PrivateZone: Providing a Private Execution Environment using ARM TrustZone

Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang, *Member, IEEE*

Abstract—ARM TrustZone is widely used to provide a Trusted Execution Environment (TEE) for mobile devices. However, the use of TrustZone is limited because TrustZone resources are only available for some pre-authorized applications. In other words, only alliances of the TrustZone OS vendors and device manufacturers can use TrustZone to secure their services. To help overcome this problem, we designed the PrivateZone framework to enable individual developers to utilize TrustZone resources. Using PrivateZone, developers can run Security Critical Logics (SCL) in a Private Execution Environment (PrEE). The advantage of PrivateZone is its leveraging of TrustZone resources without undermining the security of existing services in the TEE. To guarantee this, PrivateZone creates a PrEE using a memory region that is isolated from both the Rich Execution Environment (REE) and TEE. In this paper, we describe the design and implementation of PrivateZone. The prototype of PrivateZone was implemented on an Arndale board with a Cortex-A15 dual-core processor. We built PrivateZone by exploring both security and virtualization extensions of the ARM architecture. To illustrate the usage and the efficacy of PrivateZone, we developed an Android application based on PrivateZone framework, and evaluated the performance overhead imposed on the OS in the REE and SCLs in the PrEE.

Index Terms—Mobile Device Security, Trusted Execution Environment, ARM TrustZone.

1 Introduction

ARM TrustZone is widely adopted as a means of providing a Trusted Execution Environment (TEE) for mobile and embedded devices, and is utilized to protect security-critical assets such as crypto-keys, payments, and DRM services [1], [2], [3]. Through hardware-based access control, TrustZone isolates security-critical services from the Rich Execution Environment (REE) that hosts general OSes such as Linux and Android. However, TrustZone is regarded as a premium, closed resource because only a limited number of partners (device manufacturers and TrustZone OS providers) can deploy the TEE services. This is because the TEE security level and trustworthiness is maintained by adopting restrictions where only strictly verified applications can be deployed in the TEE.

To allow universal access to the security resources provided by TrustZone and make the benefits of TEE available to general applications, we designed "PrivateZone," a framework whereby individual developers can utilize an isolated execution environment for application development. Specifically, with PrivateZone, developers can protect and execute Security Critical Logic (SCL) in an isolated execution environment, called the Private Execution Environment (PrEE). Developers are thus expected to define the SCL as part of their application and deploy it in the PrEE. Upon running the application, the SCL in the PrEE can be invoked for private execution (i.e., securely run the SCL without exposing sensitive information to the REE or other SCLs in the PrEE).

Since PrivateZone was developed to make the security resources publicly accessible, it presents new challenges. In the previous trusted execution environment, a trusted service (trustlet) was assumed to be non-malicious, and the TEE was designed to protect the services from compromised or malicious REE applications. However, this assumption is no longer valid with PrivateZone, since an arbitrary piece of code produced by the developers should be allowed to run within the PrEE. Thus, PrivateZone should be designed to satisfy the following security requirements.

S1. **PrEE Protection:** The PrEE should be isolated from the REE. This prevents attackers in the REE from accessing the SCLs in the PrEE.

S2. **REE Protection:** Any object in the REE should also be protected from attacks originating in the PrEE. This is because PrivateZone allows arbitrary code execution in the PrEE.

S3. **TEE Protection:** The adoption of PrivateZone should not increase the attack vectors for compromising the existing services in the TEE. Hence, both the REE and TEE should be isolated from the PrEE.

S4. **SCL Isolation:** Finally, since PrivateZone cannot verify the SCLs deployed in the PrEE in advance, SCLs should be isolated from each other in the PrEE. Also, SCLs should only be allowed to run with the lowest privilege in the PrEE.

Besides satisfying the above security requirements, PrivateZone simultaneously plays major roles in creating PrEE and in securely managing context switches across environments while an application is running. First, to create the PrEE, PrivateZone utilizes a minimal number of features available in the ARM virtualization extensions. By using the stage-2 memory translation table in ARM (similar to EPT in Intel), PrivateZone can isolate the PrEE memory from the

• *J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi and B. Kang are with Korea Advanced Institute of Science and Technology.*
E-mail: {jisjang, zpzigi, jhl9105, nhkwak, augustus92, yschoi46, brentkang}@kaist.ac.kr

REE, which satisfies the S1 (PrEE Protection) requirement. Since the memory for the PrEE is located outside the TEE, the PrEE and TEE are separated by a hardware-based protection mechanism such as TrustZone Address Space Controller (TZASC) [4], which satisfies the S3 (TEE Protection) requirement. Also, S2 (REE Protection) is satisfied because PrivateZone does not allow direct access from the PrEE to the REE. Any communication between the environments should be interposed and verified by PrivateZone. Last, to satisfy the S4 (SCL Isolation) requirement, PrivateZone isolates SCLs from each other in the PrEE by using the restricted page-table mappings and memory protection attributes of the ARM processor.

To realize trustworthy execution, previous works [5], [6], [7], [8] isolated applications from an untrusted OS by using x86 virtualization. Similarly, PrivateZone also isolates the SCLs from attackers in the REE. *However, PrivateZone targets the security of mobile devices considering the extension of TrustZone functionalities (e.g., secure storage, trusted display and keypad) in the PrEE*. To this end, PrivateZone explores the coordination of the ARM's security extensions (i.e., TrustZone) and virtualization extensions. For the provision of the TEE services in the PrEE, PrivateZone capitalizes on ARM's security extensions. The virtualization extensions are utilized for the PrEE creation, but we exclude the hypervisor implementation to simplify the design of PrivateZone and to minimize the performance impact on the REE OS (i.e., mobile device's baseline performance), as incurred by stage-2 page-table management.

Also, without using virtualization, TLR [9] and ObC [10] enable the protection of the security sensitive logic of applications by TrustZone. However, they do not support PrEE and it is assumed that the developed code is deployed inside the TEE. On the contrary, PrivateZone uses the memory outside the TEE to securely execute arbitrary SCLs, without hampering the security of the existing TEE services. Intel also publicly provides SGX [11] for the secure execution of applications. Unfortunately, at the time of writing, SGX is not available on most mobile devices and Intel's plan for licensing SGX is uncertain [12], [13].

We implemented a prototype of PrivateZone on an Arndale board with the Exynos5250 system-on-a-chip (SoC) using a Cortex-A15 dual-core processor. This ARM processor integrates both virtualization and security extensions. Linaro-Android (13.10) and SierraTEE [14] were employed as the OS in the REE and TEE, respectively. To demonstrate PrivateZone, we created an Android-NDK shared library that utilizes PrivateZone to deploy and execute SCLs in the PrEE. To evaluate the overhead on the REE OS, we ran LMBench [15] and Phoronix test suite [16]. The performance degradation of applications that leverage PrivateZone was also measured and analyzed.

Our contributions are as follows:

- We introduce PrivateZone to create a Private Execution Environment (PrEE) on a mobile device. PrivateZone aims to openly provide a PrEE to developers without undermining the security of existing services in the TEE.
- PrivateZone enables the extension of TrustZone functionalities and TEE services in the PrEE. Also, we demonstrated PrivateZone's compatibility with a mobile OS by creating an Android application that can take advantage of PrivateZone.
- We provide in-depth design considerations and share our implementation experience acquired by exploring the security and virtualization extensions on the ARM architecture.

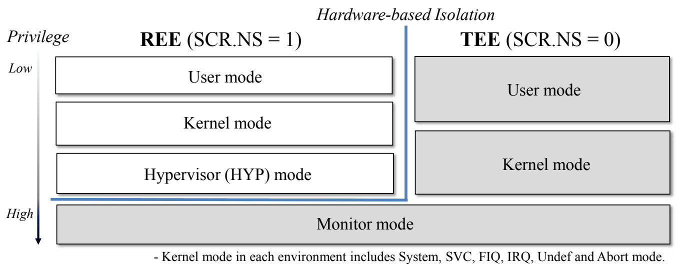


Fig. 1. Environments, modes, and privilege levels with virtualization extensions and security extensions of ARM architecture.

In the next section we review ARM's security and virtualization extensions. Section 3 describes the threat models and assumptions, and Section 4 presents the design considerations of PrivateZone. In Section 5 and 6, we show the implementation details and evaluate the performance of our prototype. We discuss the limitations of PrivateZone in Section 7, the related works in Section 8, and the conclusion in Section **??**.

2 Background

Essentially, PrivateZone is based on the ARM security extensions. However, it also uses features provided by the virtualization extensions. Hence, we provide background for both extensions mainly with the key features adopted in PrivateZone.

2.1 ARM Security Extensions

Recent ARM processors support a security extension known as TrustZone. System designers can utilize several TrustZone-based components such as the TrustZone Protection Controller (TZPC) and TrustZone Address Space Controller (TZASC) [4] to logically separate the system into two environments – the REE and TEE. For example, to build a secure IO path, a designer can configure TZPC to dynamically assign peripherals such as a keypad and display to one of the environments. Not only peripherals, but also system memory can be split into two environments by configuring the TZASC. Once the TEE is configured, pre-authorized services such as Digital Right Management (DRM) can be deployed and executed in the TEE.

ARM also introduces a Monitor mode to handle switches between the REE and TEE. Monitor mode saves and restores the context of each environment during the switches. Monitor mode accesses both environments by configuring the non-secure (NS) bit in the Secure Configuration Register (SCR). To enter Monitor mode, the process in each environment explicitly invokes a Secure Monitor call (SMC) with kernel privileges. Monitor mode has the highest privilege within the system, so can access any system configuration register, even if it is dedicated to a specific mode such as hypervisor.

2.2 ARM Virtualization Extensions

ARM also supports hardware-based virtualization for recent high-end processors. ARM uses hypervisor (HYP) mode to trap and manage events from guest VMs. Although HYP mode does not belong to the TEE, it still has a higher privilege than kernel mode in the REE. In addition to HYP mode, ARM also adopts stage-2 paging to virtualize the memory of guest VMs, which is analogous to the x86's nested paging [17]. This translates an Intermediate Physical Address (IPA) of a guest VM to a real physical address of the device.

Stage-2 paging requires the configuration of several virtualization-related registers such as VTTBR and HCR. The Virtualization Translation Table Base Register (VTTBR) holds the page table base address for stage-2 paging. By configuring the Hyp Configuration Register (HCR), stage-2 paging can be enabled (or disabled). Additionally, HCR defines the guest VM's events that are trapped in HYP mode. Translation Table Base Register (TTBR) updating and configuring a cache in the guest VMs are examples of such events. Normally, these registers are configured by the hypervisor running in HYP mode. They can also be accessed and configured in Monitor mode, however. Figure 1 describes the environments, modes, and privilege levels for the ARM architecture, including both the security and virtualization extensions.

3 Assumptions

We did not assume hardware attacks such as reverse-engineering the internals of a device by using hardware such as JTAG debuggers and logic probes. However, attackers still have access to the REE software stacks. For instance, an attacker can tamper with user applications and the OS kernel in the REE.

We can, however, trust hardware-based security components such as TrustZone. Secure boot [18] verifies the integrity of images loaded into the TEE. Also, objects in the TEE are protected by TrustZone and can be assumed to be non-malicious. In addition, IOMMU [19] restricts the malicious DMA. We assume that the device has a unique public/private key pair. The private key is protected in the e-FUSE, and is accessible in the TEE only. The public key is available to the device owner and application developers. Finally, denial-of-service (DoS) attacks, such as not scheduling protected applications, are exempted from our attack model.

4 Design of PrivateZone

4.1 Overview

PrivateZone provides a private execution environment (PrEE) to developers. Thus, the creation of the PrEE and guaranteeing the seamless execution of SCLs in the PrEE are the main roles of PrivateZone. To create the PrEE, PrivateZone utilizes ARM virtualization extensions. During a secure boot, PrivateZone creates two stage-2 page tables for the PrEE and the REE. Entries for each page table are completed in advance during boot time. As shown in Figure 2, the boot process creates three logical environments – REE, TEE, and PrEE.

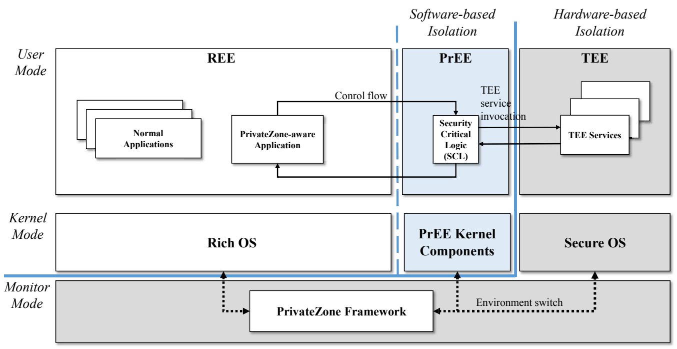


Fig. 2. PrivateZone provides an additional execution environment, PrEE, to securely execute SCLs. The PrEE is created by leveraging the ARM virtualization and security extensions. The main framework of PrivateZone is in Monitor mode to manage switching between environments.

By using PrivateZone libraries, developers can deploy the SCL in the PrEE and leverage TrustZone functionalities. For the SCL deployment, PrivateZone copies the SCL from the REE and then calculates the hash of the SCL and validates it by using the developer-provided signed hash. The valid SCL is deployed in the PrEE and invoked during the runtime of the corresponding application. Because the PrEE is logically separated from the REE by stage-2 paging, the SCL execution in the PrEE requires the intervention of PrivateZone. Therefore, once the SCL is invoked, PrivateZone saves the REE context and switches the environment to the PrEE. It also configures the PrEE context to execute the invoked SCL.

4.2 Architecture of PrivateZone

Although PrivateZone and TrustZone share the goal of providing an isolated execution environment, there are differences between the architectures, as shown in Table 1. In this section, the architecture of PrivateZone is described in terms of these differences.

4.2.1 *Isolation guaranteed by PrivateZone*

The isolation ensured by PrivateZone can be explained from two aspects:

(1) Inter-Environments: REE and PrEE are statically separated by stage-2 page tables. PrivateZone creates stage-2 page-tables for each environment and fills every entry of the tables in advance, during the boot. Once all the entries are filled, PrivateZone sets VTTBR to the address of the REE's stage-2 page-table and configures the VM flag in the HCR. From this point, only the REE is in the active state. Since the stage-2 page-table entries of the two environments are exclusive, there is no physical memory is shared by the REE and PrEE. The TEE is physically isolated from the REE and PrEE by a hardware-based access-control mechanism such as TZASC. As a result, neither the REE or PrEE can directly access the memory in the TEE.

(2) Inter-SCLs: In TrustZone's usage model, services are verified before being deployed in the TEE. However, in

TABLE 1
Comparison of PrivateZone and TrustZone

	PrivateZone	**TrustZone**
Isolation	Software-based: stage-2 paging	Hardware-based: TZASC
Service deployment	Runtime loading of SCLs	- Part of firmware upgrade procedure - Runtime loading of signed binaries
Access permission of deployed services	Strictly limited to the PrEE	Entire physical memory of device (depends on the TZASC configuration)
Communication between environments	Through copying data	Through shared memory
Interrupt source	Shares IRQ with the REE	Independent source (e.g., FIQ, depends on the configuration)
Security of services	Reliance on the strict isolation between SCLs	Verification before deployment
Ownership	Developers & device owners	TEE OS providers & device manufacturers

PrivateZone, we cannot apply this strategy because PrivateZone aims to open the framework to any application. Therefore, even malware can run in the PrEE. To protect benign SCLs from malicious code, PrivateZone isolates each SCL in the PrEE by performing memory management for every SCL. For example, it ensures that there is no shared-memory mapping between SCLs. Every memory page for SCL is assigned with the Privileged Execute Never (PXN) flag set to prevent the SCLs from executing security-critical instructions.

4.2.2 *Switch between Environments*

PrivateZone uses an SMC instruction to switch between the REE, PrEE, and TEE environments. Because the SMC instruction is not available in user mode, we inserted SMC instructions into several points in the REE and the PrEE. For the deployment and revocation of SCLs, we created a kernel driver that invokes SMC instructions. For the runtime invocation of SCLs, we patched the SCL's entry point with the SVC instruction that carries a specific immediate value (e.g., SVC #0xdeadbf). Consequently, the SVC exception handler invokes SMC if the immediate value of the current SVC exception indicates the SCL invocation.

Alternatively, we could implement PrivateZone in HYP mode. As addressed in previous works [5], [7], [8] that leveraged the x86's virtualization technologies, we can isolate the SCL by manipulating the stage-2 page table entries. Also, by using the stage-2 paging faults resulting from accesses to unmapped memory in the stage-2 page table, we can switch between the PrEE and REE environments.

Although our approach requires a few changes to the REE OS, it is still superior to implementing PrivateZone in HYP mode. That is, there is no performance overhead in the REE OS, as incurred by a stage-2 page-fault. In our approach, as adopted by NoHype [20] for memory partitioning, once the stage-2 page tables are configured at boot time, the tables are never updated while the device is running.

4.2.3 *Communication between Environments*

Although it depends on the implementation, in TrustZone's usage model, TEE services can use the environment-shared memory to communicate with clients in the REE [4]. In our approach, however, PrivateZone isolates each environment, so the REE and PrEE cannot share memory. Instead, PrivateZone provides communication between the environments. As shown in Section 4.2.4, PrivateZone copies the memory between the REE and the PrEE for private execution.

This feature could be exploited by attackers in the REE. An attacker could use PrivateZone to manipulate the OS in the REE by passing the critical kernel component addresses as an SCL parameter. Although we must assume that an attacker has full access to the REE OS, this situation should nevertheless be prevented. Thus, before performing memory operations, PrivateZone checks whether the virtual address passed as a parameter is present and falls within the user-memory range. Mapping between the virtual and physical addresses is also verified.

4.2.4 *SCL Calling Convention*

Due to the strict isolation between the REE and the PrEE, the parameters should be passed into the SCL by PrivateZone. Parameter passing can be performed by applying the following procedure.

(1) Developers register the number of parameters, as well as the type/size of each parameter during SCL deployment
(2) When the SCL is invoked, PrivateZone checks the registered information for the parameters
(3) If the input parameter is of an integer type, PrivateZone copies it from the REE
(4) If the type is pointer, PrivateZone first maps the pointed memory to the TEE and copies the value from the REE, based on the size information. It also allocates new memory to the PrEE and places the copied value in memory. Finally, it creates mapping for the newly allocated memory in the SCL's page table

Assuming that the application is compiled based on the procedure call standard for the ARM architecture (AAPCS) [21], the first four parameters are passed to registers (R0-R3) and excess parameters are placed on the stack. Thus, depending on the number of parameters, PrivateZone should update both the registers and stack for the PrEE.

The above procedure is repeated until the parameters are copied from the REE. To simplify the procedure, PrivateZone provides a parameter encoding/decoding library. This allocates page-aligned memory and places each parameter's type, size, value and the total number of parameters in the memory. If the allocated memory is insufficient, however, the library allocates additional memory pages and links it to the previously allocated memory. Using this library, developers only need to pass one parameter, namely, the address of the memory allocated first for storing the parameters to the SCL. Thus, when the SCL is invoked, PrivateZone copies the memory pointed to by R0 (the address of the memory for the stored parameters), without recursively copying every parameter. In the SCL, the encoded parameters can be decoded using the PrivateZone library. Currently, the library only supports 1st-level referencing for the pointer type, but there are no fundamental barriers to handling multi-level pointers.

4.2.5 Cache Maintenance

For private execution, cache maintenance must be performed by PrivateZone. It should be noted that the main PrivateZone operations, such as copying the SCL from the REE, are performed in Monitor mode in the TEE. To access the REE (or PrEE) memory from the TEE, PrivateZone needs to map the memory in its page table in the TEE. Therefore, to copy the parameters from the REE and place them in the PrEE, PrivateZone maps both memory regions in the TEE. After PrivateZone writes the parameters into the PrEE mapped memory region, the cache of the written memory region is invalidated. This is because memory writing from the TEE to the PrEE (or the REE) only updates the PrEE (or REE) physical memory, while the cache of the written memory region reflects the old memory, leading to a cache coherency problem. In addition to parameter copying, when the SCL's entry point is patched with an SVC instruction, the data (and instruction) caches and REE branch predictor must be invalidated. In ARM, cache invalidation differs from the cache clearing for the opposite case; the cache is updated but the memory reflects the old content.

Because PrivateZone allocates new memory pages in the PrEE to place the objects (such as the new exception vector, copied SCL and parameters), the virtual-to-physical mappings in the PrEE page-table should be updated. Consequently, the TLB should also be invalidated. Therefore, it is possible that this TLB invalidation would degrade the overall system performance.

We have two possible optimization solutions for minimizing the performance degradation incurred by the TLB invalidation. First, we can utilize the non-global (nG) flag in the page-table entries and Address Space Identifier (ASID) defined by the CONTEXTIDR register. If another ASID is assigned, the page table entries mapping the non-global pages can be updated without invalidating the TLB. This feature is normally used by the OS to manage context switching between processes without invalidating the TLB. We can utilize the Virtual Machine Identifier (VMID) that is part of the virtualization extensions. The VMID is leveraged to switch between VMs without invalidating the TLB. By using both features, we can remove TLB invalidations in a PrivateZone implementation, which reduces the impact on the REE OS. We assign different VMIDs for each environment to separate the cache between the REE and the PrEE. The ASID and nG flag are utilized to isolate the cache of each SCL in the PrEE.

4.2.6 PrEE Internals

In the ARM architecture, there are six general exceptions other than those related to HYP and Monitor mode. General exceptions such as interrupt, data abort, and supervisor call (SVC) are handled by the OS kernel. Although the PrEE aims to run the SCL in user mode, exception handlers are still required in kernel mode.

The handlers in the PrEE are made as simple as possible to minimize the attack surfaces exposed to malicious SCLs. The SVC exception handler manages system-service requests from the SCLs. The supported services are limited to memory management, random-number generation, and cryptographic operations. Depending on the service type, a service can be supported directly from the PrEE or from the PrivateZone in the TEE. For example, the PrEE handler can support memory management services for handling requests for heap allocation and deallocation. On the other hand, cryptographic operations such as message encryption using a device-specific key is performed in the TEE. To invoke TEE-based services, the handler yields control to PrivateZone by invoking an SMC instruction with the desired service number as a parameter. Before providing TEE services, PrivateZone always validates whether the requests originate from the deployed SCLs in the PrEE by checking the current value of VTTBR. This prevents a malicious REE OS from invoking TEE services without loading SCLs into the PrEE.

Other exception handlers only invoke SMC instructions. For instance, the SCL's return to the REE causes a page-fault that is trapped by the Prefetch Abort exception handler. Once the handler traps the abort, it transfers control to PrivateZone in Monitor mode. PrivateZone compares the abort address with the expected valid return-address (e.g., the instruction after SCL invocation). The valid return address is saved by PrivateZone when the SCL is invoked. If the abort address does not match the valid return address, PrivateZone regards it as an error. The interrupts are also handled outside the PrEE, such that the interrupt handler merely invokes SMC to post notification of the occurrence of the interrupt to PrivateZone. The occurrence of data-aborts is also notified to PrivateZone to handle the legitimate aborts arising within the unmapped stack area. Remaining exceptions such as FIQ and Undefined Instruction are treated as errors.

In addition to exception handlers, data objects are maintained in PrEE kernel mode to support private execution. The kernel stack temporally saves the SCL context during handling exceptions. For instance, PrivateZone references it to update the return values and parameters to be passed to the REE after SCL execution. The data structures that contain the heap-allocation information and user-mode context for each SCL are also maintained as kernel objects in the PrEE. Last, the page tables for each SCL and exception vector table that points to the address of each exception handler are also managed as kernel objects.

To guarantee private execution, the kernel components should be protected from malicious SCLs in the PrEE. At boot time, the component integrity is verified, then they are loaded into the PrEE. While the system is running, privileged instructions that can run in kernel mode are strictly limited to SMC. As stated in [22], [23], [24], privileged instructions configuring system control registers can be exploited by attackers. For instance, attackers can disable the MMU or redirect the address of an exception vector table to a malicious one by manipulating the control registers. Because the kernel in the PrEE is extremely limited, provid-

TABLE 2
Summary of PrEE Internals

Mode	Component		Description	Page attribute
User	SCL	Heap, Stack, Code & Data	Primitives to run the SCL	RW, RO, PXN
Kernel	Exception handlers	SVC	Invokes the PrEE and the TEE services	PRO, PX
		Prefetch Abort	Switches environment for returning to the REE	
		IRQ	Switches environment to the REE for handling the PrEE IRQs	
		Data Abort	Switches environment for handling legitimate data-aborts	
		Undefined Inst., FIQ	Regarded as errors	
	Data objects	Page table	- Initialized during the SCL deployment - Updated by PrivateZone for heap memory management	PRO, XN
		Kernel stack	Saves user-mode context when handling exceptions	PRW, XN
		SCL structure	Saves SCL context and heap allocation information	

ing simple system services and switching the environment to the TEE, an approach restricting the usage of privileged instructions is feasible. Also, as all the pages allocated for the SCLs are set with the Privileged Execute Never (PXN) flag, attackers cannot execute privileged instructions even if they succeed in triggering return-to-user attacks. Additionally, before SCL execution, PrivateZone sets the Writable Execute Never (WXN) flag by configuring the control register to prevent the execution of instructions fetched from the writable memory area. Finally, the exception handlers in the PrEE are small enough to allow their security to be verified manually. Table 2 summarizes the key features of the PrEE.

4.2.7 Scheduling the SCL

While the SCL is running, PrivateZone enables IRQ as an interrupt source in the PrEE. Since the IRQ is also used and handled in the REE, PrivateZone forwards the IRQ to the REE, where the REE OS handles it. Therefore, although the SCL runs on the PrEE, it is scheduled by the REE OS.

The IRQ handling procedure is shown in Figure 3. Once IRQ occurs in the PrEE, it is trapped by the IRQ handler. The handler first saves the SCL context, and invokes SMC with a parameter indicating IRQ. PrivateZone in the TEE clears the contents of all the registers (i.e., R0 - R12, SP and LR) and sets the value of LR to the address of a trampoline code in the REE to be executed after IRQ handling in the REE. SP is set to the address of the REE's kernel stack. Finally, PrivateZone switches the environment to the REE with the PC set to the address of the IRQ handler in the REE to start IRQ handling. The return process is done in reverse order. Once IRQ handling ends, the preset trampoline code is executed to switch the environment to the TEE. In the TEE, PrivateZone restores the interrupted SCL context and switches the environment to the PrEE to resume the SCL.

IRQ is enabled while the SCL is running to prevent CPU starvation in the REE. However, it incurs several security concerns. Since the REE OS takes over the scheduling task, attackers can perform a DoS attack against the SCL, which is not our attack model. Also, attackers can try to extract secrets from the SCL context. This is prevented because PrivateZone always clears all the registers before transferring control to the REE.

Depending on the SoC design, TrustZone can set Fast Interrupt Requests (FIQ) as an independent interrupt source. By configuring SCR, FIQ can be set as being non-maskable from the REE, but handled in the TEE. TrustZone can thus have its own scheduler for dispatching TEE services without interference from attackers in the REE. Although the current

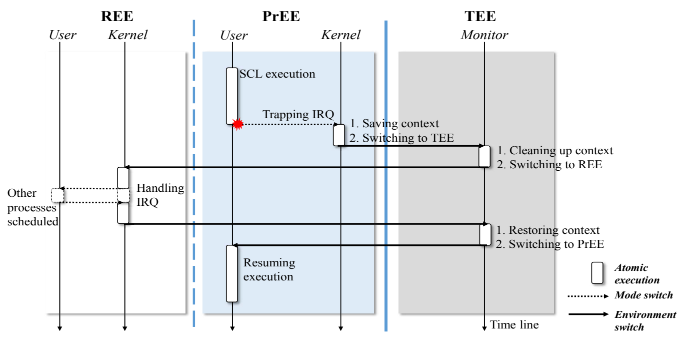


Fig. 3. Interrupts occurring in the PrEE are handled by REE OS. PrivateZone interposes switches between the REE and the PrEE to save and restore the SCL context.

TABLE 4
Lines of code in ASM (A) and C for implementation

Bootloader	REE	TEE	PrEE	Library
23 (A) + 7	34 (A) + 119	579 (A) + 4627	340 (A)	309

design of PrivateZone that shares IRQ with the REE may seem to be sufficient for private execution, we can explore further enhancement for securely scheduling SCLs.

4.2.8 PrivateZone Library

PrivateZone provides a library containing several operations that can be utilized to build PrivateZone-aware applications. Currently, only a few operations such as data encryption and memory management are supported. However, these operations are sufficient to illustrate the concept of PrivateZone. Table 3 lists the operations provided by the PrivateZone library. Especially, the domain column indicates the environment in which each operation can be invoked. For instance, genRandom() is only available in the PrEE. For operations that interface TEE services such as pzSeal(), PrivateZone always checks whether the request has been triggered from an appropriate environment before providing the services.

Additionally, we created a simple code-signing tool by using an open-source binary-instrumentation library [25] so that developers can sign the hash of the SCL before the deployment of their applications.

5 Implementation

We implemented our PrivateZone prototype on an Arndale board [26] with an Exynos5250 SoC and Cortex-A15 dual-core processor running at 1.7 GHz. For the REE OS, we

TABLE 3
Operations provided by PrivateZone Library

Operations	Description	Domain
deploySCL()	Sends a request to copy the SCL from the REE and then places it in the PrEE.	REE
revokeSCL()	Sends a request to remove the SCL deployed in the PrEE	REE
encodeParams(), decodeParams()	Serializes (or deserializes) parameters to (or from) the structured form of memory	REE/ PrEE
genRandom()	Generates a random value by using system events such as data abort as seeds	PrEE
pzSeal(), pzUnseal()	Encrypts (or decrypts) the input concatenated with SCL hash by using a device's symmetric key	PrEE
pzAttest()	Encrypts the input concatenated with SCL hash by using a device's private key	PrEE
pzMalloc(), pzFree()	Allocates (or deallocates) the memory in the PrEE	PrEE

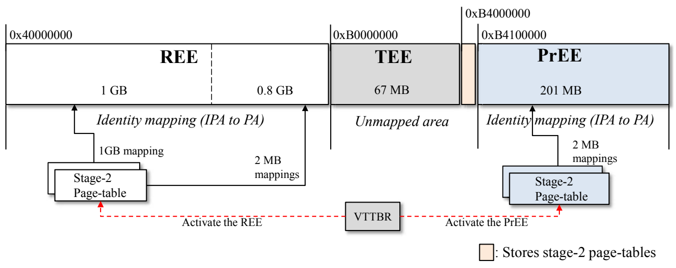


Fig. 4. Memory mapping of D-RAM for current prototype. The REE and the PrEE occupy 1.8 GB and 201 MB, respectively. They are separated using the stage-2 paging. Each environment has a dedicated stage-2 page-table which is pointed to by VTTBR. The TEE is isolated from both the REE and the TEE by using a hardware-based access-control mechanism such as TZASC. The size of each environment can be flexibly adjusted based on the device SoC design.

installed Linaro-Android (13.11). Open-source TrustZone software [14] was used as the TEE OS. The trampoline and PrivateZone kernel driver were newly deployed in the REE. The TEE hosts the main framework of PrivateZone in Monitor mode. In the PrEE, the exception handler code is loaded at boot time. We added a few lines of code to the bootloader [27] to initialize the physical memory of each environment. The PrivateZone library is also part of the PrivateZone implementation. An implementation summary is described in Table 4. The C implementation (4627 LoC) in the TEE ports crypto functions and SHA1 hashing to show the PoC of the TEE services.

5.1 Environment Creation

At boot time, each environment is created and initialized. Two phases are involved in the environment creation: (1) splitting the physical memory and (2) creating stage-2 page tables.

For splitting the physical memory, we adjust the number of D-RAM banks to be assigned for the REE in the bootloader. The Arndale board can have eight D-RAM banks giving 2 GB of memory. Therefore, each D-RAM bank occupies 268 MB. We assign seven D-RAM banks (1.8 GB) for the REE. The remaining one D-RAM bank is assigned for the TEE (67 MB) and PrEE (201 MB). Using this configuration, the REE OS recognizes that only 1.8 GB of memory is available in the system, such that memory-management in the REE is performed in the configured memory area.

Although the physical memory is split at boot time, the compromised OS in the REE can still access the physical memory assigned for the PrEE. Thus, we create two stage-2 page tables for each environment to prevent abnormal accesses. ARM processors with virtualization extensions support two page-table formats: short-descriptor (32-bit entries and up to two levels of lookup) and long-descriptor (64-bit entries and up to three levels of lookup) [28].

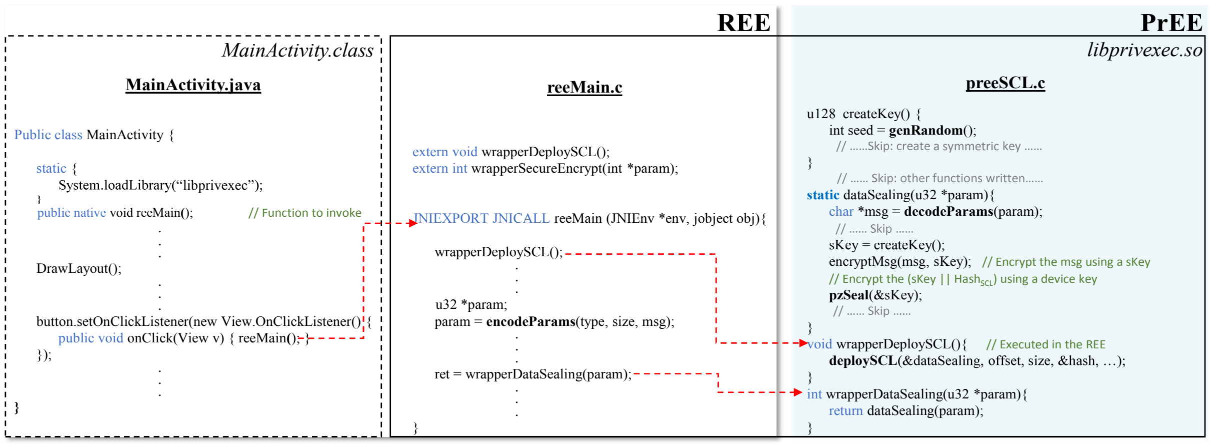
In our PrivateZone prototype, we use the short-descriptor format for paging in the REE, PrEE, and TEE, and the long-descriptor format for stage-2 paging. The stage-2 memory map of D-RAM is described in Figure 4. All the mappings are identical in terms of the Intermediate Physical Addresses (IPAs) to physical addresses. Also, the remaining physical memory area for accessing the peripherals (0x0-0x3FFFFFFF and 0xC0000000-0xFFFFFFFF) is mapped to the REE by using two 1-GB blocks. Once the mappings have been created, VTTBR is initialized to the address of the stage-2 page table of the REE. To enable stage-2 paging, we set the VM flag in the HCR. The virtualization-related registers and stacks are banked for each core, so we also configured the required registers of the second core at boot time.

5.2 Use Case - Android NDK Application

By using the native development kit (NDK) and PrivateZone, we created a simple Android NDK application that performs a conceptual data sealing. The application consists of two parts: Java code and a native shared library. The Java code defines the application layout (textboxes, buttons). It also loads the shared library and invokes functions in the library. The shared library is written in C, and defines the SCL that runs on the PrEE. We used a customized linker-script to separate the SCL from the other application logic. The REE part of the library was located in the default sections such as .text and .data. However, we isolated the SCL in a ".privatezone" section.

The operation of the sample application is as follows. That part of a library (reeMain.c) that runs on the REE deploys and invokes the SCL (preeSCL.c). When the SCL is invoked, plain text is passed as an SCL parameter. The SCL in the PrEE creates a symmetric key by using a random number generated by PrivateZone. The plain text is encrypted by the symmetric key. The key and the hash of the SCL are encrypted (pzSeal) with a device-specific key. The hash can be referenced to prevent the malicious SCLs from unsealing the data. Finally, the concatenation of the encrypted text, key and hash is returned to the REE as a result of executing the SCL.

Figure 5 describes the application structure. Note that the SCL deployment is performed by using wrapper functions instead of calling deploySCL(&dataSealing, ...) in the REE. Because the library is dynamically loaded during the application runtime, directly passing the entry point for the SCL as a parameter causes unexpected behavior.



MainActivity.java

```
Public class MainActivity {

    static {
        System.loadLibrary("libprivexec");
    }
    public native void reeMain();          // Function to invoke
                    .
                    .
                    .
    DrawLayout();
                    .
                    .
                    .
    button.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) { reeMain(); }
    });
                    .
                    .
                    .
}
```

reeMain.c

```
extern void wrapperDeploySCL();
extern int wrapperSecureEncrypt(int *param);

JNIEXPORT JNICALL reeMain (JNIEnv *env, jobject obj){

    wrapperDeploySCL();
                .
                .
                .
    u32 *param;
    param = encodeParams(type, size, msg);
                .
                .
                .
    ret = wrapperDataSealing(param);
                .
                .
                .
}
```

preeSCL.c

```
u128  createKey() {
    int seed = genRandom();
      // ......Skip: create a symmetric key ......
}
      // ...... Skip: other functions written......
static dataSealing(u32 *param){
    char *msg = decodeParams(param);
      // ...... Skip ......
    sKey = createKey();
    encryptMsg(msg, sKey);   // Encrypt the msg using a sKey
    // Encrypt the (sKey || HashSCL) using a device key
    pzSeal(&sKey);
     // ...... Skip ......
}
void wrapperDeploySCL(){     // Executed in the REE
    deploySCL(&dataSealing, offset, size, &hash, …);
}
int wrapperDataSealing(u32 *param){
    return dataSealing(param);
}
```

Fig. 5. Sample code for an Android NDK application that leverages PrivateZone. Here, libprivexec.so is a shared library that contains the SCL to be executed in the PrEE. MainActivity simply loads the shared library and invokes the services defined in the library. In particular, the functions in bold are provided by the PrivateZone library.

Specifically, &dataSealing in the REE returns the runtime linker address instead of the loaded SCL address. Thus, we defined dataSealing() as a *static* function in the .privatezone section and created additional wrapper functions – wrapperDeploySCL() and wrapperDataSealing() – for use in the REE.

In the example, the SCL can be remotely attested by sending a verifier the SCL's hash (with a nonce) that is signed by the device's private key. Also, the input message for the encryption can be protected by the secure I/O (i.e., trusted display and keypad). We discuss future work for enhancing PrivateZone in Section 7.

6 Evaluation

6.1 Security Analysis

In addition to the TEE and the REE, adopting PrivateZone adds one more environment – PrEE – to the system. We do not assume the existence of malicious manufacturers who hide malware in the TEE, so we trust the TEE services. However, the REE and part of the PrEE can be exploited by attackers. In this section, we analyze attacks that can be originated from each environment and discuss the mechanisms whereby PrivateZone defends against attacks.

6.1.1 *Attacks Triggered from the REE*

Since several components of PrivateZone are located in the REE, they can easily be exploited by attackers. As shown in SeCReT [29] and attacks "in the wild" [30], [31], attackers with kernel privileges are free to invoke SMC instructions with manipulated parameters. Thus, they can probe the vulnerability of PrivateZone. However, because PrivateZone in the TEE consists only of a few lines of code, its security can be manually validated before framework deployment.

Malicious parameters fed into the SCLs also threaten the PrEE and REE security. Even without kernel privileges, attackers can attempt to manipulate the REE kernel's critical objects by passing the object addresses to PrivateZone as SCL parameters. With the kernel privilege, they can also maliciously modify the page-table entry for the parameters in the REE so that it points to the physical address within the PrEE. To prevent such attacks, PrivateZone always verifies the validity of the address before performing memory operations. Attackers can also try to modify the SCL and its pre-calculated hash before the SCL's deployment. To prevent this kind of attack, the pre-calculated hash can be signed and encrypted by using the developer's private key and device's public key, respectively.

Finally, instead of loading the SCL into the PrEE, attackers can run the SCL in the REE to perform a man-in-the-middle attack. Thus, developers must always design a proper attestation mechanism and adopt it in their application. As discussed in 7.2, designing the mechanism also requires PrivateZone-provided functionalities such as signing the message with a device-specific (hardware embedded) key.

TABLE 5
LMBench latency measurement results (in μs)

	Linux	w/ PrivateZone	Overhead
null syscall	0.235	0.312	32.87%
open / close	5.362	5.380	0.34%
signal handler install	0.655	0.743	13.44%
signal handler overhead	2.796	2.797	0.04%
fork	169.230	169.438	0.12%
execve	184.659	184.748	0.05%
page fault	2.063	2.199	6.59%

6.1.2 *Attacks Triggered from the PrEE*

Attackers can also load a malicious SCL into the PrEE. However, as shown in Section 4.2.6, the privilege for malicious code is strictly limited to user-level privileges. Even in kernel mode, no privileged instruction other than SMC is allowed to run. Also, each SCL and kernel object are separated based on the page-table configuration. Only PrivateZone in Monitor mode can update tables. Especially, as the critical kernel objects and SCLs are never located on the same physical page, a ret2dir attack [32] that bypasses PXN protection is not possible in the PrEE. Also, the system services that SCLs can leverage are restricted to those provided by PrivateZone, preventing Iago attacks [33] that exploit malicious OS services to compromise the SCLs. Moreover,

TABLE 6
Application benchmarks results

	Linux	w/ PrivateZone	Overhead
decompress-bzip2	64.42 s	66.58 s	3.35%
build-php	448.15 s	462.71 s	3.25%
compress-7zip	1293 MIPS	1280 MIPS	1.01%
openssl	11.91 signs/s	11.59 signs/s	2.69%
ebizzy	10201 records/s	9936 records/s	2.60%
sudokut	143.51 s	145.95 s	1.70%
minion	344.04 s	351.79 s	2.25%
unpack-Linux	119.2 s	121.36 s	1.81%
SQLite	12.44 s	12.83 s	3.14%

TABLE 7
Breakdown of SCL execution time (in μs)

Components			Linux	w/ PrivateZone
Non-SCL	Execution		1258.1	1247.2
SCL	Deployment	SCL init.	N/A	209.4
		Hash check	N/A	2440.6
	Invocation	Param delivery	N/A	274.3
		Execution	197.5	523.4
	Revocation		N/A	112.4
Overall			1455.6	4807.3 (3.30x)

the exception-handler code in the PrEE is small enough to be manually verified, which minimizes the Trusted Computing Base (TCB) of the PrEE.

Finally, like the attackers in the REE, the malicious SCLs in the PrEE can arbitrarily invoke the TEE services to perform a brute-force attack [29], [30], [31]. However, the insecure communication channel to the TEE is an open (and orthogonal) problem that needs to be further studied in academia. Thus, adopting PrivateZone does not introduce new TEE-attack surfaces in terms of exploiting the insecure communication channel.

6.2 Performance of REE

Although PrivateZone does not impose overhead by manipulating the stage-2 page-tables, the REE components of PrivateZone affect the performance of the OS in the REE. In this section, we show the results of the benchmarks that mainly measure the overhead imposed on the REE OS.

6.2.1 Microbenchmarks

As microbenchmarks, we ran LMBench [15] that measures individual OS operations. Table 5 shows the average latency after running LMBench ten times. The null syscall represents the worst microbenchmark case, resulting in a 33% overhead. Note that the SCL invocation is performed by executing an SVC instruction. Thus, whenever the system-call handler is executed, PrivateZone checks whether the current SVC exception is related to the SCL invocation. This check routine consists of less than 10 instructions, and is the main cause of the overhead. However, as shown in the table, the overhead is significantly reduced as the latency increases. As a result, system calls such as fork, open, and execve are seldom affected by PrivateZone.

6.2.2 Application Benchmarks

We measured the performance of applications with PrivateZone using the Phoronix test suite [16] that provides various benchmarks. For the disk-bound benchmarks, we ran unpack-Linux and SQLite. For the processor-bound benchmarks, we ran seven benchmarks including compressing files, building php, signing files, measuring web server workloads, and solving complex problems. We used the default configuration and ran each benchmark ten times. The benchmark test results are listed in Table 6. Compress-7zip, openssl, and ebizzy show the throughput when handling the corresponding test case, so a higher value indicates better performance. The other benchmarks shows the average time required to complete each case. The benchmarks exhibit around 3% overhead imposed by PrivateZone.

6.3 Performance of PrEE

6.3.1 SCL Performance Analysis

To evaluate the SCL performance, we created a simple application, the operation of which is similar to that of the sample code shown in Figure 5. The application invokes the SCL, passing a 1-KB string as an input. In the SCL, a symmetric key is created using a random number provided by the TEE service. The input is XORed using the key. Once the XOR operation is complete, the key is also encrypted by the TEE service that encrypts the input by using a device-specific key based on the AES-CBC algorithm. We built the application as an executable binary by running the ndk-build script.

We ran the application 100 times and analyzed the time required to execute the SCL. In addition to the private execution of the SCL, we also executed the application without PrivateZone to estimate the overhead incurred by environment switching. We replaced the invocation of the TEE services such as random number generation and AES encryption with equivalent operations in the REE.

Table 7 shows a breakdown of the execution. Because the non-SCL part operation is same in both cases, there is negligible difference in the elapsed time for the non-SCL execution. For the SCL execution, we analyzed the overhead for three stages: deployment, invocation, and revocation. As shown in the table, calculating SHA1 hash of the SCL in the deployment incurred the largest overhead. The overhead for the SCL initialization such as copying the SCL and creating the new page table was around 209 μs. In the invocation, the 1-KB parameter delivery took 274 μs. For the delivery, PrivateZone copies the memory, but also invalidates the copied area cache by executing DCIMVAC operations. Although the SCLs perform the same operations in both cases, execution with PrivateZone took longer than with Linux due to the environment-switching overhead. In our evaluation, the overhead incurred by switching between the REE and the TEE was approximately 192 μs, resulting from switching the modes and saving (and restoring) the context of each environment. On the other hand, the switch between the PrEE and the TEE took less, at 71 μs. This is because the procedure for switching between the PrEE and TEE is even simpler than the switch between the REE and the TEE. We also measured the overhead induced by increasing the parameter size and the hash check. Table 8 shows the results. Specifically, param delivery indicates the time that elapses for encoding parameters, copying the memory and invalidating data caches.

TABLE 8
Performance of parameter delivery and hash check (in μs)

Param delivery				Hash check			
4 KB	8 KB	16 KB	32 KB	4 KB	8 KB	16 KB	32 KB
677.2	1419.1	2555.8	4696.4	2440.6	4704.4	9383.9	18950.1

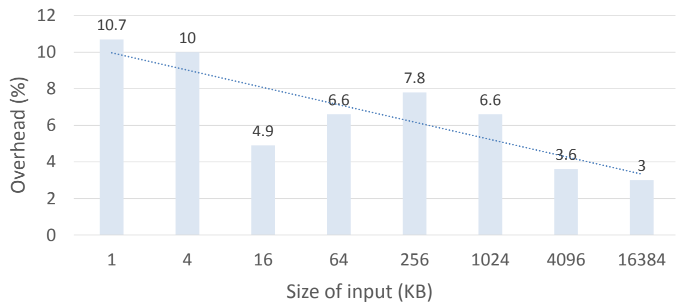


Fig. 6. Performance of AES Crypt, comparing between PrivateZone-ported and unmodified versions. Input file sizes vary from 1 KB to 16 MB. The results show the average overhead for 100 runs as imposed by PrivateZone, which indicates that the overhead is reduced as the input size increases.

6.3.2 *Porting PrivateZone and Macrobenchmarks*

As an application case study, we ported AES Crypt [34] to PrivateZone and measured the performance overhead imposed by the private execution. AES Crypt is open-source software that enables users to create a symmetric key and encrypt files using the key. It uses the AES and SHA256 algorithms to encrypt and decrypt the files. We deployed the encryption and hashing logic of AES Crypt as an SCL in the PrEE. The SCL is approximately 32 KB. Also, the SCL is repeatedly invoked to process the input based on a granularity of 2 KB until the encryption is completed.

Finally, we compared the ported version performance to that of the original version that runs on Linux. We ran each version 100 times with input files, the size of which varied from 1 KB to 16 MB. Figure 6 shows the average overhead for each input size. The overhead is a maximum of 10.7% and a minimum of 3% with inputs of 1 KB and 16 MB, respectively. The results reveal a tendency for there being less overhead as the input size increases. This is because, once the SCL is deployed, the SCL hash calculation that induces the greatest overhead is not performed again during the application runtime. Instead, the SCL invocation is repeated until the cryptographic operations are complete. Thus, we expect that the optimization that minimizes the number of SCL invocations will also reduce the overhead.

7 Discussion

7.1 Limitation of PrivateZone

PrivateZone isolates the SCLs from the REE OS. This approach prevents attacks that would provide malicious system services to the SCLs [33], [35]. However, this approach also limits the usability of PrivateZone. That is, the system services that developers can use to design the SCLs are limited to those provided by PrivateZone. Because the current PrivateZone prototype does not support multi-threaded applications, developers must write the SCL sequentially. Also, file operations are not supported in the PrEE, so SCL is developed to handle the file content as a parameter instead of directly opening the file in the PrEE. Additionally, data sealing and unsealing mechanisms associated with SCL versioning must be considered. To enhance the applicability of PrivateZone, required services should be verified and further explored. At the same time, the increase in TCB of PrivateZone as a result of providing more services should be minimized. We will address this issue in our future work.

7.2 Hardware Resources as PrivateZone Services

Developers can design a remote attestation for SCLs by using PrivateZone's cryptographic services. For instance, the hash of the SCL combined with a nonce value can be signed by using a device's private key and sent to a verifier (see Appendix A). This requires PrivateZone to access a hardware component such as secure storage containing the device's private (and public) key.

In addition to secure storage, PrivateZone can provision trusted hardware resources to the SCLs as TEE services. For example, we can consider using an LED light that is dedicated and isolated for TrustZone to indicate the correct SCL execution in the PrEE. Keypad and display can also be dynamically controlled as TEE resources by configuring TZPC [4], [36], [37]. PrivateZone can provide an interface to those hardware resources to allow valid SCLs to use the resources. Finally, the SCL's data needs to be properly sealed (and unsealed) using a key that is derived from the SCL's information (e.g., the SCL hash) and the device-specific key. For cryptographic operation optimization, we can adopt hardware-based cryptographic accelerators as well [38]. The exploration of more useful hardware resources and the creation of PrivateZone services that link them will also be addressed in our future work.

7.3 PrivateZone in HYP mode

Implementing PrivateZone in HYP mode could be advantageous over the current implementation in that it does not require any change in the OS or application in the REE. However, unlike the x86's virtualization techniques, the ARM architecture does not allow the direct invocation of hypervisor call (HVC) in user mode. Therefore, to directly trap a user-mode request such as SCL deployment in HYP mode, PrivateZone needs to set the Trap General Exceptions (TGE) flag in HCR to trap the general exceptions that occur in user mode. Such general exceptions include SVC exceptions that can be synchronously triggered by a user application. Thus, PrivateZone can use the SVCs instead of HVCs to create a direct communication channel between the application and PrivateZone. Intuitively, this approach might entail some performance degradation of the REE OS because every exception in user mode causes a switch to HYP mode regardless of the exception origination.

Alternatively, by patching that part of the REE OS, we can invoke HVC in kernel mode to selectively handle a request from user mode. This approach is analogous to the current design except for fact that the main framework of PrivateZone runs in HYP mode. *However, we expect that this approach will complicate the design of PrivateZone because some PrivateZone components need to remain in Monitor mode to support the TEE services.*

8 Related Work

8.1 Various Forms of TEE

The TEE offers an execution space that is isolated from the REE. Hence, any technique that provides an isolated environment can be leveraged as a TEE source. Intel SGX [11] enables applications to be executed while ensuring integrity and confidentiality of their security-critical code. Haven [39] and VC3 [40] protect applications from an untrusted cloud environment by using SGX. Remote attestation [41] and secure OTP [42] were also introduced as use-cases of SGX.

System Management Mode (SMM) is a special operating mode that is triggered by a System Management Interrupt (SMI). Because SMM operates while the REE is halted, previous works utilized SMM as the TEE to deploy and perform the security functionalities such as the integrity monitor [43], [44], [45], malware detection [46], and stealthy debugger [47].

TrustZone, which provides the TEE, was leveraged to read the sensors safely [48], run the critical part of the mobile applications [9], protect the UI [49] and OTP [36], and perform remote attestation [37]. In addition to protecting critical services, TrustZone also hosts kernel-integrity monitoring tools [22], [23], and a framework for the memory acquisition of the rich OS [50] and securing a communication channel [29].

Many previous works created the TEE based on the isolation and privilege-layer introduced by a hypervisor. With the trust in the hypervisor, previous works studied techniques for protecting the integrity of a guest OS [51], [52], [53] and bridging the semantic gap [54], [55], [56].

PrivateZone is differentiated from those works by introducing an isolated execution environment, PrEE, rather than crowding the TEE (by running more applications inside the TEE) that is dedicated to the secure hardware or hypervisor. We attempted to find the best combination of security and virtualization extensions in the ARM architecture, and will continue to explore other areas in which PrivateZone can be enhanced.

8.2 Secure Process Execution

Several systems use virtualization techniques to protect an application from an untrusted OS. Inktag [8], SP^3 [57] and Overshadow [7] are based on hypervisors, and leverage the cryptographic methodology to hide application code and data from the OS. TrustVisor [5] is a special-purpose hypervisor that protects a selected portion of an application. MiniBox [6] introduces two-way protection between the OS and the application by combining Google Native Client (NaCl) and TrustVisor. Process out-grafting [58] and Proxos [59] split the execution in some level (e.g., privileges or system calls) to protect the application. These systems are similar to PrivateZone in that the virtualization techniques are utilized. However, instead of implementing a hypervisor, PrivateZone uses a small part of the virtualization technique – stage-2 paging – to statically create an isolated execution environment. Also, PrivateZone mainly explores and leverages features that are specific to the ARM architecture for securing mobile devices.

Besides the virtualization techniques, different techniques were also utilized to achieve the same goal. Flicker [60] leverages AMD's Secure Virtual Machine (SVM) extensions and TPM to securely run the piece of the application without requiring VMM. Virtual Ghost [61] uses a compiler-based approach to protect an application from an untrusted OS. XOM [62] is a specially designed secure processor that uses XOMOS [63] as its dedicated OS. In addition to virtualization, the techniques used in those works can also act as good references for enhancing PrivateZone.

8.3 Opening the TEE

The fact that third-party developers have only limited access to hardware-based TEEs is a well-known problem. Thus, several studies have attempted to address this issue. ObC [10] enables third-party developers to deploy trusted applications in the TEE. TrustICE [64] isolates the secure code from the TEE and the REE, but it does not support the REE protection and the scalability on multi-core systems. OpenSGX [65] provides essential components for SGX development based on QEMU. Those studies contributed to making the real hardware-based TEE or virtual TEE publicly accessible.

On the other hand, GlobalPlatform [66] provides standard specifications for the TEE. To enable fast and efficient TEE-application development, Open-TEE [67], [68] aims to provide a virtual TEE that complies with GlobalPlatform's standards. OP-TEE [69] and Sierraware [14] provide an open-source TrustZone OS that also supports the GlobalPlatform standards.

Previous studies have highlighted that the aim is to make the TEE publicly accessible. In this study, not only the openness of the TEE but also the security of the three environments (i.e., the TEE, REE and PrEE), and PrivateZone's coordination with the existing TEE services, were considered as part of our study.

9 Conclusion

PrivateZone provides the benefits of the isolated execution to general applications, whereas the access to TrustZone technologies is currently limited to a few pre-authorized ones approved by the TEE business alliances. Using PrivateZone, developers can create applications to run SCLs securely without crowding the TEE. In summary, not only the protection of the SCLs but also the security of the existing TEE services, and the provision of the services to the SCLs, were considered in our design of PrivateZone.

Appendix A
Remote Attestation Example

Figure 7 describes the simple remote attestation example that can be designed by application developers. $PrvK_D$ and $PubK_D$ denote the device's private key and public key, respectively. Notation Comp(X,Y) indicates the byte-comparison between X and Y. Encrypt() and Decrypt() indicate asymmetric crypto fuctions such as RSA. The malicious SCL might abuse a TEE service to forge the hash, provided that the service allows the autonomous use of the private key without calculating and concatenating the hash. Thus, the use of the private key is only available through the invocation of pzAttest() that entails the hash calculation of the current SCL.

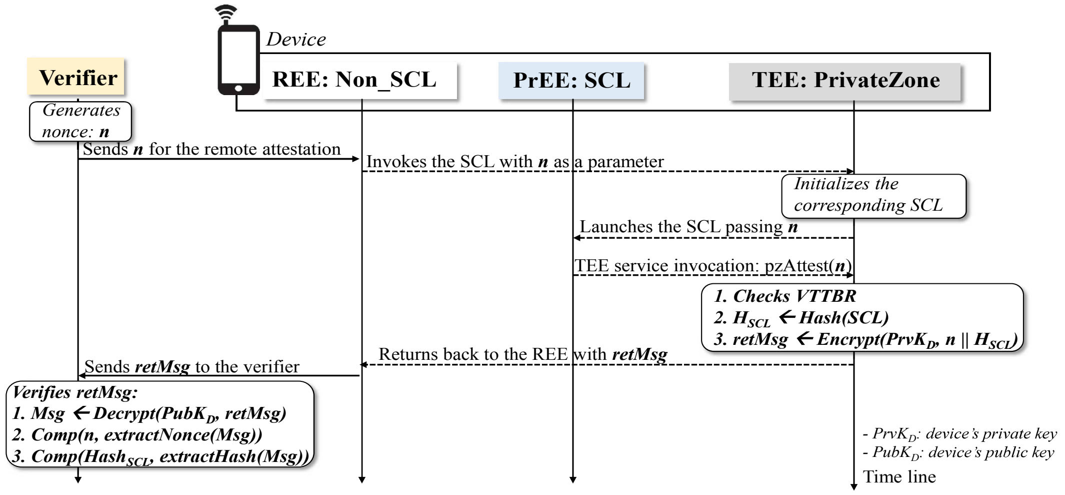

Device
Verifier
REE: Non_SCL
PrEE: SCL
TEE: PrivateZone
Generates nonce: n
Sends n for the remote attestation
Invokes the SCL with n as a parameter
Initializes the corresponding SCL
Launches the SCL passing n
TEE service invocation: pzAttest(n)
1. Checks VTTBR
2. H_{SCL} ← Hash(SCL)
3. retMsg ← Encrypt($PrvK_D$, n || H_{SCL})
Returns back to the REE with retMsg
Sends retMsg to the verifier
Verifies retMsg:
1. Msg ← Decrypt($PubK_D$, retMsg)
2. Comp(n, extractNonce(Msg))
3. Comp($Hash_{SCL}$, extractHash(Msg))
- $PrvK_D$: device's private key
- $PubK_D$: device's public key
Time line

Fig. 7. Remote attestation example using PrivateZone. (1) The verifier generates a random nonce value and remotely sends it to the application running on the device. (2) The non-SCL part of the application receives the request for the remote attestation. (3) The SCL is initialized and invoked with the nonce value as a parameter. (4) The SCL invokes the TEE service –pzAttest– to create a return message to be sent to the verifier. (5) PrivateZone verifies if the pzAttest is invoked in the PrEE by checking the VTTBR. Also, it calculates the hash of the SCL, and creates a return message by encrypting the concatenation of the hash and the nonce with a device's private key. (6) The verifier decrypts the return message with a device's public key, and verifies the nonce and the hash extracted from the return message.

REFERENCES

[1] (2015, Oct.) Credential storage enhancements in android 4.3. http://nelenkov.blogspot.ch/2013/08/credential-storage-enhancements-android-43.html.

[2] (2015, Oct.) Proxama. http://www.proxama.com/products-and-services/trustzone.

[3] (2015, Oct.) Discretix. http://www.discretix.com/products-solutions.

[4] (2015, Oct.) Arm security technology - building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[5] 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.

[6] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *2014 USENIX Annual Technical Conference*, 2014.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, pp. 1–1.

[12] S. Johnson, U. Savagaonkar, V. Scarlata, F. McKeen, and C. Rozas, "Technique for supporting multiple secure enclaves," 2012, uS Patent App. 12/972,406. [Online]. Available: http://www.google.com/patents/US20120159184

[13] F. X. McKeen, C. V. Rozas, U. R. Savagaonkar, S. P. Johnson, V. Scarlata, M. A. Goldsmith, E. Brickell, J. T. Li, H. C. Herbert, P. Dewan *et al.*, "Method and apparatus to provide secure application execution," 2015, uS Patent 9,087,200.

[14] (2015, Oct.) Sierraware. http://www.openvirtualization.org/.

[15] 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.

[16] (2015, Oct.) Phoronix test suite. http://www.phoronix-test-suite.com/?k=home.

[17] (2015, Oct.) Amd-v nested paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[18] 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.

[19] (2015, Oct.) Corelink system memory management unit. http://www.arm.com/products/system-ip/controllers/system-mmu.php.

[20] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "Nohype: virtualized cloud infrastructure without the virtualization," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 350–361.

[21] (2015, Oct.) Procedure call standard for the arm architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf.

[22] 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.

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

[24] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, V. Adve, S. K. Sahoo, C. Geigle, B. Ding, Y. He, Y. Wu *et al.*, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 191–206.

[25] (2015, Oct.) The eresi reverse engineering software interface. http://www.eresi-project.org/.

[26] (2015, Oct.) Arndale board. http://www.arndaleboard.org/wiki/index.php.

[27] (2015) u-boot-linaro-stable. https://git.linaro.org/?p=boot/u-boot-linaro-stable.git;a=summary.

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

[29] 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*.

[30] (2015, Oct.) Full trustzone exploit for msm8974. http://bits-please.blogspot.kr/2015/08/full-trustzone-exploit-for-msm8974.html.

[31] (2015, Oct.) Here be dragons: Vulnerabilities in trustzone. http://atredispartners.blogspot.kr/2014/08/here-be-dragons-vulnerabilities-in.html.

[32] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Conference on Security Symposium, SEC*, vol. 14, 2014, pp. 957–972.

[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] (2015, Oct.) Aes crypt. https://www.aescrypt.com/.

[35] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015.

[36] 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.

[37] 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.

[38] (2015, Oct.) M-shield mobile security technology. http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf.

[39] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[40] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, S&P*, 2015.

[41] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013, p. 10.

[42] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, p. 11.

[43] J. Wang, A. Stavrou, and A. Ghosh, "Hypercheck: A hardware-assisted integrity monitor," in *Recent Advances in Intrusion Detection*. Springer, 2010, pp. 158–177.

[44] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 38–49.

[45] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Xen 0wning trilogy," *Invisible Things Lab*, 2008.

[46] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "Spectre: A dependable introspection framework via system management mode," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–12.

[47] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 55–69.

[48] 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.

[49] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014, p. 8.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 297–312.

[55] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 586–600.

[56] A. Saberi, Y. Fu, and Z. Lin, "Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14), San Diego, CA*, 2014.

[57] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 71–80.

[58] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 363–374.

[59] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 279–292.

[60] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 315–328.

[61] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: protecting applications from hostile operating systems," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 81–96.

[62] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[63] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 178–192.

[64] H. Sun, K. Sun, Y. Wang, J. Jing, and H. Wang, "Trustice: Hardware-assisted isolated computing environments on mobile devices," in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*. IEEE, 2015, pp. 367–378.

[65] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "OpenSGX: An Open Platform for SGX Research," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2016.

[66] (2015, Oct.) Device specifications. https://www.globalplatform.org/specificationsdevice.asp.

[67] T. Nyman, B. McGillion, and N. Asokan, "On making emerging trusted execution environments accessible to developers," in *Trust and Trustworthy Computing*. Springer, 2015, pp. 58–67.

[68] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-tee-an open virtual trusted execution environment," *arXiv preprint arXiv:1506.07367*, 2015.

[69] (2015, Oct.) Linaro: Op-tee. https://wiki.linaro.org/WorkingGroups/Security/OP-TEE.

Jinsoo Jang received the B.S. degree in Information and Computer Engineering from Ajou University, South Korea, in 2007. He also received the M.S. degree in Information Security from Korea Advanced of Science and Technology (KAIST) in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, KAIST. His research interest includes system security, especially in the trusted execution environment (TEE).

Changho Choi received the B.S. degree in Computer Science and Electrical Engineering from Handong Global University in 2012. He also received the M.S. in the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2014. He is currently working toward the Ph.D. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interest includes system security, especially in Intel SGX.

Jaehyuk Lee received the B.S. degree in Computer Science and Electrical Engineering from Handong Global University in 2013. He also received the M.S. in the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2015. He is currently pursing his Ph.D. at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interest includes system-level security, hypervisor security, and trusted execution environment.

Nohyun Kwak received the B.S. degree in Computer Science and Engineering from Pohang University of Science and Technology (POSTECH), South Korea, in 2002. He also received the M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2005. He is currently working toward the Ph.D. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). His research interest includes system security, especially in Intel SGX.

Seongman Lee received the B.S. degree in Computer Science from Chungnam university in 2015. He is currently working toward the master degree at the Graduate School of Information Security, Korea Advanced Institute of Science and Technology (KAIST). His research interest includes code reuse attack (CRA).

Yeseul Choi received the B.S. degree in Computer Science Engineering from Handong Global University in 2015. She is currently working toward the M.S. degree at the Division of Computer Science, Korea Advanced Institute of Science and Technology (KAIST). Her main research interest includes computer system security.

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.