configuration instruction. Note that SCTLR_EL2 also controls the MMU so the attacker might execute this instruction to disable the MMU. To inhibit this attempt, we map the hypervisor virtual address as identical to the physical address. Hence, even if the MMU is disabled, the value verification routine that follows the SCTLR_EL2 setup is still effective. The downside of using WXN is that it requires a TLB flush. Because the hypervisor mode does not provide the address space identifier (ASID), we just flush out the entire TLB for the hypervisor. On the other hand, the ASID is available in kernel mode for both the secure and non-secure states. Hence, we expect the performance improvement of SelMon when it is applied for kernel mode protection. After enabling the WXN, we save and restore the stack for the privileged and non-privileged regions, respectively. The stacks are assigned on a per-core basis but we omit the stack pivot logic, which is based on a current CPU ID. Finally, we activate the watchpoint monitoring by clearing the debug mask bit in the PSTATE and jump to the non-privileged region.

```
1   /*********** Non-privileged region ***********/
2   hvc #REQNO                  //Hypervisor call
3
4   /************* Exception vector *************/
5   //...(Omitted: verify the current exception)...
6
7   L6:
8   mov x4,#SCTLR_NOWXN_LO      //Get the SCTLR value
9   add x4,#SCTLR_NOWXN_HI
10  msr SCTLR_EL2,x4            //Set the SCTLR
11  isb                         //Synchronization barrier
12
13  mov x5,#SCTLR_NOWXN_LO      //Get the SCTLR value
14  add x5,#SCTLR_NOWXN_HI
15  cmp x4,x5                   //Validate the set value
16  b.ne L6                     //If invalid, loop back
17
18  tlbi ALLE2                  //Invalidate TLB
19  isb                         //Synchronization barrier
20
21  adr x4,savedNonPrivStack
22  str sp,[x4]                 //Save the non-priv stack
23  adr x4,savedPrivStack
24  ldr x4,[x4]
25  mov sp,x4                   //Switch to the priv stack
26
27  b priv_reg              //Jump to the privileged region
```

Listing 6: Entry to the privileged region. The hypercall (hvc), which automatically disables the watchpoint, is trapped by the exception vector. DEP is disabled before entering the privileged region.

5.5.3 Entry to the privileged region. The entry to the privileged region requires the non-privileged code to invoke a hypercall, which is trapped by the exception vector. As described in Section 2.3, the exception disables the watchpoint monitoring by automatically setting PSTATE.D and thus makes the privileged region accessible. In addition, the vector turns off the DEP by clearing the WXN in SCTLR_EL2 to make the privileged code executable. The stack is also switched for the privileged region.

In contrast to the privileged code, the gate to the privileged region (Listing 6) is always accessible (executable) regardless of which region is activated. Hence, the attacker can attempt to directly jump in the middle of the gate code to disable the DEP. This makes the privileged region executable without invoking the hypercall. However, this malicious behavior can readily be detected by SelMon because the watchpoint is still active due to the absence of the exception. In particular, continuing to execute the gate code without disabling the watchpoint monitoring incurs a watchpoint exception that can be distinguished by investigating the exception class (EC) field in the exception syndrome register (ESR). For example, line 22 in Listing 6 causes the watchpoint exception because the `savedNonPrivStack`, which is the storage of the non-privileged region stack, is part of the privileged data monitored by the active watchpoint.

5.5.4 Exit from the hypervisor. At the exit from the hypervisor, the saved kernel context including the general registers and watchpoint configurations are restored. Because this routine is also exposed to the non-privileged region, the attacker might try to abuse the instructions that update the configurations. However, the exit routine ends with an `eret` instruction that de-privileges the CPU mode to kernel and returns to the point immediately after the hypercall invocation in kernel. Hence, the attacker also lose its hypervisor privilege.

5.6 Compatibility with Debugging Activity

The watchpoints are shared resource between different processor modes. However, SelMon does not interfere with the use of watchpoints outside of the currently protected mode because it saves and restores the previous settings at the entry to and exit from the hypervisor mode. For example, the user and kernel mode debuggers, i.e., the `gdb` [9] and `kgdb` [11], which utilize the watchpoints for setting the data breakpoints, are still available with the presence of SelMon. In addition, the dynamic reconfiguration of the DEP does not affect the security of other modes because the system control registers are banked for each mode. Debugging the privileged software such as the hypervisor and trusted firmware generally uses a hardware debugger with JTAG interface. Because the hardware debugger also configures the watchpoints to monitor data access, it is

Table 3: System registers leveraged for building SelMon.

System register	Description
DBGWVR & DBGWCR	Configure watchpoint
DAIFCLR	(Un)mask debug exception
MDCR	Configure watchpoint exception handling location
SCTLR	Configure DEP
ESR	Validate trapped exception

necessary to disable SelMon to prevent any undesirable corruption of the configuration.

6 IMPLEMENTATION

Lines of code. SelMon is sufficiently small to be manually audited or formally verified [35]. The LoC of the current implementation of the hypervisor is approximately 770 in assembly without including the kernel patch for invoking the critical operation emulation (520 LoC and 250 Loc for the privileged and non-privileged regions, respectively). Note that the operation conducted by the non-privileged region in our implementation is limited to emulating the critical OS kernel operations such as the page table and TTBR updates (this is sufficient to demonstrate the feasibility of SelMon); the additional security components described in the previous work [16] (e.g., tracking kernel memory double mapping) are not implemented. Hence, the LoC of the non-privileged region could be larger when the kernel monitor is fully implemented.

Thin hypervisor. We reserved the physical memory range from 0xFE000000 to 0xFEFFFFFF (16 MB) by modifying the Linux device tree file (`juno.dtsi`) for the hypervisor implementation. Then, we created a secondary page table that maps all physical memory ranges other than the reserved area for the hypervisor isolation. We locate the secondary page table from 0xFE000000. Once the page-table setup is completed, the control registers for enabling the secondary paging, such as virtualization translation table base register (VTTBR_EL2), the virtualization translation control register (VTCR_EL2), and hypervisor configuration register (HCR_EL2) are configured.

Kernel patches. We follow the previous approaches [16, 30] to implement the capability of kernel integrity monitoring in the hypervisor. The instructions that set up the security-critical system registers such as TTBR and VBAR are replaced with hypervisor calls (i.e., `hvc`). In addition, we insert the hypervisor calls in the page-table management functions (e.g., `set_pte` and `set_pmd`) to verify and emulate the update of the write-protected kernel page tables in the hypervisor. Note that the physical memory hosting the kernel text is set as read-only in the secondary page table so that the patches in the text are also immutable.

SelMon. To apply SelMon to the thin hypervisor, we divide the hypervisor memory (16 MB) into two halves, which have 8 MB each. The low (0xFE000000–0xFE7FFFFF) and high (0xFE800000–0xFEFFFFFF) halves are assigned to the privileged and non-privileged regions, respectively. Because the privileged region location and size are aligned to the power of 2, it satisfies the watchpoint setting requirement (Section 2.3). In our implementation, we only need one watchpoint for enabling SelMon. However, the location and size of each regions can be flexibly adjusted by using multiple watchpoints as well.

Finally, Table 3 summarizes the system registers that SelMon utilizes to realize the intra-region privilege separation. Those are defined as part of a high-end ARM processor's (i.e., ARMv8-A [6]) specification and are generally supported in production devices unless they are intentionally disabled or modified by manufacturers. Note that even if the system registers are available, we expect that the manufacturer's intervention is necessary to deploy SelMon in production devices. This is because the secure boot [15] that checks the integrity of the loaded images (e.g., secure OS and hypervisor) using the manufacturer's key at boot time is generally employed in modern devices. ARMv8-M, the low-end specification for microcontrollers, also defines debug facilities including the watchpoint. However, due to the architecture discrepancy between ARMv8-A and ARMv8-M, we expect that more design consideration will be required to build SelMon on low-end devices. We discuss this in Section 8.

7 EVALUATION

In this section, we first analyze the possible attack surfaces of SelMon and discuss how they are effectively blocked. Then, the performance overhead incurred by SelMon is measured.

7.1 Security analysis

The security of SelMon depends on the proper configuration of general hardware features (e.g., the watchpoint and DEP). The attacker's successful manipulation of the configuration could lead to bypassing SelMon. In this regard, we describe how SelMon effectively restricts the attacker from accessing the features.

Hypervisor code. Similar to the previous works [16, 17, 21, 27], we also assume that the non-privileged region code does not contain instructions that can configure the critical system registers (e.g., TTBR and VBAR). In addition, the page table and watchpoint update instructions do not exist in that region. Because the non-privileged region code is mapped with read-only permission in the hypervisor page table, the attacker cannot modify the region. The remaining option for the attacker is injecting a malicious code to the writable region and executing it; however, this attack is simply prevented because we always enable the DEP at the entry to the non-privileged region.

On the other hand, security-critical operations are performed in the privileged region. As explained in Section 5.4, the privileged region is neither accessible nor executable during the non-privileged region activation. Thus, the attacker cannot manipulate the privileged region as well as reuse the critical instructions in that region.

Mode switch. As discussed in Section 5.5.1 and 5.5.4, the watchpoint is configured at the entry to and exit from the hypervisor. Although the hypervisor entry and exit handlers are mapped as read-only, they are still executable regardless of the current region privilege. Hence, the attacker might try to abuse the watchpoint configuration instructions. Clearing one of the TDE, KDE, and MDE flags disables the watchpoint monitoring so that the privileged region can be accessible to the attacker. Maliciously reconfiguring the watchpoint address (DBGWVR) and control (DBGWCR) registers will disable the watchpoint protection as well.

At the entry to the hypervisor, the watchpoint-related registers are set to predefined values. Thus, we can verify them with constant values immediately after the configuration. When the mode switches back to kernel, the register values are restored to kernel's setting. Because the kernel settings are saved in the privileged region, the attacker cannot manipulate them. Furthermore, abusing the exit routine is not advantageous because the attacker directly loses its hypervisor privilege. That is, the last instruction, `eret`, in the exit routine switches the CPU mode to kernel.

Region switch. Between region switches, SelMon configures the DEP and debug masking (D) flag in the PSTATE. Because PSTATE.D is configured using the immediate value (#8), the attacker cannot abuse the instruction to reconfigure the flag with a general register that delivers a malicious value. Care must be taken for the DEP configuration because it requires to change the system control register (SCTLR). Similar to the watchpoint configuration protection, we verify the configured value after its being written into the SCTLR. Because other system settings such as for the MMU are controlled by the SCTLR, not only the DEP flag but also other flags need to be protected. The SCTLR value should be constant so that the verification is simple. We just compare the written value with the predefined legitimate value. Due to the timing gap between the configuration and verification, the MMU can be disabled for a while. However, the verification is still effective and deterministic because we map the virtual address to be identical to the physical address. Therefore, the attacker cannot bypass the verification.

In particular, the gate to the privileged region must be executed with the predefined interface, the hypercall, for a successful region switch. Any attempt that jumps to the gate without the hypercall invocation will generate a watchpoint exception. Finally, we do not enable interrupts during the hypervisor mode because of small set of the functionality. Thus, interrupts during the gate code execution do not need to be considered from the security perspective [27]. Even if interrupts are enabled, they will not hamper the security of SelMon because a single exception vector is used for both the privileged and non-privileged regions. We can enforce that the interrupts are securely handled by the privileged region.

Untrusted kernel. The colluding of untrusted kernel by the attacker needs to be considered. For example, the attacker can return into a malicious kernel code once the hypervisor privilege is obtained. Hence, any other region other than the hypervisor should not be executable once the hypervisor mode is activated. Eliminating this attack vector is straightforward. We just map the kernel region with a never execute (NX) permission in the hypervisor page table.

Effectiveness against real-world attacks. Because SelMon does not aim to detect or remove software vulnerability, it cannot prevent an attacker from exploiting existing vulnerabilities of trust anchors ([3, 4, 12]). However, as a baseline defense of the system, SelMon protects the integrity of the trust anchor and isolates critical operations, which greatly restricts the attacker's ability to compromise the system.

7.2 Performance

We evaluate the overhead of SelMon by measuring the hypervisor, application, and OS performance. Specifically, the performance of hypervisor operation primitives, which are required to implement the kernel integrity monitor (e.g, TTBR update), is measured. Then, we ran three benchmarks, namely SPEC CPU2006, LMBench, and Phoronix Test Suite, to measure the performance of the application and OS that run on the original and SelMon-hardened hypervisors.

Table 4: Simple operation performance of original and hardened hypervisors. Average latency (in μs) and standard deviation of 10 runs of each case of SelMon are presented.

Operation	Baseline	SelMon	Stdev	Overhead
Mode switch	3.2	4.2	0.63	1.31×
TTBR update	4.0	5.2	1.03	1.30×
Page table update	4.4	6.5	1.08	1.47×

7.3 Hypervisor

Adopting SelMon imposes an overhead due to its privilege separation and domain switches. As demonstrated in Section 5.1, we applied SelMon to the thin hypervisor that monitors the OS kernel. Then, the performance of hypervisor operations such as the update of TTBR, page table, and kernel memory is evaluated. Specifically, the times for each operation performed on the SelMon-hardened and original hypervisors are measured and compared.

As can be observed in Table 4, SelMon imposes a 31% overhead for the simple switch between the kernel and hypervisor. This overhead includes a constant time for the hypervisor entry, the entry to the non-privileged region, and the hypervisor exit. In addition to the simple switch time, the TTBR update case requires more time to execute the update instruction in the non-privileged region. Because the instruction execution does not require additional region switches, it does not increase the overhead compared to that of mode switch. On the other hand, the page table update case requires switches between the non-privileged and privileged regions. The update request is sent by invoking the hypervisor call from the non-privileged region, which switches the region to privileged. Once the page table is updated, the non-privileged region is reactivated through the region switch gate. These additional operations are the main factors that increase the overhead of SelMon, compared to the other two cases.

The performance of accessing the OS kernel is also estimated (Table 5). A single read and write operation imposed an overhead of approximately 30% similar to the simple switch. However, the overhead was obscured as the processing data size increases. When the size of the write and read operation is bigger than 80 bytes, the overhead dramatically decreased down to 6% and 2%, respectively. Note that we premap the OS kernel region to the hypervisor; thus, the overhead for the page table update (i.e., hypervisor or secondary page tables) is not included in this result.

7.4 Application and OS benchmarks

The performance impact of SelMon to applications is evaluated by running LMBench and SPEC CPU2006 benchmarks. On the other hand, we used the kernel test collection provided by Phoronix Test Suites to evaluate the overhead imposed to the OS kernel.

LMBench. We measure the latency of basic OS operations such as `open` and `execve` on two different environments: OSs running

Table 5: Performance of read and write operations on original and SelMon-hardened hypervisors. The average latency of 10 runs (in μs), standard deviation, and overhead are described. The size of operations varies from 8 to 800 bytes. The target memory region is pre-mapped; hence, the operations do not include the overhead of the page table update.

	8 bytes		80 bytes		800 bytes	
Operation	Baseline	SelMon	Baseline	SelMon	Baseline	SelMon
Read	3.8	5.1 (σ: 0.56, 1.34×)	7.9	8.1 (σ: 1.1, 1.02×)	50.7	52.1 (σ: 3.63, 1.02×)
Write	4.4	5.7 (σ: 0.67, 1.29×)	7.8	8.3 (σ: 0.82, 1.06×)	51	53.7 (σ: 2.98, 1.05×)

Table 6: LMBench results from Linux OSs on original and SelMon-hardened hypervisors. The average latency (in μs) and standard divation of 10 runs of each SelMon test case are described.

	Baseline	SelMon	Stdev	Overhead
syscall	0.690	0.690	0.004	1.00×
read	1.545	1.546	0.009	1.00×
write	1.310	1.313	0.004	1.00×
stat	4.939	4.953	0.012	1.00×
open+close	10.886	10.956	0.209	1.01×
signal handler	6.532	6.534	0.225	1.00×
sock stream	26.093	26.531	0.919	1.02×
fork+exit	565.940	745.142	7.134	1.32×
fork+execve	1466.65	1899.20	60.620	1.29×
\bin\sh -c	5639	7249	37.824	1.29×

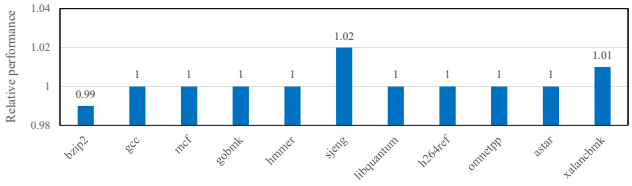

Figure 4: SPEC CPU2006 with original and SelMon-hardened hypervisors. The results are normalized to the measurement from the original hypervisor (lower is better).

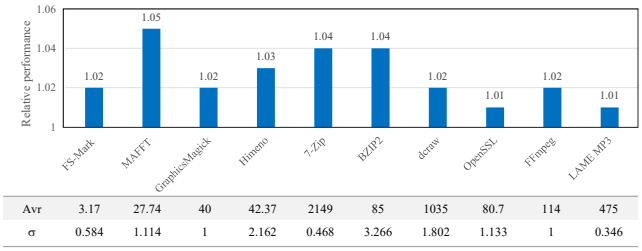

Figure 5: Performance of OS, evaluated by running Phoronix Test Suite with original and SelMon-hardened hypervisors. The average of 5 runs with SelMon is normalized to that of the original hypervisor (lower is better).

on original and hardened hypervisors. Table 6 indicates the results in microseconds. Simple operations, which only introduce low latencies, including syscall, read, write, stat, open, signal handler, and sock, did not show any significant impact of SelMon. We suspect that this is because each test case does not invoke the hypervisor during measurement of the execution time. More specifically, considering that the hypervisor is invoked for the context switch (TTBR update) and the OS page-table update, those tests do not include such OS operations during the measurement.

However, the last three test cases, i.e., the fork, execve, and shell execution, imposed an overhead of approximately 30%, which is aligned to the overhead of SelMon imposed on the hypervisor operation primitives. As described in Section 7.3, the overhead was incurred due to the constant latency of the mode and region switches. The results indicate that the last three cases frequently invoke the hypervisor operations that update the OS page tables for the new process creation. Additionally, due to the relatively high latency of the last three test cases, there might be several context switches that invoke the hypervisor to support the TTBR update.

SPEC CPU2006. Figure 4 shows the performance of CPU2006 test cases with the hardened hypervisor. The result is normalized to the performance measured with the original hypervisor. Contrary to the result of LMBench, the overhead measured with CPU2006 was negligible. A maximum overhead of 2% was observed in `sjeng`. Specifically, bzip2 with SelMon shows better performance. Although we did not thoroughly analyze the reason for this result, we expect this is due to the noise incurred by various factors (e.g., CPU throttling). The result implies that the portion of operations that invoke the hypervisor (i.e., page table and TTBR update) is not significant in each applications. Due to the longer runtime (in seconds) compared to that of LMBench, most of the SelMon overhead was obscured.

Phoronix Test Suite. We evaluate the performance impact to the OS kernel due to the adoption of SelMon. As can be observed in Figure 5, ten kernel test programs from Phoronix Test Suite are used. The overhead varies from 1% to 5% with OpenSSL and MAFFT. In addition to MAFFT, 7-ZIP and bzip2 introduce a relatively high overhead of 4%. Note that bzip2 in Figure 5 takes a much larger input (256 MB) than bzip2 in Figure 4 (at most 50 MB). We expect that large file compression requires frequent context switches and page faults due to disk read and copy operations. This in turn leads to more frequent invocations of the hypervisor because of the implanted hypercalls in the context switch and page fault handlers. Finally, compared to SPEC CPU2006, most test cases from Phoronix result in higher overhead due to their kernel intensive operations.

8 DISCUSSION

Although we show the efficacy of SelMon by porting it to the hypervisor, it can be adapted to protect the software running in kernel

mode, which also provides the required hardware features (i.e., DEP and watchpoint). Because the features are generally available in both the secure states, OS kernels hosted in the rich execution environment (REE) and the TEE created by TrustZone can perform self-protection by arming with SelMon. Some engineering effort is expected to realize this. For instance, the privileged part that verifies and emulates the critical operations needs to be located in the region that is protected by SelMon. Most importantly, the gate code needs to be allocated in the memory region the virtual and physical addresses of which are equally mapped. This is to prevent the attacker from abusing the instruction that updates the SCTLR, which can be exploited to turn off the MMU. In spite of this requirement, our approach is still more flexible than SKEE [17] in that we do not require a particular memory address (e.g., 0x0) to be reserved for the security application.

Although our approach helps reinforce the security of TrustZone by protecting the secure OS in the TEE as aforementioned, it is not compatible with the monitor mode that generally acts as a gatekeeper of the TEE. This is because the watchpoint exceptions are not supported in the monitor mode in essence. As discussed in Section 2.5, previous approaches are not suitable for the protection of monitor mode as well. Hybridizing SelMon and the software fault isolation (SFI) [19, 38, 45] could be a reasonable approach to address this problem. For instance, the lack of watchpoint support, which enforces the non-accessibility, can be compensated by instrumenting read and write operations so that they are enforced not to access the privileged region. We set aside this for our future work.

The specification for the ARM microcontroller (e.g., ARMv8-M) also defines debugging properties including the watchpoint. However, reproducing SelMon with this low-end architecture requires amendments to the current design due to the architectural distinction. For example, the low-end specification does not support MMU and DEP. Thus, we cannot setup the permission of each region's components using the page table and dynamically adjust the DEP policy. Furthermore, to reduce the cost in building resource constraint devices, the debug facilities are generally subject to be removed from the device. We expect employing the memory protection unit (MPU) together with compiler technique [22] to be a possible solution to emulate such missing components. We will further explore this to implement SelMon on low-end devices.

On the other hand, x86 provides hardware watchpoints as well. Unfortunately, the size of the possible monitoring range is limited to 8 bytes per individual watchpoint; thus, it is not suitable for implementing SelMon using watchpoint on x86. Instead, the memory protection key [13], which enables the memory regions to be partitioned into sixteen parts and selectively disables (and enables) the access to each region, could be a viable hardware feature for deploying SelMon to x86-based systems.

9 CONCLUSION

We first analyzed the existing approaches for the self-protection of privileged software and showed why these approaches are not sufficient to be generally adopted in mobile devices. Then, we demonstrated, SelMon, a mechanism to protect the privileged software on the ARM architecture by leveraging general hardware features such as the DEP and watchpoint. To show the effectiveness of our approach, we applied SelMon to the thin hypervisor that monitors the OS kernel integrity. In the performance evaluation, SelMon imposed an overhead of approximately 30% in the hypervisor primitive operations. However, most of the overhead was obscured in SPEC CPU2006 due to its minimal portion of hypervisor invocation in the entire runtime of each test case. In the OS performance evaluation with Phoronix, a maximum overhead of 5% was observed. Although we ported SelMon in the hypervisor, it can similarly be adopted by other modes such as secure and non-secure kernels because of the availability of the required hardware features (e.g., the watchpoint), which highlights the portability of our approach.

10 ACKNOWLEDGMENTS

We sincerely thank our shepherd Landon Cox and all the anonymous reviewers for their feedbacks on this paper. This work was supported by research fund of Chungnam National University, NRF (NRF-2017R1A2B3006360), and ONR (N00014-18-1-2661).

REFERENCES

[1] 2008. Architecture Reference Manual (ARMv7-A and ARMv7-R edition). *ARM DDI C* 406 (2008).
[2] 2017. Android for Work on Samsung KNOX devices. https://kp30.s3.amazonaws.com/0b2e7bf6ffc167b609daed41001d00e9.pdf
[3] 2017. Full TrustZone exploit for MSM8974. http://bits-please.blogspot.kr/2015/08/full-trustzone-exploit-for-msm8974.html
[4] 2017. Here Be Dragons: Vulnerabilities in TrustZone. http://atredispartners.blogspot.kr/2014/08/here-be-dragons-vulnerabilities-in.html
[5] 2018. AMD-V Nested Paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf
[6] 2018. ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile
[7] 2018. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C$_{t}$rustzone$_{s}$ecurity$_{w}$hitepaper.pdf
[8] 2018. Enabling Intel Virtualization Technology Features and Benefits. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf
[9] 2018. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/
[10] 2018. Juno ARMćç Development Platform SoC. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0515f/DDI0515F$_{j}$uno$_{a}$rm$_{d}$evelopment$_{p}$latform$_{s}$oc$_{t}$rm.pdf
[11] 2018. Kgdb. https://elinux.org/Kgdb
[12] 2018. Lifting the (Hyper) Visor: Bypassing Samsung's Real-Time Kernel Protection. https://googleprojectzero.blogspot.com/2017/02/lifting-hyper-visor-bypassing-samsungs.html
[13] 2018. Memory protection keys. https://lwn.net/Articles/643797/
[14] 2018. Second-level page table walk. https://developer.arm.com/docs/ddi0333/latest/memory-management-unit/mmu-descriptors/second-level-page-table-walk
[15] William Arbaugh, David J Farber, Jonathan M Smith, et al. 1997. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 65–71.
[16] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: real-time kernel protection from the ARM trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 90–102.
[17] Ahmed M. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A lightweight Secure Kernel-level Execution Environment for ARM. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.
[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 164–177.
[19] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-Resilient Layout Randomization for Mobile Devices.. In *NDSS*.

[20] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling execute-only memory for COTS binaries on AArch64. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 304–319.

[21] Yeongpil Cho, Donghyun Kwon, Hayoon Yi, and Yunheung Paek. 2017. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*.

[22] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–303.

[23] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2015. Protecting Data on Smartphones and Tablets from Memory Attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, 177–189.

[24] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 292–307.

[25] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual ghost: Protecting applications from hostile operating systems. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 81–96.

[26] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure virtual architecture: A safe execution environment for commodity operating systems. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 351–366.

[27] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 191–206.

[28] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. 2018. Shielding software from privileged side-channel attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1441–1458.

[29] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 287–305.

[30] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture. *Proceedings of the Third Workshop on Mobile Security Technologies (MoST)* (2014).

[31] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure execution of unmodified applications with ARM trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 488–501.

[32] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM trustzone. In *In Proc. of the 26th USENIX Security Symposium*.

[33] Kari Kostiainen, Jan-Erik Ekberg, N Asokan, and Aarne Rantala. 2009. On-board credentials with open provisioning. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM, 104–115.

[34] Shih-Wei Li, John S Koh, and Jason Nieh. 2019. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1357–1374.

[35] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 143–158.

[36] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An architecture for secure active monitoring using virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 233–247.

[37] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 67–80.

[38] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures.. In *USENIX Security Symposium*. 1–12.

[39] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 335–350.

[40] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. 2009. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 477–487.

[41] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen.. In *NDSS*.

[42] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming Smartphones into Secure One-Time Password Tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 976–988.

[43] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 380–395.

[44] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. 2009. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 545–554.

[45] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 79–93.

[46] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *Security and Privacy, 2016. SP 2016. IEEE Symposium on*. IEEE.

[47] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 558–569.