Retrofitting the Partially Privileged Mode for TEE Communication Channel Protection

Jinsoo Jang, and Brent Byunghoon Kang*, *Member, IEEE,*

Abstract—ARM TrustZone provides a Trusted Execution Environment (TEE) to isolate security-critical services, which are generally invoked from the Rich Execution Environment (REE) through a communication channel established by executing the Secure Monitor Call (SMC) with the general registers configured as input parameters. Unfortunately, the communication channel has been abused by adversaries to incur misbehavior of the TEE, to analyze the internal working of the TEE, and to exploit its vulnerabilities. We therefore propose the TEE defense (TFence) framework that enables the creation of a partially privileged (par-priv) process, which benefits from the coordination of the system mode and virtualization extension. More specifically, on ARM architecture, direct invocation of hypercall and SMC is not allowed in the user process; however, we limitedly escalate the privilege of the process to enable it to directly communicate with trust anchors such as hypervisor and TrustZone. This approach enabled us to remove the kernel dependency when the process communicates with the TEE, which also reduces the attack surface to the critical part of the application involved in the communication. Besides, direct communication with the hypervisor facilitates the adoption of application-shielding approaches to protect the critical part and to restrict arbitrary access to the TEE.

Index Terms—Mobile Device Security, Trusted Execution Environment (TEE), ARM TrustZone, Communication Channel Security.

1 Introduction

ARM TrustZone technology has been widely employed to create the Trusted Execution Environment (TEE) in ARM processor-based devices such as tablet PCs and smartphones. The trusted applications (TAs) such as crypto key management [1], [2], payment [3], and authentication [4] are deployed inside the TEE to protect their confidentiality and integrity. On the other hand, the client application (CA) as a counterpart of each TA has been deployed in the Rich Execution Environment (REE) to trigger the TAs. Specifically, the CA leverages the TrustZone kernel driver to send messages to the TEE by executing the Secure Monitor Call (SMC) instruction with kernel privilege, which creates a communication channel between the REE and the TEE.

Unfortunately, this communication channel has been abused by adversaries to attack the REE, and to find and exploit the vulnerabilities in the TEE. For example, as shown in [5], [6], the adversary can send a malicious message to the TEE to escalate his privilege. The adversary can also perform a brute-force attack against TEE by continuously executing SMC instructions with arbitrarily crafted messages as parameters of the instructions. By doing so, the adversary can analyze the internal working in the TEE [7] and exploit the vulnerabilities of the TAs (and TEE OS) to exfiltrate the secrets stored in the TEE and to obtain full control over the TEE [8], [9], [10], [11], [12]. *This problem fundamentally stems from the fact that the (1) message authentication, (2) message integrity protection, and (3) message verification are not strictly enforced since they have been conducted by the potentially malicious kernel.*

- *J. Jang, and B. Kang are with Korea Advanced Institute of Science and Technology.*
  E-mail: {jisjang, brentkang}@kaist.ac.kr.
- ** B.Kang is a corresponding author.*

Manuscript received January 8, 2018; revised xx xx, 2018.

To address these problems and protect the communication channel between the CA and the TEE, we propose the TEE defense (TFence) framework. TFence enables developers to create a partially privileged (par-priv) process, which can directly communicate with trust anchors such as the hypervisor and TrustZone without depending on the kernel. Particularly, an application that runs in par-priv mode can directly execute hypercall (HVC) and SMC instructions. At the same time, TFence restricts the par-priv process from executing other security critical instructions that should not be executed by user applications.

We created the par-priv mode by leveraging the overlooked hardware feature –System mode– which is one of the privileged processor modes available on ARM architecture. By running the process in System mode, we escalate its privilege. Besides, to prevent the par-priv process from abusing its escalated privilege, we implemented TFence as a micro-hypervisor that interposes every interaction between the par-priv process and the kernel to prevent any malicious behaviors.

In terms of securing TEE, the benefit of adopting par-priv mode is that we can remove the kernel dependency when the process needs to interact with the TEE. The removal of this dependency also reduces the attack surfaces. In other words, to establish a secure communication channel, a small part of the pre-authorized application that is involved in communication needs to be protected; *the kernel objects (e.g., APIs and function pointers) scattered over the memory do not have to be monitored and protected*. The direct communication also enables existing approaches for x86 [13], [14] to be readily employed on ARM architecture. We adopt application compartmentalization and shielding approaches to protect and authenticate that part of the CA that becomes involved in creating and sending messages. In addition to protecting the TEE, abusing the TEE to attack the REE (e.g.,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

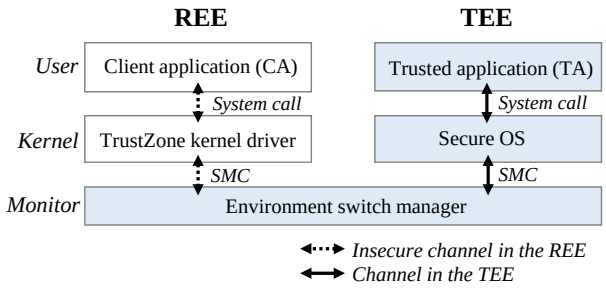

Fig. 1. Communication channel between CA and TA. The channel in the REE is insecure in the presence of a compromised kernel.

BOOMERANG attack) [5], [6] can be prevented because TFence restricts the arbitrary message delivery into the TEE and also validates every message.

We implemented the TFence prototype on Arndale board equipped with an ARM Cortex-A15 dual-core processor [15]. In the performance evaluation, the activation of hypervisor mode imposes maximum 11% and 6% of overhead to the overall system running the micro- and application benchmarks. Notably, in spite of the overhead attributable to hypervisor mode activation, we gain some performance benefit from the removal of kernel dependency on the invocation of privileged instructions, which were 2.5% and 17.6% for the SMC and hypercall invocations of TFence compared to the result of runs without TFence, respectively.

The contributions of our work can be summarized as follows:

- We introduce par-priv mode execution by leveraging System mode and virtualization extension on ARM. Our approach does not require any change to the hardware architecture design.
- By taking advantage of the par-priv mode, we propose a new mechanism for communicating with the TEE, which enables the application to directly communicate with the trust anchors without depending on the kernel.
- To secure the TEE, we propose a way to authenticate the message sender, and to protect and verify the message bound for the TEE, which was not considered in the original design of TrustZone.

2 Background and Related Work

2.1 TrustZone Service Invocation and Vulnerability

TrustZone aims to provide a TEE to devices based on an ARM processor such as smartphones, tablets, and DTVs. The technology is a security extension to the processor that enables the system (e.g., memory, register and peripheral) to be partitioned into two domains: the REE and the TEE.

Based on the protection and isolation guaranteed by TrustZone, trusted applications (TA) that handle critical resources such as user credentials [4] and digital rights [16] have been deployed and executed in the TEE on mobile devices. TrustZone is also utilized to host OTP [17], remote attestation [18], security monitors [19], [20], memory forensic framework [21], stealthy debugger [22] and an architecture for provisioning credentials [23] in the TEE. In addition, TrustZone components are analyzed in-depth and leveraged to realize the TEE virtualization [24], [25].

On the other hand, the client application (CA) in the REE, which is deployed as a counterpart of TA, places parameters in the domain shared memory and asks the kernel to invoke the Secure Monitor Call (SMC) instruction to use the TAs. The invocation of SMC switches the processor mode to Monitor which is introduced as part of the TrustZone technology for managing the switches between the two domains. Depending on the design of the TEE application, some TEE-based services (e.g., periodic kernel-integrity monitoring) do not need to be explicitly invoked by the CA because it can be triggered by the timer interrupts that are dedicated to the TEE. However, as of this writing, all the studies in which the processor is explicitly switched to Monitor mode use the same mechanism–invoking SMC instruction with kernel privileges.

```
1 MOV R0, #TA_id //1st param: invoked TA id.
2 LDR R1, =req_buf//2nd param: request buffer.
3 LDR R2, =res_buf //3rd param: response buffer.
4 SMC #0 //SMC with an immediate value 0x0
```

Listing 1. Example message format for TA invocation.

Problem. As already presented in [5], [8], [9], [10], [11], [12], [26], the communication channel between the CA and TA is vulnerable (Figure 1). That is, adversaries can easily compromise the messages transferred to/from the TEE. For instance, as can be seen in Listing 1, the message transferred to the TEE is created by setting several general registers (e.g., R0 - R2). The messages might contain the service number indicating the desired TA and parameters such as the address of the request/response buffers. An adversary can easily manipulate the register values and application-specific data structures in the buffers to deliver a maliciously crafted message to the TEE, which enables the adversary to cause the misbehavior of the TEE, to analyze the internal working, and to find and exploit the vulnerabilities of TAs to compromise the entire security of the TEE.

SeCReT [26], as the first work to secure the channel, provides the session key (to CAs) that can be utilized to encrypt the message. To protect the session key from an untrusted kernel, SeCReT interposes mode switches and removes the key from the memory during the kernel mode execution. However, the approach that leverages the session key might entail certain security problems. For example, even if the original key is securely protected, a copy of the key can be created during execution of the CA and be exposed to the adversary.

Machiry et al. showed a BOOMERANG [5] attack that enables a malicious user process to arbitrarily access the kernel memory by abusing the TEE. Specifically, they crafted the message delivered to the TEE such that it leads the TEE to modify the critical kernel objects. In essence, this attack was possible because the TEE never checks the validity of the message semantic (e.g., mapping request to the kernel memory). To address this problem, they proposed a mitigation that enforces the TEE to consult kernel whenever memory mapping to the REE is needed. However, the proposed solution was specific for a BOOMERANG attack, which aims to protect the REE; thus, the message integrity and restriction of the arbitrary SMC invocation from the REE were not considered.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

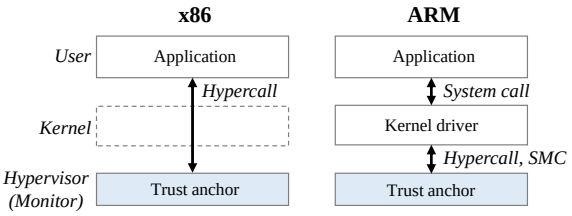

Fig. 2. ARM vs. x86 in communication with trust anchors. Specifically, as an architecture design difference, ARM counts on the kernel to interact with trust anchors.

In this work, we attempt to protect the TEE by securing the communication channel between the CA and the TA. Notably, our work is differentiated from previous TrustZone-based work [17], [18], [19], [20], [21], [23], [26], [27], [28], [29] in that we partially escalate the privilege of the CA, enable application-level TEE-service invocation, and verify the legitimacy of the invocation, whereas the previous work depended on the potentially malicious OS kernel to invoke the SMC instruction for access to the TEE.

2.2 Communication with Trust Anchor on x86 vs. ARM

Research on x86 that aims to protect an application from an untrusted OS have utilized user-level hypercalls as a means of directly communicating with a trust anchor such as a hypervisor [13], [14], [30], [31], [32]. For instance, McCune et al. [13] used hypercalls to allow Pieces of Application Logic (PAL) to communicate with the TrustVisor without awareness of a malicious OS. Overshadow [30] creates a secure communication channel by enabling the user-mode shim to invoke the hypercalls for interacting with the trustworthy hypervisor. In addition, to protect the high-assurance process (HAP) from an Iago attack [33], Inktag [14] utilizes user-level hypercalls to deliver security-critical information such as the list of memory maps for HAP. As shown in the previous work on x86, it seems quite natural for security frameworks to use hypercalls to build a secure communication channel since hypercalls are allowed to be invoked in any processor mode (i.e., both in user and kernel modes).

In contrast, on ARM architecture, a hypercall (i.e., HVC in ARM) is designed as a privileged instruction; hence, a user process is not allowed to invoke hypercalls, as shown in Figure 2. Not only a hypercall, but also SMC is defined as a privileged instruction [34]. Thus, the user process that needs to communicate with the hypervisor or TrustZone-based TEE would need to request the kernel to invoke those instructions.

From the perspective of security, the restriction of the direct interaction between the process and trust anchor (i.e., the kernel dependency on communication with the trust anchor) has several limitations. That is, an adversary can compromise the kernel to arbitrarily control the communication between the shielded process and the trust anchor, and to attack the trust anchor. Furthermore, in terms of building a security framework on ARM, the restriction can increase the complexity of the design of the security framework compared to that of x86. For instance, we might need to implement an additional crypto function to protect the communication channel on ARM, whereas a direct channel is essentially available on x86.

TABLE 1
ARM architecture modes, registers, and privileges. Notably, System mode shares the same register set with user mode, but it runs with kernel privilege.

	User	System	Supervisor	Abort	...
Core register	R0 - R12	R0 - R12	R0 - R12	R0 - R12	...
	SP	SP	SP_svc	SP_abt	...
	LR	LR	LR_svc	LR_abt	...
Special register	-	-	SPSR_svc	SPSR_abt	...
Privilege	User	Kernel			

2.3 ARM Hardware-assisted Virtualization

The high-end ARM processors support hardware-assisted virtualization. The hypervisor mode (HYP) is used for configuring the virtualization environment and managing the multiple guest VMs. In particular, various guest VM events can be trapped in HYP by setting up the hypervisor control registers, such as Hyp System Trap Register (HSTR). The stage-2 paging enables VM isolation. Paging translates the guest VMs physical address (that is the intermediate physical address) into the real hardware address (physical address). This mapping information is available in the stage-2 page table, whose base address is indicated by the Virtualization Translation Table Base Register (VTTBR). To activate stage-2 paging, the VM flag in the Hyp Configuration Register (HCR) must be set. Hardware-assisted virtualization is also available in x86 architecture and has been leveraged to build various security benefits, including kernel integrity monitor [35], [36], [37] and application shielding [13], [14], [30]. In our work, we used virtualization to protect the communication channel between the CA and TEE.

2.4 Revisiting System Mode on ARM

Table 1 indicates the available modes together with the privileges and registers of each mode on high-end 32-bit ARM processors such as Cortex-A15. The modes from System to FIQ are generally regarded as kernel modes. User and kernel modes are banked for each security domain: the REE and the TEE.

Notably, System mode shares all the registers with user mode. The only difference between the System and user modes is that System mode has higher privileges than user mode. As shown in the table, System mode has the kernel privilege similar to other modes such as Supervisor and Abort. According to ARM [38], System mode was originally introduced to facilitate nested exception handling without corrupting the saved return address in LR.

However, System mode has either never been used, or been used to a very limited extent in high-end OSs such as Linux (instead, Supervisor mode is used for handling the exceptions). According to our analysis of the Linux kernel (4.4.32) implementation, a kernel compiled with the ARM instruction set (32-bit instruction set) does not use System mode. Although a kernel compiled with the Thumb-2 instruction set (16/32-bit mixed instruction set) utilizes System mode, it is limitedly utilized in the macro that tentatively switches mode to System for pushing and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

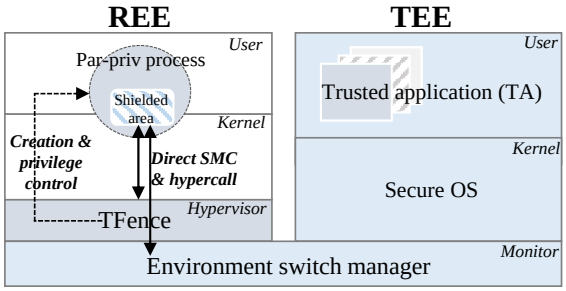

Fig. 3. TFence enables creation of a partially privileged (par-priv) process capable of directly communicating with trust anchors without kernel dependency. This approach reduces the TCB for secure communication to that part of the process that is responsible for creating and sending a message. Also, it facilitates the authentication, integrity protection, and verification of the message delivered into the TEE.

popping the SP and LR of user mode, whereas the same operation can be conducted without using System mode in a kernel compiled in an ARM instruction set. In TFence, this overlooked hardware feature is leveraged to build a secure communication channel as discribed in Section 4.2.

3 Attack Model and Assumptions

We assume a mobile/embedded system environment that is built based on ARM processors supporting virtualization and TrustZone extensions. The hardware-based memory protection of TrustZone is properly configured; thus, there exists no misconfiguration-based vulnerabilities such as allowing read/write access to the TEE by creating a memory mapping to that area. Further, since the device manufacturer and the TrustZone OS and trusted application (TA) providers are not malicious, software stacks built in the TEE are not malicious either. The general TEE application development model that enables TEE providers to verify the client application (CA) and TA before their deployment in the device is assumed. Secure boot [39] guarantees the integrity of the images loaded at boot time. We also assume the presence of IOMMU and its proper configuration [40]; therefore, DMA attacks such as direct manipulation of the physical memory are not available. Finally, attacks based on physical access [29], [41] and side channels are beyond the scope of our attack model.

On the other hand, an adversary can arbitrarily send malicious messages to the TEE. To achieve this, an adversary can either exploit the vulnerability of the legitimate CA or create own malware. By doing so, he can compromise the kernel by abusing the TEE privilege and semantic gap between the REE and the TEE [5]. However, the adversary can also exploit the REE OS vulnerability to escalate his privilege; thus, the REE is basically untrusted in our attack model. Based on the control over the kernel, he can freely attempt to compromise the TEE (e.g., exfiltrating the secrets or perpetuating his attack [42]).

4 Design of TFence

TFence (TEE-defense) aims to prevent abusing the TEE and to protect the TEE by securing the communication channel between the CA and TA, as described in Figure 3. To this end, TFence creates a partially privileged (par-priv) process by leveraging the System mode of ARM, which enables the process to directly communicate with trust anchors such as the hypervisor and TrustZone. The advantage of direct communication is that it removes the kernel dependency when the process communicates with the trust anchors and thus reduces the attack surfaces of the communication to that part of the application that is involved in message creation and transmission; *hence, widely dispersed kernel APIs and data structures that have been involved in the TEE communication do not need to be traced, monitored and protected*. TFence protects the relevant part of the application and the integrity of messages sent to the TEE by adopting an application-shielding approach, which also benefits from the direct communication channel built by TFence. Moreover, all the messages bound for the TEE are trapped and verified by TFence before they are passed into the TEE. In this section, we describe the design detail of TFence and how its goal is achieved.

4.1 Boot Time Initialization

TFence is designed as a micro-hypervisor that runs with a higher privilege than the OS. We initialize TFence as part of the boot procedure of the device to benefit from secure boot. Because secure boot [39] performs chained verification of the loaded images, the integrity of our TFence implementation as well as the boot-loader, TrustZone OS, and REE OS can be protected.

During the boot, we activate the hypervisor mode by configuring hypervisor-related registers. In addition, the exception vector for the hypervisor mode is mapped in the memory that is isolated from the REE OS. In general, the exception vector defines the address of the handlers for each exception, which should be individually mapped for the activated domains (e.g., REE OS, hypervisor, and Monitor). TFence only needs to handle part of the hypervisor-trap exceptions. We simplify the implementation by disabling the MMU in hypervisor mode. Thus, TFence runs based on the physical memory without maintaining the page tables for the hypervisor mode, whereas the REE OS continues to use the virtual address. TFence also enables stage-2 paging which is similar to the nested paging in x86 [43], and creates stage-2 page-table mappings for the REE. The Intermediate Physical Address (IPA), which is the physical address in the view of the REE OS is identical to a real physical (machine) address in the mapping. Note that the memory region for TFence hypervisor implementation is not included in the mapping to protect TFecne from the adversary.

Permanent invariants. Although we assume the kernel is untrusted, part of kernel components that are patched for the TFence implementation need to be protected from malicious modification. To achieve this, TFence defines some REE kernel objects as permanent invariants and protects them by using stage-2 paging. These objects are the kernel static area (e.g., code and data), and exception vector (and handler code) that are initialized at boot time and should not be updated during runtime. TFence simply configures read-only permission in the stage-2 page that contains these objects. Not only the object itself, but also the mappings to the object (i.e., the REE page table entries addressing to the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

object) should be immutable during the runtime of the device. In other words, every process that is newly scheduled should have the same page table entries for those objects. TFence also defines the mapping information as permanent invariants, but it does not write-protect them in stage-2 paging to minimize the performance overhead incurred by frequent stage-2 paging faults. The permanent invariants are checked on demand during the par-priv process execution to prevent a confused deputy attack [5].

4.2 Creating Par-priv Process

TFence utilizes the System mode to escalate the privilege of the application in the REE (e.g., CA), which enables it to directly invoke privileged instructions such as HVC and SMC. The System mode is superior to other kernel privileged modes (e.g., IRQ, FIQ and Abort) for creating the par-priv process in the following aspects:

(1) Using System mode simplifies the implementation. System mode and user mode share every register including the stack pointer (SP) register and link register (LR). Thus, creating the par-priv process by leveraging System mode is straightforward, because System mode only needs to switch the mode bits in the Current Program Status Register (CPSR) to System. On the other hand, other modes such as Supervisor have their own copies of registers such as SP and LR. Thus, the dedicated register values need to be updated whenever a mode switch happens for par-priv mode execution.

(2) In the ARM architecture, LR is used to store a return address. Depending on the processor mode, the value of LR can be automatically updated through hardware. For instance, LRs dedicated for the exception modes, such as IRQ and Abort, are automatically updated when corresponding exceptions occur. This behavior is unsuitable for our purpose, which is running a user process with privileged mode, because it can break the control flow by overwriting LR. By contrast, LR for the system mode is not affected by any exception occurrence.

(3) System mode is barely used in Linux; thus, utilizing it requires less effort for investigating the compatibility of TFence with the existing kernel implementation.

In the following subsections, we describe how TFence creates the par-priv process and confines its privilege to the direct interaction with trust anchors.

4.2.1 Privilege Escalation

On ARM, the CPSR reflects the current status of program execution including the current mode. The flag for the current mode has five bits to present each mode, and specifically 0b11111 is predefined for System mode.

To convert the mode from user to System, we create and export the system call that handles the request for creating the par-priv process. When the user process invokes the system call by invoking the supervisor call (SVC), the mode switches to kernel (i.e., Supervisor mode) and the CPSR of the user process is saved as an SPSR in the kernel stack. The information in the SPSR (e.g., CPU mode flag) is restored when the process is rescheduled. Thus, the system call handler simply updates the mode flag of the SPSR to System. This makes the process run in the System mode when its execution is resumed.

TABLE 2
Group of system operations and its trap.

Co-processor (CP)	System operations	Trapped by
CP15	ID, system control, memory protection (control), memory fault, cache & address translation, TLB, performance monitor, memory mapping, DMA, security extension, process & thread context, vendor-specific feature	HSTR
CP14 (Device specific)	Jazelle functionality & ThumbEE configuration	HSTR
	Accesses to debug registers	HDCR
	Accesses to Trace registers	HCPTR
CP0-13 (Device specific)	Vendor-specific or optional features (e.g., Floating-point instructions)	HCPTR

4.2.2 Privilege Restriction

Since System mode is one of the kernel privileged modes, any process (even malware) running in System mode can execute any privileged instructions. Thus, the privilege for the par-priv process should be appropriately confined to building a direct communication channel between the par-priv process and the trust anchors.

Restricting privileged operations. Privileged instructions other than HVC and SMC should not be allowed to be invoked by the par-priv process to avoid thwarting the security of the REE. This is because some privileged instructions can configure the control registers that can be abused by an adversary. For instance, the System Control Register (SCTLR) contains flags for configuring security critical features such as MMU, cache, alignment check, and writable execute never (i.e., DEP) attributes.

In essence, hardware-assisted virtualization enables access to those control registers to be trapped. Thus, to protect the control registers from a malicious par-priv process, TFence activates the hypervisor-trap whenever the par-priv process is executed. *In our implementation, all the accesses to the critical control registers are possibly trapped by solely leveraging Hyp System Trap Register (HSTR) because our development board only provides CP15 and CP14 partially*. However, depending on the device, other optional features defined by CP0-14 can also be trapped by hypervisor as shown in Table 2. Any trap caused by access to the control register is regarded as abnormal behavior because the par-priv process should perform only user-level operations. The trap is disabled when the processor mode switches back to the kernel for handling exceptions. We achieved this by inserting HVC instructions in every entry/exit point to/from the kernel to invoke TFence at every mode switch between the par-priv process and kernel.

On the other hand, some privileged instructions such as MRS (Move to core Register from Special register), MSR (Move to Special register from core Register), and CPS (Change Processor State) cannot possibly be trapped by hypervisor-trap configuration. These instructions can be abused to read or write to registers dedicated to other modes such as LR, SP, and SPSR. For instance, a malicious par-priv process using those instructions can read or write the stack pointer value of other privileged modes (e.g., Supervisor mode). In addition, it can arbitrarily change its mode to other kernel-privileged modes. To address this problem, TFence always saves dedicated register values of other modes before entering par-priv mode, and restores them when the mode switches back to the kernel. Further, TFence prevents the par-priv process from accessing the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

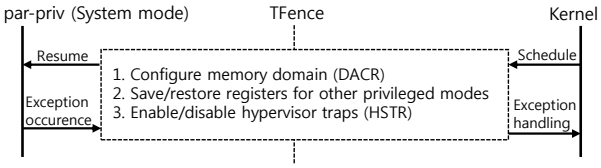

Fig. 4. Transition between par-priv process and kernel.

kernel memory; thus, an adversary would not benefit from changing the current mode to another kernel-privileged mode.

Memory access control. TFence isolates memory for the par-priv process by leveraging the default memory-domain configuration in Linux and DACR. On a 32-bit ARM processor, the first-level page table entry can define the mapped memory region as one of the sixteen domains by using its 4-bit domain flag. In Linux, 0, 1, and 2 are assigned to the kernel, user, and device memory region, respectively. On the other hand, DACR defines the access permission for each of the sixteen domains by using 2-bit flags for each domain. The possible permissions are (1) No access (0b00), (2) Client: permission check against page table attributes (0b01), (3) Reserved: unpredictable effect (0b10) and (4) Manager: no permission check (0b11). Linux sets DACR in a way to ensure that all the domains have Client permission (0x55555555), and thus every memory access follows the page-table configuration.

In TFence, although the par-priv process runs with System mode, the process should not access to any other domain except the user. *Because the kernel, user, and device domains are already classified in Linux, we simply need to adjust the access permission settings in DACR dedicated to each core to isolate the par-priv process in the user domain.* Specifically, when the par-priv process is executed, TFence configures the DACR value to 0x4 to ensure that only the user domain is accessible. The value of DACR is restored to the default value (0x55555555) when the mode switches back to the kernel. Note that although System mode also has a privilege to manipulate DACR, malicious code running as a par-priv process cannot manipulate the DACR due to the access restriction to the control registers enforced by TFence. Finally, as utilized in ARMLock [44] and Shreds [45], we also obtain the performance benefit from using DACR instead of stage-2 paging for user domain isolation. In other words, leveraging DACR does not impose overhead incurred by the TLB and cache maintenance, contrary to page-table approaches.

Transition gate protection. In TFence, the exception vector and return_to_user kernel code play roles in triggering TFence at every switch between the par-priv process and kernel (Figure 4). We achieved this by implanting hypercalls to invoke TFence at the entering/exiting points of the kernel. Once invoked by the hypercalls, TFence (de)activates the privilege operation restriction and the memory protection for the par-priv process as described above. Thus, the transition gates between the par-priv process and the kernel should be protected as well. The location of the exception vector can be varied based on the configuration of control registers, such as VBAR and SCTLR; however, Linux maps it at 0xFFFF0000 with read-only access permission for both the user and kernel modes. The malicious par-priv process cannot reconfigure the address of the mapped exception vector since access to the control registers is restricted by TFence. Moreover, the physical memory area that contains the exception vector and the return_to_user code is immutable because it is protected by stage-2 paging at boot time. Finally, because TFence already isolates the kernel memory that contains the page tables from user domain, even malicious code running with par-priv mode cannot reconfigure the mapping to the transition gates.

4.3 Building Shielded Process

4.3.1 Overview

A TA that is deployed in the TEE performs security-critical operations such as crypto, whereas its counterpart–a client application (CA)–simply invokes the TA. Although the CA does not conduct any critical operations, as can be seen in Section 2.1, it can be abused to send maliciously crafted messages to the TEE. To prevent this, we adopt the application shielding approach.

Figure 5 illustrates an example pseudocode of the shielded application. The application can be separated into two parts (i.e., non-shielded and shielded) based on the 4-KB page granularity by using a linker script. The non-shielded part first escalates the privilege of the process to the par-priv by invoking a system call. Then, it enables TFence to recognize the shielded part by using a hvc_shielded_part_registration, which leads TFence to validate the integrity of the shielded part.

The shielded part is invoked by a subroutine call from the non-shielded part, which causes a stage-2 paging fault since a different stage-2 page table (from that of the REE kernel) is maintained for the shielded part. The fault is trapped and handled by TFence to switch the stage-2 page table, and thus enables the shielded part. Owing to the separation of the stage-2 page tables, the shielded part is completely isolated from the non-shielded part and REE kernel. In addition, the heap protection hypercall enables any dynamically allocated memory in the shielded part to be registered and protected by TFence as well.

In general, the shielded part might play a critical role in creating a message bound for the TEE and directly sending it to the TA. The advantage of the shielding is that we can authenticate the message sender by using the protected (immutable) part as an identity of specific CA; in turn, it enables generation of a strict message verification policy based on a close correlation between the CA and TA. Moreover, it ensures that the message is created without adversary intervention, provided that part of the CA is free of adversary-controllable bugs.

4.3.2 Initialization

To protect the TEE, only the authorized code is allowed to send any message to the TEE and the message integrity should be guaranteed. We achieve this by isolating the application logic responsible for creating and sending the message from the remaining part of the application and designate that part to be protected as a shielded area by TFence.

We enable TFence to recognize the shielded part by using a hvc_shielded_part_registration, which can be directly invoked by a CA running as a par-priv process. Once TFence

```
convert_par_priv_process ()                          //system call
hvc_shielded_part_registration (addr, size, entry)
                                                     shielded part
in_buf_addr = malloc_and_init (in_size)
out_buf_addr = malloc_and_init (out_size)
hvc_heap_protection (in_buf_addr, in_size)
hvc_heap_protection (out_buf_addr, out_size)
create_input_message (in_buf_addr)
/* direct SMC invocation */
invoke_TA (TA_id, in_buf_addr, out_buf_addr, format_id)
hvc_shielded_part_termination ()
```

Fig. 5. Example pseudocode of shielded process, which aims to authenticate and protect the logic responsible for TEE communication, and thus restricts arbitrary access to the TEE.

receives the request for the shielded part registration, it locks the REE memory mapped to the shielded part and calculates the hash of the part by using the start address and the size delivered as parameters of the hypercall. In addition, it validates the hash against the pre-calculated hash that is part of the metadata of the CA, which also contains the allowed entry points of the shielded part and is signed with a device specific key (or the TEE provider's private key). Remote attestation conducted between TFence and the remote root of trust would also be a reasonable option for the validation. Either way would require TFence to cooperate with TrustZone to use the device specific key.

Once hash validation has been completed, TFence saves the REE page-table mappings information of the shielded parts (i.e., the addresses of the 1st and 2nd page table entries and their values for mapping the shielded part) as tentative invariants and configures the access permission of the shielded part in the stage-2 page tables. There are two of these page tables, i.e., for the non-shielded and shielded parts, which are maintained by TFence. The table for the non-shielded part is generated at boot time, and maps the entire REE region as described in Section 4.1. During registration, TFence configures the access permission of the shielded part memory to read-only and non-executable in this page table (before the hash validation). On the other hand, the stage-2 page-table for the shielded part is created during the registration process, and it only contains the mapping for the shielded part of the CA (e.g., separated code and data for the shielded part) including the exception vector.

The transition between the two stage-2 page tables happens based on the occurrence of stage-2 paging faults. Any control flow transition to the shielded from the non-shielded part causes stage-2 instruction fetch faults due to the non-executable permission set in the stage-2 page-tables of the non-shielded part. TFence validates the faulting address against the registered entry points and configures the Virtualization Translation Table Base Register (VTTBR) to map the stage-2 page-table of the shielded part if the fault is legitimate. The switch to the opposite side (e.g., context switch to another process) happens in a similar manner based on the occurrence of faults due to a missing mapping to the non-shielded part in the stage-2 page table of the shielded part and the approach followed by TFence to handle this fault. Because we assign a different virtual machine identifier (VMID) for each of these two parts, an additional TLB invalidation between the transitions is not required.

4.3.3 Runtime Protection

Memory access control. For heap memory protection, TFence provides hypercalls to enable the shielded part to notify the starting address of the new allocation and its size to TFence. If the heap allocation requires a new page mapping, the newly mapped area is configured as read-only and read/write in the stage-2 page tables of the non-shielded and shielded parts, respectively. In addition, TFence keeps the physical address of the area and the REE page-table mapping information to that area as tentative invariants. The invariant list is checked before creating a new invariant to prevent existing heap objects from being overwritten or the mapping to them from being redirected to a malicious mapping [33].

Stack protection is triggered when a stage-2 paging fault happens due to entry to the shielded part. TFence configures the access permission of the stack in the stage-2 page tables as read-only and read/write for the non-shielded and shielded parts, respectively. Therefore, only the shielded part should be able to manipulate the stack without incurring any stage-2 page fault. Furthermore, as in the case of heap memory protection, the protected stack and mapping information are also saved as tentative invariants for future verification.

On the other hand, to handle corner cases such as direct kernel access to the user space memory (e.g., *copy_to_user*), TFence provides a hvc_allow_access to enable the shielded part to explicitly register the start address and size of the memory area allowed to be manipulated by the kernel. Hence, any stage-2 write fault is validated against the registered information, and the write attempt is emulated by TFence if it is confirmed as being legitimate. The semantic of the written value also has to be validated by the shielded code.

In signal handling, the kernel can manipulate the register values before the signal handler routine starts. Hence, TFence regards signal handling as an untrusted operation. Thus, it does not allow any write attempt to the memory of the shielded part until signal handling finishes. Any SMC instruction invocation that happens during signal handling is trapped and ignored by TFence. To explicitly notify the start and end of signal handling, we inserted hypercalls to the kernel code such as *setup_frame* and *restore_sigframe*, which are set as invariants and protected during boot time, as described in Section 4.1.

Register protection. During execution of the shielded part, the occurrence of exceptions switches the processor mode to kernel. Before starting the exception handling routine, the context of the process such as general registers (R0-R12, SP and LR), preferred return address and Current Program Status Register (CPSR) is stored in the kernel stack. To protect the context from the compromised kernel, TFence copies the values to the isolated memory and restores them when the mode switches back to the par-priv mode.

Note that this register protection procedure is different from that for restricting the privilege of the par-priv process as discussed in Section 4.2.2. In other words, the protection

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

TABLE 3
Summary of TEE protection mechanism.

Sender authentication	Load-time: hash validation of the shielded part by using CA metadata (§4.4.1). Runtime: SMC provenance validation (§4.4.3).
Message integrity	Tentative invariants maintenance and check between mode switches (§4.4.2).
Message verification	Message format validation and sanitization by trapping SMC (§4.4.3).

shown here aims to protect the par-priv process from the kernel, whereas that in Section 4.2.2 prevents a malicious par-priv process from attacking the kernel.

4.4 TEE Protection Mechanism

In this section, we describe the TEE protection mechanism of TFence, which adopts the concept of an application-shielding approach on x86 [13], [14], [30], [31] by benefiting from the par-priv mode.

4.4.1 Message Sender Authentication

To ensure that only the authorized code is allowed to send any message to the TEE, we first separate the application into two parts (i.e., non-shielded and shielded). The shielded part contains the logic responsible for creating and sending the message. For example, the CA for DRM service can isolate the functions for directing the encrypted stream to the request buffer and invoking SMC in the shielded part. Then, the hash of the shielded part is validated against the pre-calculated hash that is part of the metadata of the CA. Once hash validation has been completed, the shielded part is protected by configuring the stage-2 page tables, and also registered as the authorized code allowed for the SMC invocation. Whenever the SMC is invoked, it is trapped by TFence and the message sender is authenticated based on this registered information.

4.4.2 Message Integrity

Since the memory of the shielded part (i.e., code, data, stack and heap) is protected by using stage-2 paging, the adversary cannot directly access the memory to manipulate the message. However, more sophisticated attacks such as Iago [33] or multi-core-based time-of-check-to-time-of-use (TOCTOU) attacks should also be considered to protect the message integrity. For instance, an adversary with kernel privilege can try to overwrite the existing heap objects when a new heap memory is allocated. Furthermore, while the shielded part is running, an adversary can timely remap the page tables on another core expecting one of the protected objects to be written with a malicious payload.

TFence prevents Iago style attacks by maintaining memory objects and their page-table mapping information as tentative invariants; thus, whenever a new memory allocation happens, the list of the invariants is always checked first. A TOCTOU attack is also detected by the invariant check mechanism, which is performed whenever mode switches occur between the user, kernel, and Monitor. We discuss the efficacy of the defense mechanisms in Section 6.1.

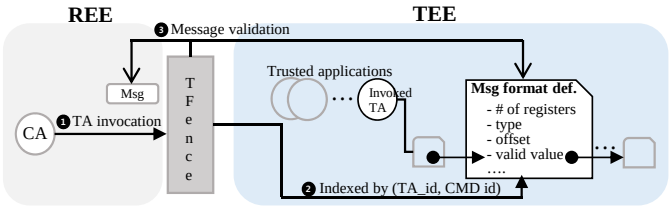

Fig. 6. TFence traps TA invocation and verifies the message bound for the TA.

4.4.3 Message Verification

Depending on the TA type or TEE OS implementation, the message format can be varied. For example, Qualcomm's TrustZone implementation defines several SMC invocation formats based on the atomicity of the operation of the requested TA and the number of parameters delivered to the TA [46]. The registers for the parameters can contain a simple integer value or pointer that maps the address of the message buffers for the request (and response) to (and from) the TA. Besides, the message buffers themselves can contain the application specific (i.e., TA) data structure that defines several different types of members (i.e., scalar and pointers) for communication between the CA and TA.

Thus, TFence needs to recognize the semantics of message formats to properly validate the message when the SMC instruction is invoked. This requires current TA validation processes conducted by TEE providers or the TA development process to be reinforced with a supplementary task that analyzes and extracts the semantics of a message consumed by each TA. The metadata defining the available message formats can either be provided with the corresponding TA, or can be part of the TEE and updated by the firmware upgrade procedure of the device when new formats are defined. By doing so, the predefined message formats can be shared with TFence and application developers.

In TFence, we designed the PoC message format to enable it to present serialized information such as the number of registers used, the type (i.e., scalar or pointer), and offset of each object in the buffers, and imposed a unique message format identifier. Accordingly, the identifier can be provided to TFence when the SMC is executed. The current TFence prototype utilizes the immediate value of the SMC instruction to place the identifier. As the immediate value consists of 4 bits on 32-bit ARM processors, it can present up to sixteen formats associated with each TA identifier. In the case that TA needs to define more than sixteen formats, we can reserve one of the general registers to present more information. On 64-bit processors, the size of the immediate value is 16 bits, and we expect this to be sufficient to be used for delivering the format information.

Trapped and verified SMC. To verify every message sent to the TEE, instead of allowing the SMC instruction invocation to directly switch the processor mode from par-priv to Monitor, TFence traps all the SMC instruction execution that occurred in the REE and verifies its legitimacy (Figure 6). This verification process might be disadvantageous in performance compared with the alternative design that verifies messages in the TEE. This is because of the overhead

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

TABLE 4
TEE-related CVEs and TFence defense mechanism.

Group	CVE id	Exploit primitive	TFence defense mechanism
G1	CVE-2014-4322	Memory bound check failure, integer overflow	Sandboxing (exploit isolation) in par-priv mode
	CVE-2015-4421	Memory bound check failure, arbitrary memory write	
G2	CVE-2015-6639, CVE-2015-6647	Memory bound check failure, buffer overflow	Message authentication, integrity protection, and verification
	CVE-2016-2431	Blindly trust the system call parameters from TA, arbitrary memory write	
	CVE-2013-3051, CVE-2015-4422	No memory bound check, arbitrary memory write	
G3	CVE-2016-5349	Pointer sanitization failure, arbitrary memory read	Pointer sanitization based on non-falsifiable message format
	CVE-2016-8763, CVE-2016-8764	Pointer sanitization failure, arbitrary memory write	

imposed by the additional mode switch from TFence to the TEE after trapping and verifying the messages. However, we adhere to this design decision to minimize the increase in the TEE complexity and to prevent the creation of new attack surfaces.

Once the SMC invocation is trapped, TFence verifies several conditions: (1) It checks whether the current process invoking the SMC was registered through TFence hypercall. The value of the Translation Table Base Register (TTBR) is compared against the information registered during the initialization process. (2) TFence checks if the SMC is invoked by the shielded part by verifying the current value of the Virtualization Translation Table Base Register (VTTBR) that has a separate values for the shielded and non-shielded parts. (3) The parameters fed into the TEE should also be validated. TFence checks the parameters based on the current message format definition, information of which is delivered as the immediate value of the SMC instruction.

For the brevity of the explanation, we assume that the message format is the same as shown in Listing 1. R0 contains the requested service number (i.e., TA identifier) that is an integer value. TFence can validate it against the available service numbers in the TEE. R1 contains the virtual address of the request buffer. TFence converts the virtual address to the physical, and checks if it falls within the memory area of the shielded part (i.e., protected stack and heap) by using the tentative invariants. The virtual address of the response buffer to which the TA writes its response is held in R2. It is validated in a similar way by using invariants. Specifically, the translated physical address should not belong to one of the protected objects such as the code of the shielded part, the REE kernel, and the TEE. Finally, once all the conditions are validated, TFence updates the register values such that they reflect the physical address of the parameters and invokes SMC to switch the processor to Monitor mode.

Note that, although our example message format is very simple, there should not be any technical barrier to validate application-specific data structures in the request and response buffers. To this end, the definition of the precise message format is significant and would require an enhancement of the current TA investigation or development procedure. In addition, the format identifier should be imposed to ensure that it is tightly coupled with the TA identifier and each of its operations (or commands) to prevent malicious CAs from bypassing the procedure TFence uses for the validation and sanitization of pointers.

5 Case Study

In this section, we generalize the publicly disclosed TEE-related vulnerabilities and classify them into one of three groups as shown in Table 4. Additionally, we discuss the effectiveness of TFence to hinder the attacks that exploit the vulnerabilities.

Compromising TrustZone kernel driver (G1). Adversaries have exploited the vulnerabilities in the TEE-related drivers to escalate the privilege to kernel and thus freely send crafted messages to the TEE. For instance, both CVE-2014-4322 [47] and CVE-2015-4421 [8] exploit the vulnerability–absence of memory address bound check in the TrustZone driver–to compromise the kernel. Because TFence also requires part of the TrustZone driver to be migrated to the shielded part, the same vulnerability could be exploited in the par-priv mode. However, TFence confines the escalted privilege to the par-priv mode. This strictly restricts the allowed privileged operation to the direct communication with the trust anchors, whereas exploitation of the TrustZone kernel enables the adversary to fully control the REE. Moreover, the message verification conducted by TFence would still be effective even in the presence of the compromised OS.

Exploiting TEE vulnerability (G2). The vulnerabilities in the TEE (i.e., TA or TEE OS) have been exploited to compromise the TEE (G3 in Table 4). CVE-2015-6639 [9] exploits the buffer overflow vulnerability to compromise Qualcomm's TEE DRM service. CVE-2013-3051 [48] and CVE-2015-4422 [49] abuse the fact that a vulnerable TA does not conduct any memory bound-check for the input from the CA. CVE-2016-2431 [12] exploits the vulnerability of a TEE OS system call, which blindly trusts the parameters from any TA. The attack techniques exploiting these CVEs other than CVE-2016-2431 are similar in that they abuse some TA commands that are not supposed to be invoked by the legitimate CA. Hence, the attacks can be prevented by TFence enforcing a strict message verification policy established for each process (i.e., the CA). For example, TFence can define and use the list of allowed TAs and commands for each CA to fulfill the message verification. However, depending on the attack techniques, we also expect the current pointer verification by TFence to be sufficient to neutralize some CVEs. Note that since the CVE-2016-2431 exploits the interface between the malicious TA and TEE OS, TFence needs to prevent the first attack phase –compromising the TA– to incapacitate this attack.

BOOMERANG attack (G3). The TEE generally believes that the pointer values in the message from the REE are properly sanitized. Unfortunately, the sanitization is conducted based on the message format provided by the message sender (i.e., CA), which can be easily compromised; the adversary can provide an falsified message format to the REE sanitizer to stealthily manipulate one of pointers such

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

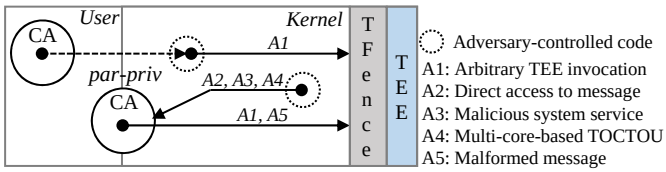

Fig. 7. Attacks against TEE with TFence.

that it represents the physical address of one of the REE kernel objects. Consequently, the adversary can abuse the TEE to attack the REE. By performing this confused deputy attack, namely BOOMERANG [5], CVE-2016-5349 [50] and CVE-2016-8762 [51] showed that the arbitrary memory read and the privilege escalation to root can be achieved without exploiting the REE vulnerabilities. TFence can effectively address the BOOMERANG for the following reasons: (1) message verification is performed based on the information queried from the TEE, which cannot be directly manipulated by the adversary (2) the pointers in the message are always regarded as the virtual address; thus, they are converted to physical address and sanitized by TFence before being delivered to the TEE.

6 Evaluation

6.1 TEE Attack and Defense

Real-world attacks against the TEE have abused the insecure communication channel to deliver a malicious payload to vulnerable trusted services as discribed in Section 5. As the baseline defense of the TEE, TFence restricts arbitrary messages from being fed into the TEE. In this section, we discuss the efficacy of TFence in terms of the TEE protection with several attack scenarios shown in Figure 7.

Attack 1. Arbitrary invocation of SMC instructions with kernel and par-priv modes. Those are blocked by TFence since the message senders are not registered by hypercalls and are thus not authorized.

Attack 2. Direct access to the shielded part memory by the kernel privileged process, which aims to manipulate the message. This attack is detected by TFence due to the stage-2 paging faults incurred by the access to the protected area.

Attack 3. Malicious system services [33]. We deployed the rootkit that hooks a memory allocator and reuse the already mapped memory for the new heap-memory allocation request from the shielded part. If this attack succeeds, the adversary can induce the shielded part to self-modify one of the protected objects. However, this attack was prevented by TFence because all the memory objects in the shielded part are maintained as tentative invariants, which can be checked whenever new memory is allocated.

Attack 4. Multi-core based TOCTOU attack. We exploit the fact that the client applications of the TEE-based crypto services (e.g., DRM and secure storage) open the encrypted file, copy the contents to the shared buffer, and send it to the TEE as part of the message to be processed by the invoked TA. We first modify the file content such that it presents the corrupted stack layout with malicious payload. When the shielded process requests new memory for the buffer, we inform the virtual address of the new allocation to another core through the inter-processor interrupt (IPI). Then, the core that receives the interrupt updates the page table of the client application to remap the new buffer with the protected stack, expecting the stack to be corrupted while the shielded part is running. However, the attack failed due to the invariant check mechanism of TFence that includes page-table mapping validation and is conducted with every mode switch (and SMC traps).

Attack 5. Malformed message delivery. We sent malicious messages to the TEE, which contain a pointer to the static region of the REE OS, and to the code and data of the shielded part as the addresses of the response buffers, respectively. This attack can lead the TEE to corrupt the objects pointed to [5], [6]. TFence prevented the attack based on the message verification mechanism that checks the message by leveraging the permanent (and tentative) invariants and message format definition of the currently invoked trusted application (TA).

Limitation. Although TFence provides a way to authenticate and protect the message, the effectiveness of the message verification depends on the correctness and concreteness of the message format definition, which needs to be explored further. In our prototype of TFence, only simple information such as the type (e.g., scalar or pointer) of each member of the message is described in the definition; hence, confused deputy attacks [5] can be prevented by TFence to a limited extent; unfortunately, it is possible for certain messages to be valid in terms of the message verification, but remain effective to exploit some vulnerabilities in the TEE. Besides, although we address the memory allocation-based Iago attack, other malicious system service-based attacks can lead the shielded CA to create and deliver a malicious message to the TEE. We aim to address these limitations in our future work.

6.2 Par-priv Mode Security Analysis

The adoption of par-priv mode should not introduce new attack surfaces to the system. However, since the par-priv process is designed to run in System mode that has kernel privilege, an adversary might attempt to exploit this to run malicious code as the par-priv process. Thus, to protect the REE from a malicious par-priv process, any attack that can be crafted with the kernel privilege should be considered and prevented by TFence.

The adversary abusing the par-priv mode could try to manipulate the kernel memory or dump the memory of other processes. To this end, the adversary would need to modify the page-table entries to map the target object in the memory or remove the protection attributes set in the page-table. Memory-bound attacks such as these are prevented as TFence isolates the par-priv process in the user memory domain by leveraging DACR. Any kernel object, including the page-table, is located in the kernel domain, which is isolated from the user domain to prevent a malicious par-priv process from accessing them.

The invocation of privileged instructions is also allowed in System mode. Thus, the adversary could attempt to manipulate DACR configuration, and thus to neutralize the domain separation. Apart from this, he could simply disable the MMU or the page-table based access-permission check

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

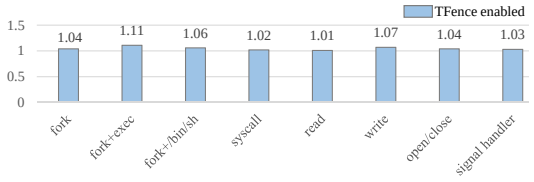

Fig. 8. LMBench results of TFence normalized to Linux. In most cases, the overhead is less than 7%.

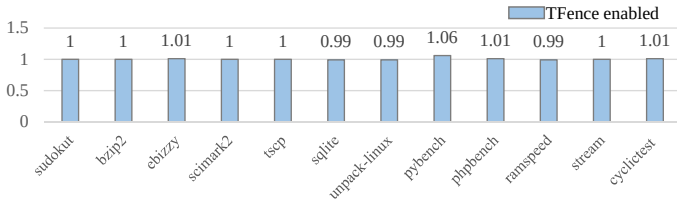

Fig. 9. Application benchmarks of TFence normalized to Linux. The result indicates that the maximum overhead imposed by enabling TFence is 6%.

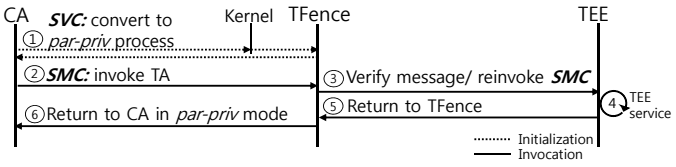

Fig. 10. CA with TFence removes kernel dependency when communicating with TEE.

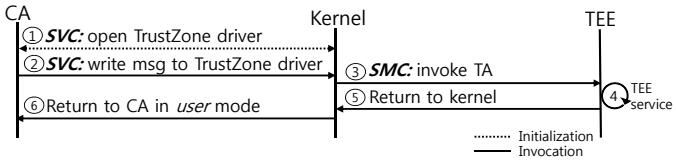

Fig. 11. Normal CA, which relies on kernel to communicate with TEE.

by configuring the SCTLR. However, TFence traps and prevents any attempt to perform these security-critical operations by leveraging hardware-assisted hypervisor traps. With the ability to directly communicate with the trust anchors, the malicious par-priv process might try to perform a brute force attack by sending arbitrary messages to the TEE. This is also prevented since TFence traps and verifies all the messages bound for the TEE by configuring the hypervisor traps for SMC invocation and using the pre-defined message formats.

The aforementioned protective measures would need to be ensured by timely triggering TFence at every switch between the par-priv and kernel modes. To this end, the transition gates also require protection. Although the exception vector that invokes hypercalls when the mode switches to kernel is mapped in the user domain, it is mapped with read-only access permission for both the user and kernel. Access to the page-tables and privileged instructions that can be exploited to relocate the exception vector is also restricted by TFence. The return_to_user also invokes hypercalls when the mode switches from kernel to par-priv mode. Being mapped in kernel domain, its access by the par-priv process is obviously prevented. Finally, as shown in the BOOMERANG attack, the adversary can abuse the TEE to incapacitate the transition gates patched in the kernel. However, this attack is also hampered by the message verification process that checks each pointer-type member against invariants maintained by TFence.

6.3 Performance

In this section, we analyze the performance overhead imposed on the REE OS and the CA, as incurred by TFence.

6.3.1 REE OS

We measured the overhead imposed on the REE OS by running LMBench [52] and Phoronix Test Suite [53].

Microbenchmarks. Adopting TFence incurs performance overhead in the following respects: (1) enabling the stage-2 paging (2) execution of the transition gates on each exception occurrence. Figure 8 shows the result of running LMBench. We ran each case 100 times for both Linux and TFence-enabled Linux, and estimated the overhead based on the average latency from each case. The overhead is normalized to Linux. Most operations caused less than 7% of overhead except fork+exec, for which it is approximately 11%.

Application benchmarks. Figure 9 summarizes the normalized overhead obtained with the Phoronix Test Suite, which was relatively smaller than that observed with LMBench. In most cases, the overhead for each test was less than 1% regardless of the test type. Exceptionally, pybench that performs system bound tests and estimates the average performance results of Python showed 6% overhead, which is the worst result achieved for our application benchmarks. Compared to other test cases, we suspect that pybench performs highly memory-intensive operations that require frequent page-table walks and TLB flushes.

6.3.2 Client Application with TFence

In this section, the performance of SMC and HVC invocations, and the overall performance of CAs that leverages TFence to secure communication were measured and analyzed.

Performance of communication with TA. In contrast to a normal CA (Figure 11), a CA with TFence (Figure 10) does not depend on the TrustZone kernel driver. We measured the elapsed time for communication between a CA and TA in both cases. For a normal CA, we created two versions of TrustZone kernel drivers: a character device driver and proc file. The drivers only contain primary functions such as the *ioctl* or *write_proc* handlers, and virtual-to-physical memory translation. The *ioctl* and *write_proc* handlers simply copy the messages from the CA by using *copy_from_user*, configure general registers for the parameters, and invoke SMC. As creation of a message copy is unnecessary for a CA with TFence, the CA simply configures parameters and invokes the SMC.

Table 5 summarizes the result of the experiment. A CA with TFence outperforms normal CAs in all aspects regardless of the kernel interface types used for SMC invocation. In the initialization phase, a CA with TFence was significantly faster than a normal CA (improved 83.5% against opening

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2840709, IEEE Transactions on Dependable and Secure Computing

TABLE 5
Performance of communication between CA and TA.

Type	Normal CA (proc file)		Normal CA (char device)		TFence CA w/o Trap		TFence CA w/ Trap	
Initialization	open(proc_tzdrv)	112.6μs	open(char_tzdrv)	54.5μs	convert_par_priv	9.0μs	convert_par_priv	9.0μs (-83.5%)*
Invocation	write_proc	25.7μs	ioctl	16.2μs	SMC	7.7μs (-52.5%)†	SMC w/ trap	15.8μs (-2.5%)†

* Against open(char_tzdrv), † Against ioctl

char_tzdrv) due to the simplicity of the operation required for par-priv process creation. In particular, the latency of par-priv process creation involves elapsed time for configuring the SPSR to System followed by the interposition of TFence to partially restict the privilege of the CA.

For TA invocation, direct SMC execution with a TFence trap outperforms a normal CA that uses the *ioctl* interface with 2.5% performance improvement. Note that the use of *ioctl* includes the latency for message copying, memory translation, and mode switches between the user, kernel, and Monitor, whereas the SMC with traps includes the time consumed for trapped message verification, memory translation, and re-invocation of the SMC, and the round trip latency between the par-priv, hypervisor, and Monitor modes. In our evaluation, message verification was performed as described in Section 4.4.3. We expect the performance of SMC with TFence to be fluctuant depending on the verification policy and the complexity of the message format to be checked.

Performance of hypercall. We compared the hypercall performance by creating a hypercall that simply invokes TFence but immediately returns to the previous mode that invoked the call. For a normal CA, we prepared two versions of hypercall invocation that are performed by (1) a new system call and (2) device drivers (character device and proc file). Each driver provides *ioctl* and *read_proc* interfaces to the normal CA to execute the HVC instruction. On the other hand, a CA with TFence does not require any kernel component for hypercall invocation. Thus, (3) we directly execute the hypercall in a CA with TFence.

We ran each case 100 times and evaluated the average latency. For (1), we directly executed the SVC instruction with the new system-call number in a normal CA. The device drivers for (2) are simpler than the TrustZone kernel drivers used for the SMC performance evaluation. Particularly, the drivers directly execute the hypercalls designated for this evaluation without copying any parameter from the CA.

The results are presented in Table 6. The hypercall invocation with TFence outperforms the two other cases owing to the removal of kernel dependencies. The worst performance was observed for the case with the kernel driver. This is because the time complexity of the run with the kernel drivers and standard libc functions (e.g., *read*) is the highest among the three cases. Specifically, interaction with the kernel through the *read_proc* interface results in higher latency than that with *ioctl*. However, depending on the library and system call implementations, the latency and the performance of hypercalls can be varied.

6.3.3 Open Source TEE with TFence

To evaluate the end-to-end overhead of securing the communication channel, we applied TFence to the open source TEE software–SierraTEE [54]. The TrustZone driver

TABLE 6
Hypercall performance comparison.

Type	System call	Kernel driver		TFence hypercall
		read_proc	ioctl	
Invocation	9.1μs	26.7μs	11.5μs	7.5μs (-17.6%)†

† Against system call

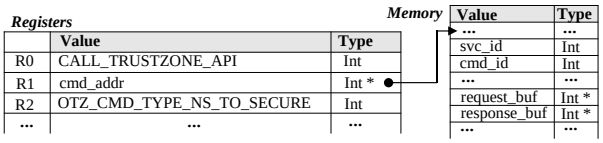

Fig. 12. Message format used in SierraTEE.

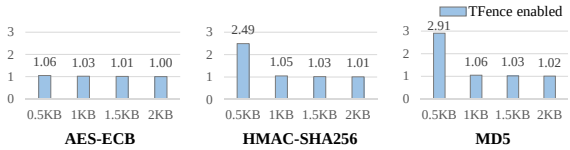

Fig. 13. Crypto service overhead with TFence normalized to SierraTEE.

(*otz_client.ko*) was isolated and executed as part of the shielded part in par-priv mode, which required some kernel-level functions to be removed (e.g., *ioctl* and *copy_from_user*) or replaced with user-level functions (e.g., replacement of *kmalloc* with *calloc*). In addition to the TrustZone driver, the TEE APIs (*otz_api.o* and *otz_TEE_api.o*) and wrapper functions to invoke them were also isolated in the shielded part. Besides, the message verification was performed based on the SierraTEE message format (Figure 12).

Table 7 presents the lines of code for implementing TFence and a shielded process that invokes SierraTEE crypto services. Particularly, the LOC for TEE_kernel_driver indicates changes to the original code of SierraTEE kernel driver and APIs. In our work, we manually performed the program analysis and separation, which could be erroneous depending on the program behavior and complexity (e.g., user input processing). Several works [55], [56], [57] propose automatic methods for program separation and verification. We will explore the feasibility of coordination between TFence and those systems.

The performance overhead was measured by using a Sierra TA that provides several crypto services: AES, HMAC, and message digest (MD5). The size of the input text varies from 0.5 KB to 2 KB. The CA prints out the input text and the output of the crypto operations on each run. Figure 13 shows the result of our experiment. The maximum overhead was approximately 191% for the smallest input size (0.5 KB) with MD5. However, the overhead

TABLE 7
LOC for applying TFence to SierraTEE.

Components		Language	LOC
Non-shielded	TFence_API	ASM	25
	crypto_service_call	C	137
Shielded	TFence_API	ASM	21
	TEE_kernel_driver	C	361
TFence	Hypervisor	ASM + C	1823 + 74
	Kernel patch	ASM	63

dramatically decreased as the input size increased in all cases. Thus, for an input size of 2 KB, the overhead became negligible. According to our analysis, some overhead was always added to the CA with TFence due to the additional operations such as calculating and comparing the hashes of the pages, and creating stage-2 page mapping. However, this constant latency was amortized when the overall runtime of the CA was sufficiently large.

7 Discussion

7.1 Alternatives for Direct Communication

As an alternative means of realizing direct communication between an application and trust anchors, we can consider leveraging the hypervisor trap instead of System mode. For instance, configuring the Hyp Configuration Register (HCR) enables some instructions (e.g., DC ZVA) invoked in user mode to be trapped in the hypervisor. The advantage of this alternative approach might be minimizing the number of kernel patches and the overhead for the par-priv mode configuration (note that the application shielding is still required).

However, we expect an increase in the development complexity of hypervisor compared with the current design because of the following reasons. The alternative approach requires more hypervisor logics to distinguish and emulate the trapped instruction to support both the original and new functionalities of the instruction. Furthermore, the availability of the instructions is OS-dependent because the user-mode accessibility of certain instructions can be configured by the OS kernel. This might lead to another trapping of control instructions to restrict and emulate the OS behavior. We will further explore alternative approaches for TFence enhancement.

Message encryption can also be considered as a possible solution to protect the communication channel without creating a par-priv process. However, this approach additionally requires ensuring the confidentiality of the shielded part to protect the crypto logic and keys. Moreover, the confidentiality requirement might result in placing additional functions such as (un)marshaling of parameters transferred between the shielded and non-shielded parts, which was not necessary in our approach, which only needs to guarantee the integrity of a message.

7.2 Compatibility with 64-bit Processor

Some hardware features (e.g., HSTR and DACR) leveraged by TFence are not available in the 64-bit ARM architecture (AArch64). However, porting TFence onto AArch64 remains feasible because there are alternatives to the deprecated features. Instead of HSTR, we could use HCR, which also enables security-critical system operations to be trapped. For memory isolation between the user and kernel, we can use Translation Table Base Registers (TTBRs) and an Address Space Identifier (ASID) to replace the operation of DACR. That is, since the user and kernel memory are naturally separated by using two TTBRs (i.e., TTBR0 and TTBR1 for the user and kernel, respectively) on 64-bit Linux, we can manipulate TTBR1 and ASID to remove the kernel mapping without flushing TLBs whenever par-priv mode is entered.

7.3 Performance Optimization

TFence consistently imposes overhead on the overall system owing to the hypervisor activation, which is around 6% with LMBench. We expect this overhead to be possibly addressed by dynamically enabling and disabling TFence based on the existence of the par-priv process. More specifically, when the last par-priv process is terminated, we can disable the stage-2 paging. This disabling might expose TFence to an adversary that directly accesses the physical memory. However, as shown in [58], we can utilize TrustZone technology, TZASC [59], to protect hypervisor-related memory (i.e., TFence) when the stage-2 paging is disabled. In addition, to dynamically (de)activate the transition gate that is inserted in each exception handler, we can leverage the Vector Base Address Register (VBAR), which enables the exception vector to be remapped to the address specified in the VBAR. In particular, we can map the patched exception vector for TFence only when there exists a par-priv process. The performance optimization of TFence will be addressed in our future work.

7.4 TEE Security for IoT Device

According to the new ARM architecture design for the microcontrollers (ARMv8-M [60]), the optimized version of TrustZone will be available for low-power devices as well. For efficiency purposes, ARMv8-M exempts the Monitor mode and SMC instruction, and enables the domain switch between the REE and the TEE to be performed in a more direct way by introducing a Secure Gateway (SG) instruction (note that this is confined to low-power devices; thus, high-end devices such as mobile phones would continue to use the conventional version of TrustZone). We expect attacks that exploit the arbitrary domain switch and the trusted service invocation to continue to be possible even with the new design. However, due to the resource constraint (e.g., ARMv8-M neither contains an MMU nor a hypervisor), a different approach would have to be found to enhance the TEE security. In future, we could explore the attack vectors on these devices to devise efficient defense mechanisms.

8 Conclusion

We proposed a new mechanism for accessing TEE services, which lessens an adversary's opportunities to compromise the TEE. As future works, we intend to further explore the enhancement of the message verification mechanism to defeat the attempt to exploit the potential vulnerabilities in the TEE. In addition, we will validate the compatibility of

our proposal with TEE standards, and optimize the design of TFence to bridge the gap.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. NRF-2017R1A2B3006360) and the Office of Naval Research (ONR) TPCP program.

REFERENCES

[1] (2017, May) Android for work on samsung knox devices. [Online]. Available: https://kp30.s3.amazonaws.com/0b2e7bf6ffc167b609daed41001d00e9.pdf

[2] (2017, May) Credential storage enhancements in android 4.3. [Online]. Available: http://nelenkov.blogspot.ch/2013/08/credential-storage-enhancements-android-43.html

[3] (2017, May) Discretix. [Online]. Available: https://www.trustonic.com/use-cases/mobile-financial-services

[4] (2017, May) Securing the future of authentication with arm trustzone-based rusted execution environment and fast identity online (fido). [Online]. Available: https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf

[5] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17), San Diego, CA*.

[6] (2017, May) War of the worlds - hijacking the linux kernel from qsee. [Online]. Available: http://bits-please.blogspot.kr/2016/05/war-of-worlds-hijacking-linux-kernel.html

[7] (2017, May) A software level analysis of trustzone os and trustlets in samsung galaxy phone. [Online]. Available: https://www.sensepost.com/blog/2013/

[8] (2017, May) Cve-2015-4421. [Online]. Available: https://firmwaresecurity.com/tag/cve-2015-4421/

[9] (2017, May) Cve-2015-6639. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6639

[10] (2017, May) Cve-2015-6647. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6647

[11] (2017, May) Cve-2016-0825. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-0825

[12] (2017, May) Cve-2016-2431. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2431

[13] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

[14] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 265–278, 2013.

[15] (2017, May) Arndale board. [Online]. Available: http://www.arndaleboard.org/wiki/index.php

[16] (2017, May) Drm agent for embedded deployment. [Online]. Available: https://www.insidesecure.com/Markets-solutions/Content-Protection-and-Entertainment/DRM-Agent-for-Embedded-Deployment

[17] H. Sun, K. Sun, Y. Wang, and J. Jing, "Trustotp: Transforming smartphones into secure one-time password tokens," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 976–988.

[18] W. Li, H. Li, H. Chen, and Y. Xia, "Adattester: Secure online mobile advertisement attestation using trustzone," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 75–88.

[19] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "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, 2014, pp. 90–102.

[20] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," 2014.

[21] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones," in *Computer Security-ESORICS 2014*. Springer, 2014, pp. 202–218.

[22] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 33–49. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ning

[23] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, "On-board credentials with open provisioning," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM, 2009, pp. 104–115.

[24] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 541–556. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua

[25] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable and Secure Computing*, 2016.

[26] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "Secret: Secure channel between rich execution environment and trusted execution environment," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15), San Diego, CA*.

[27] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "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, 2014, pp. 67–80.

[28] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 365–378.

[29] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *Security and Privacy, 2016. SP 2016. IEEE Symposium on*. IEEE, 2016.

[30] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2. ACM, 2008, pp. 2–13.

[31] Y. Cheng, X. Ding, and R. Deng, "Appshield: Protecting applications against untrusted operating system," *Singaport Management University Technical Report, SMU-SIS-13*, vol. 101, 2013.

[32] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1607–1619.

[33] S. Checkoway and H. Shacham, "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 253–264.

[34] "Architecture reference manual (armv7-a and armv7-r edition)," *ARM DDI C*, vol. 406, 2008.

[35] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

[36] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 1–20.

[37] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.

[38] (2017, May) Arm system mode. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/Caccfegc.html

[39] W. Arbaugh, D. J. Farber, J. M. Smith *et al.*, "A secure and reliable bootstrap architecture," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997, pp. 65–71.

[40] (2017, May) Corelink system memory management unit.

[Online]. Available: http://www.arm.com/products/system-ip/controllers/system-mmu.php
[41] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "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*, ser. ASPLOS '15. ACM, 2015, pp. 177–189.
[42] (2017, May) Unlocking the motorola bootloader. [Online]. Available: http://bits-please.blogspot.kr/2016/02/unlocking-motorola-bootloader.html
[43] (2017, May) Amd-v nested paging. [Online]. Available: http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf
[44] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 558–569.
[45] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *Security and Privacy, 2016. SP 2016. IEEE Symposium on*. IEEE, 2016.
[46] (2017, May) android kernel scm. [Online]. Available: https://android.googlesource.com/kernel/msm/+/android-5.1.0_r0.6/arch/arm/mach-msm/scm.c
[47] (2017, May) Cve-2014-4322. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2014-4322
[48] (2017, May) Cve-2013-3051. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2013-3051
[49] (2017, May) Cve-2015-4422. [Online]. Available: http://www.huawei.com/en/psirt/security-advisories/2015/hw-432799
[50] (2017, May) Cve-2016-5349. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-5349
[51] (2017, May) Cve-2016-8762. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-8762
[52] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.
[53] (2017, May) Phoronix test suite. [Online]. Available: http://www.phoronix-test-suite.com/?k=home
[54] (2017, May) Sierraware. [Online]. Available: http://www.openvirtualization.org/
[55] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments." in *NSDI*, vol. 8, 2008, pp. 309–322.
[56] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, 2004, pp. 57–72.
[57] S. Liu, G. Tan, and T. Jaeger, "Ptrsplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2359–2371.
[58] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.
[59] (2017, May) Arm security technology - building a secure system using trustzone technology. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf
[60] (2017, May) Whitepaper - armv8-m architecture technical overview. [Online]. Available: https://community.arm.com/docs/DOC-10896

Jinsoo Jang is a postdoctoral fellow at Korea Advanced Institute of Science and Technology (KAIST). He received his Ph.D. and M.S. in Information Security from KAIST in 2018 and 2014, and B.S. in Information and Computer Engineering from Ajou University in 2007. His research interest includes system security, particularly in the trusted execution environment (TEE).

Brent Byunghoon Kang is currently an associate professor at the GSIS (Graduate School of Information Security) at KAIST (Korea Advanced Institute of Science & Technology). Before KAIST, he has been with George Mason University as an associate professor in the Volgenau School of Engineering. Dr. Kang received his Ph.D. in Computer Science from the University of California at Berkeley, and M.S. from the University of Maryland at College Park, and B.S. from Seoul National University. He has been working on systems Security area including OS kernel integrity monitor, trusted execution environment, hardware-assisted security, botnet malware defense, and DNS analytics. He is currently a member of the IEEE, the USENIX and the ACM.