PrOS: Light-weight Privatized Secure OSes in ARM TrustZone

Donghyun Kwon, Jiwon Seo, Yeongpil Cho, Byoungyoung Lee, Yunheung Paek, *Member, IEEE*

Abstract—TrustZone is a hardware security technique in ARM mobile devices. Using TrustZone, software components running within the secure world can be completely isolated from the normal world, which ensures hardware-enforced security access control over the underlying computing resources. In order to support multiple trusted applications, TrustZone runs its own operating system, called the secure OS, within the secure world. Unfortunately, attackers have been exploiting privilege escalation vulnerabilities in a secure OS, as reported in most of major secure OSes from product vendors including Samsung, Huawei, and Qualcomm. More critically, as all trusted applications are running on the same secure OS instance, compromising the secure OS leads to compromising all trusted applications, rendering the secure OS as a single point of failure endangering the entire TrustZone's security.

This paper presents PrOS, our mechanism to privatize secure OSes through direct virtualization of TrustZone. PrOS allows each trusted application to run with its own secure OS such that the secure OS is no longer a single point of security failure. One particular challenge for PrOS lies in how efficiently to implement software-only virtualization for TrustZone for a practical deployment in real systems despite the condition that the current ARM architectures do not support hardware-assisted virtualization for TrustZone. As opposed to the common belief that software-only virtualization is inefficient and sluggish, we have found several common design features inherent in the secure OS to leverage for optimally tailoring the TrustZone virtualization scheme. We implemented PrOS on a 64-bit ARM development board. According to our evaluation, PrOS incurs 0.02% and 1.18% performance overheads on average in the normal and secure worlds, respectively, demonstrating its effectiveness in the field.

Index Terms—Security, TrustZone, Virtualization.

1 INTRODUCTION

The TrustZone technology is a hardware-level approach to security in ARM systems. TrustZone-based security solutions are built into an ARM system by chip manufacturers or product vendors who want to provide secure endpoints and a device root of trust. TrustZone enforces the security principle of privilege separation through partitioning all of the hardware and software components into two worlds: the normal world and the secure world. The normal world is generally used to execute rich operations (including a traditional operating system (OS) and its applications) which is prevented from accessing the secure world. On the contrary, the secure world is allowed to execute trusted operations (including a secure OS and its trusted applications (TAs)) running with a higher privilege than the normal world. Leveraging this higher privilege, the secure world can exclusively access various trusted IO devices, enabling sophisticated TAs such as secure key-rings, mobile payment, and digital right management (DRM).

In principle, TrustZone must be able to ensure secure executions of multiple TAs merely by running all of them in its own secure OS within the secure world. Unfortunately, TrustZone maintains only one secure OS in its secure world, and similar to numerous security issues plagued in the normal OS, the secure OS is simply a software product which cannot be vulnerability-free. In fact, several privilege escalation vulnerabilities have already been reported in major secure OSes running on commodity devices from various vendors including Samsung, Huawei and Qualcomm. This situation raises a critical concern on security guarantees of TrustZone. If any of TA is adversarial (or compromised), they can launch privilege escalation attacks using these vulnerabilities to take over the secure OS, consequently compromising all the TAs running within TrustZone.

A natural approach to address this security concern would be *privatizing* the secure OS, that is, assigning each TA its own individual secure OS, in a way to prevent the secure OS from being a single point of security failure. From the technical standpoint, the security *anchor* indispensable for this approach is a software layer with the highest privilege in the system, named as the *security access controller* (SAC), whose task is to facilitate the OS privatization by supporting multiple secure OSes that run respectively in their private execution environments isolated from each other.

To privatize a secure OS, previous studies have commonly implemented a SAC such that it can manage secure OSes running in the normal world, as illustrated in Figure 1. For example, in TrustICE [1], normal and secure OSes are all generated in the normal world. When the normal OS requests security services from a TA, the secure OS private to the TA is initiated to execute. The SAC lies in the secure world and intervenes with every switch between OSes. The central role of their SAC is to isolate the OSes from each other, thereby protecting unauthorized access to a secure OS from the outside. However, to fulfill this role, TrustICE mandates that only one OS should run at a time, exclusively occupying the entire normal world while suspending all the other OSes, without elaborate access control mechanism. Obviously, such a strategy to exclusively run one OS at a time might suffer from low resource utilization particularly in multi-core systems. As another example, in vTZ [2], the SAC is realized by combining a hypervisor and the secure monitor in TrustZone, as shown in Figure 1. It employs the hypervisor to construct virtualized execution environments in the normal world, each of which privately hosts a guest secure OS. Thus, unlike TrustICE, vTZ

- *D. Kwon, J. Seo, B. Lee and Y. Paek are with Seoul National University.*
- *Y. Cho is the corresponding author. (ypcho@ssu.ac.kr)*
  Y. Cho is with Soongsil University.

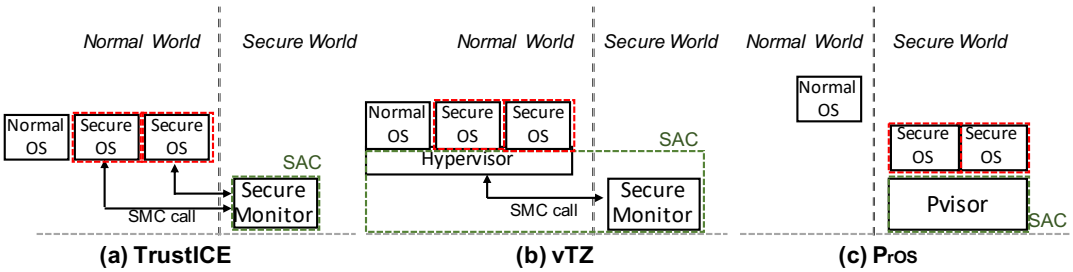

Fig. 1: The comparison of designs for secure OS privatization.

is able to execute the normal OS and secure OSes simultaneously by utilizing the multi-core systems. However, the virtualization in the normal world inevitably imposes a performance overhead on the normal applications as well as TAs. Meanwhile, the secure monitor in vTZ empowers every guest secure OS to have effectively a higher privilege than the hypervisor so that the secure OSes can protect themselves from any normal world entities including the hypervisor. Sadly, such a forceful reversal of privilege hierarchy [1] unavoidably entails a substantial amount of modification to the existing hypervisor code, thus possibly impeding a wide acceptance of vTZ in a real system already deployed in the field.

In this paper, we propose an alternative approach for the secure OS privatization, called *PrOS*, where our SAC residing in the secure world manages the multiple instances of the secure OS privatized for TAs directly in the same world, instead of remotely in the normal world. To accomplish this in PrOS, our SAC, which we call *Pvisor*, virtualizes TrustZone in the secure world, as shown in Figure1. It is noteworthy that since secure Oses and PrOS working in the secure world, it does not regulate the operation of normal world software and does not cause slow down to the normal OS execution. However, a naive implementation of TrustZone virtualization would suffer from considerable runtime overhead as well as high design complexity since we have to rely on software-only virtualization due to no hardware-assisted virtualization support in ARM TrustZone. Fortunately however, we have recently discovered that we can drastically simplify the TrustZone virtualization to enable light-weight runtime support for privatized secure OSes in PrOS by exploiting the three characteristics inherent in the secure OS design listed below.

- The secure OS adopts the *request-response* execution model where the request is issued from the normal world (i.e., a host application) and the response is then processed in the secure world (i.e., TA). This regularity of secure OS execution provides PrOS an opportunity to simplify a CPU virtualization mechanism, relieving us from complex scheduling issues.
- The secure OS uses relatively a small amount of physical memory than that of the normal OS, which in turn offers PrOS a better way to implement a memory virtualization mechanism more efficiently.
- The secure OS always relies on the normal OS to handle complex I/O operations, and the trusted I/O operations are never shared among different TAs. This eliminates a need for PrOS to cope with heavyweight I/O emulations when virtualizing complex IO services.

We have implemented a prototype of PrOS and evaluated it while running Android 7.1.2 (with Linux kernel 4.4.71) in the normal world and OP-TEE 2.5.0 as the secure OS to be virtualized by PrOS. According to our comprehensive benchmark results,

1. Note in an ordinary circumstance that a guest OS should have lower privilege than the underlying hypervisor.

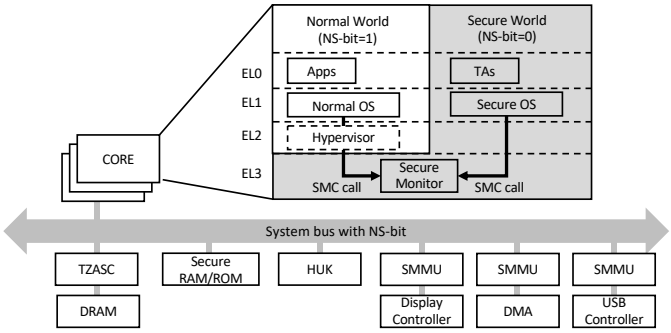

Fig. 2: High-level architecture of ARM TrustZone.

PrOS incurs 0.02% and 1.18% overheads in the normal and secure worlds, respectively. Also to demonstrate practical aspects of PrOS, we performed extensive use-case studies on TAs, namely bitcoin wallet (which manages a private key of a user and displays the bitcoin transaction information), safe vault (which stores private photos and videos only accessible through TrustZone), and DRM players (which decrypts and decodes a DRM content, and displays it to the secure screen). Our experimental results confirm that PrOS was able to not only run both of these TrustZone-based applications in a virtualized individual secure OS but also provide seamless user experiences with any noticeable delays.

The remainder of this paper is organized as follows. §2 provides necessarily background on TrustZone. §3 describes threat models of this paper. §4 presents the design of PrOS. §5 describes implementation details of PrOS. §6 shows use cases of PrOS and §7 evaluates PrOS. §8 discusses the related work and §9 concludes the paper.

2 Background

2.1 TrustZone

TrustZone is enforced by extending hardware components, overseeing access control across the entire architecture (shown in Figure 2). First, in the CPU-cores, the highest exception level (i.e., EL3), has been added to run a secure monitor that is responsible for the transition between the secure and normal worlds. Both the normal and secure OSes are running at EL1, both of which can enter the secure monitor by executing an SMC (Secure Monitor Call) instruction. The secure monitor then determines the subsequent security state of the world by setting the `NS-bit` of the Secure Configuration Register (SCR) to "1" for the normal world and "0" for the secure world. `NS-bit` is attached to transactions and spread across the system through the system buses, which allows hardware components of TrustZone to identify the security state of the transaction and enforce access control based on it. The system bus of TrustZone can determine the security state of devices

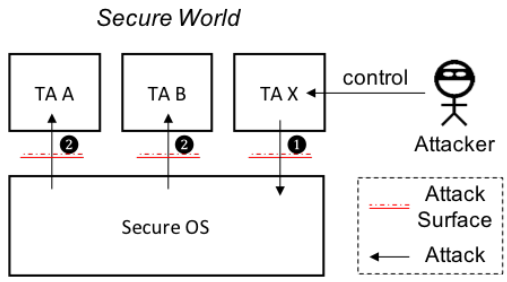

Fig. 3: Attack surface in the secure world where multiple TAs are running.

connected to it. Only the devices configured as secure are allowed to handle (or generate) secure transactions.

TrustZone supports more specific security configurations, especially for safeguarding memory subsystems and interrupt mechanisms. TrustZone Address Space Controller (TZASC) is a hardware component located in front of the DRAM controller. Leveraging TZASC, DRAM is logically partitioned into eight separate regions, each of which is physically contiguous and configured as either secure or non-secure.

After that, TZASC permits or blocks transactions toward DRAM according to the region configurations. Generic Interrupt Controller (GIC) extended by TrustZone allows interrupts to be set as secure. If interrupts configured as secure were raised, the CPU core automatically switches to the secure world and receives the interrupts if they are not masked.

2.2 Memory Management

ARM provides several system registers for controlling its memory system. System Control Register (SCTLR) is used to turn on or off the Memory Management Unit (MMU). To translate a virtual address to a physical one, MMU refers the page table pointed to by Translation Table Base Register (TTBR). TTBR also contains a special tag, called the *ASID field*, which enables efficient maintenance of consistency between TTBR and TLB when TTBR is updated. During an address translation, the ASID field is cached in TLB along with a virtual-to-physical mapping. After that, when MMU searches the address mappings cached in TLB, it excludes those having different ASID values than the current ASID, thereby eliminating the need for TLB invalidations for consistency between TTBR and TLB.

3 Threat Model

We assume that all software components running in the normal world (including the normal OS and all host applications) cannot be trusted—i.e., it is either compromised by attackers or adversarial. We assume that a secure OS is running in the secure world, which is a trusted component but it may have security vulnerabilities. Multiple TAs can run on top of the secure OS to leverage TrustZone services and each TA is owned by different third-party developers. Every TA developer has her/his own security and financial interests, so they do not trust each other. Consequently, a TA does not trust the TAs owned by other developers. Once TA X is compromised (or if the owner of TA X is malicious), TA X may attempt to launch an attack to compromise other TAs [3]. For example, such an attack can be carried out in the following steps described in Figure 3: (❶) TA X launches a privilege escalation attack (i.e., exploiting a vulnerability in the secure OS by invoking system calls with crafted parameters, which grants the secure OS's privilege); and (❷) TA X runs the malicious code with the secure OS's privilege to steal security sensitive information in other TAs.

The availability attacks against secure OSes are beyond the scope of PrOS. Moreover, we do not consider cache/timing side-channel attacks [4] and hardware attacks (e.g., cold-boot [5], bus snooping [6], and rowhammer [7]). Defense techniques against these attacks can be adopted for PrOS in the future, and thus it is orthogonal to this paper.

4 Design

In this section, we illustrate the design principles of PrOS (§4.1), and then describe the overview of PrOS, including its architecture and high-level workflow (§4.2). Next, we introduce PrOS's dynamic secure OS management mechanism that allows app developers to have their own secure OS on-demand (§4.3). Lastly, we describe our secure OS-aware virtualization mechanism that facilitates the establishment of the complete isolation boundary between secure OSes in an optimized way (§4.4).

4.1 Design Principles

The design of PrOS is constructed based on the following principles for security and performance efficiency.

- **P1. Isolated TrustZone Service:** The primary goal of PrOS is to provide the security benefit of TrustZone for various app developers who may have different security/monetary interests. Thus, PrOS has to provide privatized TrustZone services for each developer such that all developers can securely run their own TA being protected against potential threats from other developers' untrusted TAs.
- **P2. Minimum Changes in Existing Software:** The design of PrOS should be transparent to both the normal and secure worlds if possible. In other words, it should introduce a minimum change to all existing software components running in the normal world (e.g., hypervisors, normal OSes, and applications) as well as ones in the secure world (e.g., secure OSes and TAs), because it is always discouraged to entail additional changes in other existing software components for compatibility reasons.
- **P3. Minimize Trusted Computing Base:** As PrOS is governing TrustZone services, it requires additional high privileged operations controlling TrustZone's resources. Such additional operations may potentially open more attack surfaces to adversaries, so PrOS should try to minimize the TCB.
- **P4. Low Overhead:** Like any other system solutions, performance overheads to both the normal and secure worlds incurred by PrOS should be minimized. Otherwise, system developers will be reluctant to adopt PrOS, particularly in resource-constrained devices like smartphones.

4.2 Overview

PrOS provides app developers with a privatized secure OS, thus app developers are allowed to securely run their TAs even in the case that other developers' malicious TAs attempt to compromise the underlying secure OS by launching privilege escalation attacks. To realize this, PrOS supports dynamic management of secure OSes so that secure OSes can be loaded or unloaded at any time upon a request. PrOS also leverages virtualization techniques and lets each secure OS hold its own virtualized TrustZone resource, allowing secure OSes to be strongly isolated from each other and

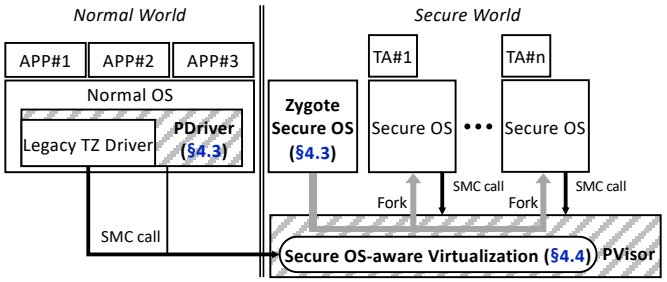

Fig. 4: The overview of PrOS.

to be simultaneously activated. In particular, PrOS applies the secure OS-aware virtualization mechanism, implementing TrustZone virtualization with minimal code size and low performance overhead.

Figure 4 illustrates the overall architecture. At the core lie the two software components: PDriver and PVisor. PDriver, placed within a normal OS (i.e., EL1), provides an interface for host applications to leverage PrOS. PVisor is implemented in the secure world (i.e., EL3) to virtualize TrustZone resources, such that multiple secure OSes isolated from each other can exist simultaneously.

The workflow of PrOS is as follows: (i) When the system is booting up, PVisor first initializes a secure OS. We call this firstly initialized secure OS as Zygote Secure OS, which will be cloned to dynamically load secure OSes of app developers; (ii) When a host application in the normal world requests a new PrOS service (i.e., launching TA in its own secure OS), PDriver forwards the request to PVisor through SMC calls. Then PVisor dynamically loads a secure OS by cloning the Zygote Secure OS; and (iii) All the following communication between the host application (in the normal world) and the TA (in the secure world) is properly routed by PDriver and PVisor.

4.3 Dynamic Management of Secure OS

PrOS allows each app developer to run her/his TA with its own secure OS. Toward this end, PrOS supports dynamic managements of secure OSes, so that each secure OS can be efficiently managed. In the following, we first describe how PrOS allocates memory space for the secure OS (§4.3.1), and loads and unloads the secure OS (§4.3.2 and §4.3.3, respectively). Then we present how PrOS supports interfaces to leverage these secure OS management services (§4.3.4).

4.3.1 Memory Allocation for Secure OS

PrOS loads a secure OS per app developer. This implies that PrOS would require large physical memory space to store multiple secure OSes at runtime, where each secure OS's memory space is isolated from the normal world and other secure OSes. In the following, we first describe the limitation of current hardware support (i.e., TZSAC), particularly in allocating physical memory space for secure OSes. Then we describe our approach, which can be divided into two steps: (1) weakly reserving the physical memory from the normal world; and (2) coordinating the normal world and the secure world to securely leverage TZSAC to perform actual physical memory allocation.

Challenge: Limited Hardware Support for Secure Memory Allocation. In order to load a secure OS, PrOS should allocate a physical memory region where the normal world is prevented from accessing. However, the underlying memory protection hardware (i.e., TZASC) only supports limited number of physically contiguous memory regions as explained in §2.1. More specifically, if physical memory regions are heavily fragmented (i.e., physical memory pages for each secure OS are interleaved) the TZSAC mechanism cannot enforce the memory protection for those memory regions. A naive approach to this problem would be reserving a large, physically-contiguous memory block that can be exclusively used for secure OSes, but this would severely decrease the physical memory utilization for normal OS.

Coordinated Physical Memory Allocation. To address this issue, PrOS develops a new physical memory allocation scheme coordinating the normal world and the secure world. This can be divided into following two steps: (1) physical memory reservation and allocation by the normal world; and (2) physical memory layout synchronization between the normal world and the secure world.

First, the normal world (i.e., PDriver) weakly reserves a large chunk of physical memory regions during the boot-time. Such reserved memory regions are allowed to hold movable objects that can be migrated to other memory location (e.g., user-space pages and page caches). When allocating the memory for a secure OS later, PDriver relocates the movable objects to return physically contiguous memory blocks. As a result, this memory management scheme enables PrOS to retain good memory utilization (as the reserved space can be used for other purposes) while minimizing the fragmentation of secure OSes (as it can return a contiguous memory region).

Second, once the normal world allocates the physical memory regions, the physical address (i.e., `PAddr`) along with its size is passed to the PVisor. Since the `PAddr` is determined by the PDriver and thus cannot be trusted, PrOS always maintains its own physical memory layout information so that it can verify the validity of an allocation (i.e., if there is any overlap with existing memory layout of the secure world or it is properly aligned to be secured by TZASC). If confirmed to be benign, PVisor starts loading a new secure OS as we describe next.

4.3.2 Loading Secure OS

After determining the physical memory address, PrOS starts loading the secure OS onto that address. PrOS employs the Zygote model [8] to efficiently handle this installation: (1) PrOS initializes the Zygote OS instance at the early boot stage; and (2) actual secure OS installation is simply cloning the execution contexts of the Zygote OS. The Zygote model works with the postulation in which we can locate the cryptography key and PRNG data in the Zygote OS instance through static analysis for source code and build processes.

Preparing Zygote OS. PrOS prepares the Zygote OS when booting up the system. More precisely, PVisor (1) initializes the Zygote OS by running its boot sequence, and (2) takes the snapshot of all physical memory contents, CPU-states, and device configurations of the Zygote OS, and (3) zeros out all cryptographic keys in the snapshot (so as to avoid key prediction attacks in the Zygote model [9]). This prepared Zygote OS will be used to quickly deploy secure OSes. It is worth noting that, since the memory contents of Zygote OS is the result of executing the code verified through trusted booting chain and is not the result of running any particular TA, its cloned version of secure OSes is trustworthy to app developers as well.

Cloning Zygote OS. PVisor clones the Zygote OS to load a new secure OS instance. It first hard-copies all the data and configurations of the Zygote OS to allocated physical memory regions (pointed by `PAddr`). Note that, since the code of the Zygote OS is not different between all secure OS instances, we does not copy the code, but make the instances share the single physical copy by using memory virtualization of PVisor (refer to §4.4.2). After cloning memory contents, PVisor further refreshes all the cryptographic keys and PRNG data in the instance, avoiding the cases that the cloned instances are using the same keys for cryptographic operations [9]. At this time, the update is implemented by copying a new value from entropy sources of PVisor. Moreover, in order to correctly authenticate a specific developer of TAs, the public key of the developer (i.e.,$\texttt{PuK}_{\texttt{dev}}$) is injected to this new OS instance such that it can correctly perform the dynamic TA provision [10], [11] (i.e., authenticate only if the TA is developed by the corresponding developer at runtime).

4.3.3 Unloading Secure OS

When unloading the secure OS, PVisor wipes all CPU-state and memory contents stored in the secure OS. In order to reuse this deleted memory pages, PVisor first configures TZASC to make the physical memory regions accessible from the normal world. Then PDriver frees the memory such that the deleted memory pages can be utilized later.

4.3.4 Provided Interfaces

Based on primitive operations regarding the secure OS management that we described before, PrOS implements interfaces that host applications (running in the normal world) can leverage PrOS's secure OS supports. We illustrate the provided programming interfaces (Figure 5) as well as its workflow (Figure 6). Particularly focusing on host applications' perspectives (which impacts the transparency of PrOS), PrOS adds two explicit interfaces for loading and unloading the secure OS on demand. We note that, other than this loading and unloading interfaces, PrOS is compatible with the legacy programming model for TA usage and thus completely transparent to host applications.

Interface for Loading. At first, host applications can initiate the loading of a secure OS by sending a request to PDriver through `PD_LoadSecureOS` with the $\texttt{PuK}_{\texttt{dev}}$ of the vendor. Upon receiving the request, PDriver uses its physical memory allocation mechanism to find available `PAddr` (as illustrated in §4.3.1). After that, it forwards the $\texttt{PuK}_{\texttt{dev}}$ and the `PAddr` to PVisor by invoking `PV_LoadSecureOS`. PVisor then loads a new secure OS cloning the Zygote OS as well as updating cryptographic keys (as illustrated in §4.3.2). Finally, PDriver increases the reference count of the secure OS and completes the loading task.

Interface for Unloading. Similar to the creation, PrOS provides an explicit interface for host applications to unload their secure OSes, `PD_UnloadSecureOS`. It takes $\texttt{PuK}_{\texttt{dev}}$ as a parameter, PDriver decreases the reference count of the corresponding secure OS (specified by $\texttt{PuK}_{\texttt{dev}}$). If the reference count becomes zero, PDriver invokes `PV_UnloadSecureOS` so that PVisor can truly unload and clear the secure OS (as illustrated in §4.3.3). It is worth noting that availability attacks are not part of PrOS's threat model as we have stated in §3 (i.e., any malicious host applications or normal OS can unload any secure OS).

Transparent Runtime Invocation Support. When host applications invoke commands to their TAs at runtime, it is possible

Interface Name	Parameter	Description
host applications → PDriver (ioctl)		
PD_LoadSecureOS	$\texttt{PuK}_{\texttt{dev}}$	Loads a secure OS and Increase its reference count.
PD_UnloadSecureOS	$\texttt{PuK}_{\texttt{dev}}$	Decreases the reference count. If it is 0, destroys the corresponding secure OS.
PDriver → PVisor (SMC call)		
PV_LoadSecureOS	$\texttt{PuK}_{\texttt{dev}}$, PAddr	Loads a secure OS.
PV_ActivateSecureOS	$\texttt{PuK}_{\texttt{dev}}$	Activates the secure OS in the current core.
PV_UnloadSecureOS	$\texttt{PuK}_{\texttt{dev}}$	Unloads the secure OS.

Fig. 5: The interfaces for host application and PDriver.

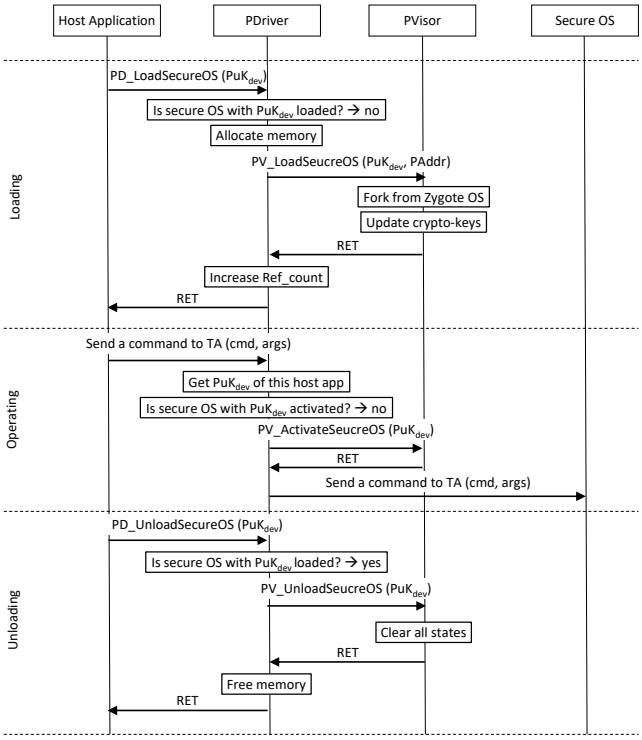

Fig. 6: The workflow of host application with PrOS.

that the corresponding secure OS is not currently activated (i.e., it was loaded before but the CPU core is currently occupied for some other secure OS to run a different developer's TA). In this case, PrOS automatically activates the corresponding secure OS. To be more specific, when a host application attempts to send messages to a secure OS through the legacy TrustZone driver, PDriver intervenes and ensures that the the target secure OS is currently activated. In other words, PDriver identifies $\texttt{PuK}_{\texttt{dev}}$ using the provided context from the host application, and checks whether the corresponding secure OS is currently activated. If not activated, PDriver invokes `PV_ActivateSecureOS`. PVisor activates the secure OS by setting the current core to use the CPU-state of the secure OS (see §4.4.1). After that, PVisor returns control to the PDriver, and in turn the TrustZone driver starts the communication with the activated secure OS. We note that this runtime invocation support as well as automatic secure OS activation mechanism is completely transparent to host applications, which requires no modification of host application code.

4.4 Secure OS-aware Virtualization

PrOS leverages virtualization techniques to support multiple privatized secure OS simultaneously. Note that as there is no hardware support for virtualization in TrustZone, PrOS needs to

use software-based virtualization techniques. However, in general software-based virtualization impose high performance overheads and complex implementations [12]. To resolve these problems, we tailor our virtualization methods leveraging the inherent design characteristics found in the practical secure OSes, such as OP-TEE [10], ObC [11], [13], and ANDIX OS [14]. In particular, we focus on following three main observations:

O1. Request-response model:. The secure OS adopts the request-response execution model, where the request is made from the normal world (i.e., a host application) and the response is performed by the secure world (i.e., TA). This is mainly due to the role of a host application and its TA: TAs are not designed to be a standalone program, but a host application drives the execution of TAs. As a result, the secure OS delegates the scheduling responsibility for TAs to the normal OS. Specifically, when a host application calls its TA, the secure OS and the TA run on a single "calling" core. This allows PrOS to take a lightweight CPU virtualization mechanism, intentionally omitting complex scheduling support.

O2. Small memory footprint:. The secure OS only holds a few memory pages, orders of magnitude smaller than a normal OS. Thus PrOS' memory virtualization can take a simple yet efficient approach, different from traditional memory virtualization techniques.

O3. Offloaded I/O operations:. The secure OS relies on the normal OS to provide complex I/O services such as file I/O and networking because the lightweight secure OS is more desirable due to its strong and high security privilege. This allows the secure OS to exclude the implementation code bases including disk driver, network driver, file system, and socket. Since such outsourcing should be performed securely, secure OS typically encrypts the data using cryptographic keys. This allows PrOS to avoid heavyweight I/O emulations to virtualize complex IO services.

In the following, we describe how PrOS performs software-based virtualization, namely CPU (§4.4.1), memory (§4.4.2), and device virtualization (§4.4.3), while each virtualization mechanism is driven by aforementioned observations.

4.4.1 CPU-State Virtualization

PrOS simplifies the CPU virtualization by leveraging the characteristic of the secure OS for request-response execution model (i.e., **O1**). Traditional virtualization techniques virtualize the CPU for the following two general operational roles: computational resources (i.e., scheduling) and CPU states. From the PrOS's perspective, however, it is not required to virtualize computational resources. Due to the request-response execution model, the CPU core to be scheduled is determined by the normal world and the secure OS will voluntarily yield the CPU core when the timer interrupt of the normal world occurs [2]. This in fact significantly simplifies the implementation of the CPU virtualization, which in turn minimizes the TCB of PrOS (e.g., the scheduler of Xen hypervisor (v4.10) [15] is about 4K SLOC).

PVisor virtualizes CPU-states per CPU core by saving and restoring CPU-states when exiting from or entering to secure OSes, respectively. PVisor employs two different save/restore schemes depending on how the context switching of the secure OS occurs. First, in the case of the world switch (i.e., from the normal world to the secure world or vice-versa), PVisor saves and restores all

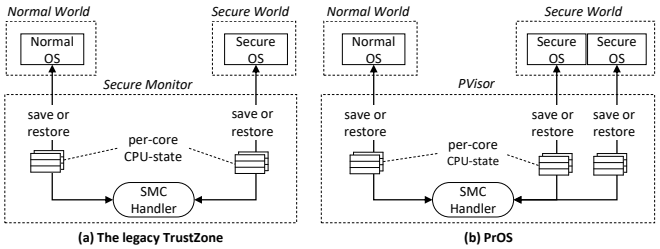

Fig. 7: Save and restore of CPU states in the legacy TrustZone and Playground

CPU-states only when world switchings occur. Second, in the case of temporal exits from secure OSes to PVisor (i.e., for memory virtualization), PVisor only saves a partial set of registers (i.e., general registers and the program counter, the stack pointer), and the process state. Since such temporal exits do not entail the world switching, PVisor can be certain that values in unsaved registers (a number of system registers such as SCTLR and TTBR) will not be changed, thereby optimizing the performance.

4.4.2 Memory Virtualization

PrOS requires to virtualize the memory of secure OSes for both security and compatibility reasons: In terms of security, PVisor should take complete control over their memory space such that secure OSes will be completely isolated and protected from each other. In terms of compatibility, PVisor should be able to offer every secure OS to the same virtual memory layout, as all secure OSes are cloned from the Zygote secure OS.

Trap-and-Emulate for MMU-related Instructions. There are various privileged instructions related to MMU, for which Secure OSes have to execute under the supervision of PVisor. This supervision is required for the following two reasons: (1) secure OSes will behave abnormally as it may not follow the underlying virtual memory semantics imposed by PVisor (e.g., TLB invalidates and virtual-to-physical translation); and (2) even worse, secure OSes may circumvent the memory virtualization of PVisor (e.g., MMU on/off, TTBR and TCR updates, and global write-execute-never) for disabling isolation between secure OSes.

Therefore, PVisor employs a trap-and-emulate techniques [16] to deprive secure OSes of directly executing MMU-related instructions. Specifically, when preparing Zygote OS instance (§4.3.2), PVisor scans through the entire code instructions to find privileged instructions. For each privileged instruction found, PVisor replaces it with a trap instruction (i.e., SMC calls) such that PVisor can securely emulate the functionality of the original privileged instruction. As a result, since all secure OSes are forked from Zygote OS, all privileged instructions are completely delegated by PVisor.

Page Table Virtualization. Similar to MMU-related instructions, PrOS should have full control over page table of secure OSes in order to prevent secure OSes from accessing to an isolated memory region assigned to each of them. This also enables PrOS to enforce write-execute-never policy, thwarting secure OSes from injecting privileged instructions arbitrarily [3].

To give PrOS the exclusive control authority over page table of secure OSes, we employs a page table virtualization technique.

2. A compromised secure OS may be able to carry out the availability attacks seizing the CPU core, but this security concern beyonds our threat model mentioned in §3.

3. It does not hinder the normal execution of secure OSes because in most cases secure OSes have already applied the write-execute-never policy to their code for the same reason.

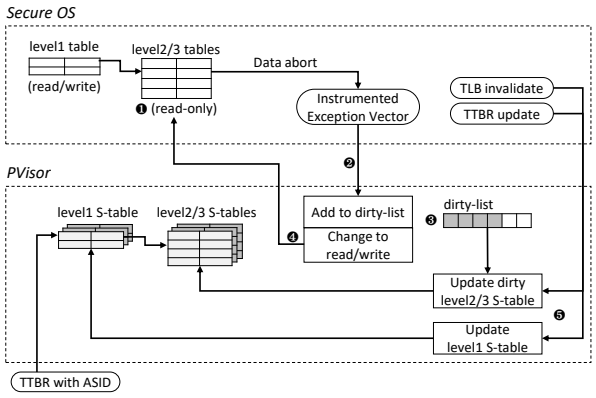

Fig. 8: The page table virtualization of PrOS based on the shadow paging.

Secure OSes are only allowed to access to virtualized page table so that even when being compromised they cannot manipulate physical page table directly managed by PrOS. To virtualize the page table, PrOS uses software-based shadow paging scheme [17]. Secure OSes manage their own page tables as usual, and PVisor creates shadow page tables (SPTs) that MMU will really refer by reflecting the secure OSes' page tables. In fact, a shadow paging scheme is not a common choice for modern memory virtualization schemes due to its inherent performance bottleneck in synchronizing the entire page table with SPTs at certain events (i.e., Translation Table Base Register (TTBR) update, TLB flush, and translation fault). However, we found that secure OS only holds a few page table entries, orders of magnitude smaller than a normal OS (i.e., **O2**), and thus such performance bottleneck becomes endurable in PrOS.

PrOS further optimizes the shadow paging scheme through minimizing the SPTs to be synchronized (illustrated in Figure 8). In other words, in the beginning PrOS marks level 2 and 3 of SPTs as read-only (❶). When secure OS attempts to update the page table, PVisor receives the data abort exception (❷)[4]. Then PVisor will add the subjected memory page (i.e., the memory page containing the SPT to be updated) to the dirty-list (❸), and enable the write permission of the memory page such that any further update will not be redirected to PVisor (❹). Lastly, when there are TLB invalidate or TTBR update, PVisor synchronizes its page table using secure OS's SPT only specified in the dirty-list (❺). In other words, SPTs not included in the dirty-list have never been changed, so those do not need to be synchronized. Note that PrOS does not perform this optimization for Level 1 SPTs to avoid modifying the secure OS (**P4**). Level 1 SPTs are much smaller than the page size (4 KB), so the subjected memory page is shared with other non-SPT data. Thus, PrOS cannot retrofit the efficient write monitoring mechanism based on the page table permission without modifying the implementation of the secure OS.

Moreover, PVisor also optimizes costly TLB invalidations by leveraging ASIDs. In particular, PVisor ensures that each secure OS uses different ASIDs when updating TTBR. Therefore, when there is context switching, PVisor does not need to invalidate TLB to prevent abnormal memory accesses through cached TLBs between different secure OS instances.

4. When instrumenting the code for trap and emulate for privileged instructions, PVisor also inserts SMC calls into the exception handler of secure OSes to trap data abort exceptions.

4.4.3 Device Virtualization

PrOS needs to virtualize devices since devices are now shared across multiple secure OSes. In general, device virtualization is often considered to be the primary cause of raising the complexity of virtualization. In the case of virtualizing secure OS, however, these OSes handle file I/O and networking by relying on the normal OS (i.e., **O3**), and thus PrOS does not need to virtualize such complex disk and network devices. Also, considering the common use-cases, trusted I/O devices should not be concurrently accessed (e.g., trusted display or fingerprint sensor should not be shared or concurrently accessed for its trustworthiness). Therefore, PrOS does not need to cater complex cases where more than one secure OS tries to use the same trusted I/O device at the same time, avoiding PrOS to employ an additional management module to guarantee fair uses of trusted I/O services.

Trusted I/O Devices. For trusted I/O devices, PrOS adopts the direct I/O mechanism [18] that assigns devices to secure OSes to use directly. To be more specific, PrOS's direct I/O scheme features following two key aspects: First, PVisor utilizes System MMU [19] (corresponding to IOMMU in x86) to limit the memory accessibility of the devices to the memory of the currently running the secure OS. This prevents potential DMA attacks. Second, PrOS saves and then restores the device's states per secure OS, such that each secure OS can seamlessly use the device even after other secure OS used the device.

By default, an activated secure OS is not allowed to access trusted I/O devices. When a secure OS attempts to use a device, PVisor promptly initiates the following device assignment process before the secure OS takes the control. To be more specific, PVisor first restores the device configuration status of the current secure OS (i.e., values in memory-mapped registers). Then PVisor properly configures System MMU to prevent DMA attacks. When the secure OS stops using a device or other secure OSes preempt the device, PVisor saves the device configuration for future restoration. This saving and restoration method appeared feasible for the devices used in our experiment, but it should be applied carefully in some devices holding hidden internal states. For example, LCD controller has a buffer between the frame buffer in DRAM and the actual screen, which is likely to hold a part of sensitive screen data. We can find a similar case from UART and its internal buffer. To prevent buffered sensitive data from being leaked to other secure OSes, they must be cleaned before handing over the control for the devices between secure OSes. Note that the saving and restoration method can only be implemented in a device-specific manner because there is no common way to save and restore device states. Fortunately, the amount of effort to develop the method may be acceptable as many trusted I/O devices, such as screen and fingerprinting sensor, have simple structure relatively. Also, even if devices are complicated, we believe that we can alleviate the difficulty with more sophisticated methods [20] that facilitate the easy saving and restoring of the internal state by referring the power management code performing the similar tasks for device suspension and restoration.

Security Resource Control Devices. PrOS emulates security critical devices (e.g., as mentioned in §2.1, this includes extended system bus, TZASC, and GIC), each of which contributes to partition the resource between the normal and secure world. Secure OSes try to configure these devices so as to protect their own security resources. In this case, PrOS should not allow the direct I/O scheme for secure OSes because these devices constitute the

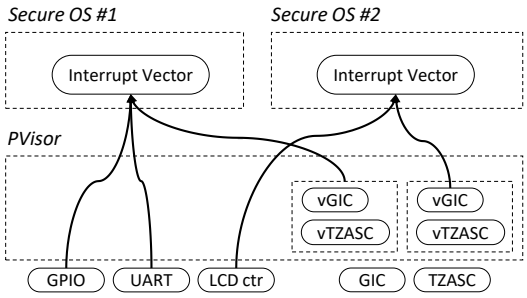

Fig. 9: Assignment and emulation for device virtualization.

primitive security guarantee of TrustZone as well as of PVisor. Toward this end, PVisor creates the virtual devices corresponding to each of those. Then it leverages trap-and-emulation mechanism (through invalidating memory mapped regions of those devices) to mediate all secure OSes' access to the devices. Finally, it emulates each device's functionality in light of security.

Interrupt Handling. By default, PVisor forces all interrupts toward the secure OS to be intercepted by setting `IRQ` and `FIQ` bits of `SCR`. If an interrupt is received, PVisor identifies which secure OS is responsible for the interrupt according to the device emulation and assignment status. If the secure OS is activated in the current core, PVisor instantly injects the interrupt to that active secure OS. Otherwise, PVisor first activates the destined secure OS before injecting the interrupt. After the secure OS finishes the interrupt handling, PVisor reactivates the previous secure OS.

5 Implementation

The prototype of PrOS was developed targeting 64-bit ARM. In this architecture, ARM provides the reference secure world software, called ARM-TF (Trusted Firmware), including a series of bootloaders with a secure boot mechanism, the power state coordination interface. In particular, the ARM-TF has a modular design in implementing a secure monitor in order to be collaborative with various secure OS developers.

PVisor is implemented based on the secure monitor of the ARM-TF by reusing the existing code and data structure (e.g., for per-core CPU states) and a build-system. As a result, PVisor was implemented in about 3K LoC, showing that PrOS has minimized the increase of TCB.

PDriver is implemented based on open source implementations of the TEE subsystem [21] and the CMA (Contiguous Memory Allocation) [22]. First, PDriver adds new APIs for PrOS and intervenes the existing APIs toward the legacy TZ driver upon the TEE subsystem, working as an intermediate proxy between host applications and the legacy TZ driver. Second, PDriver implements its physical memory allocation mechanism (§4.3.1) based on the CMA. PDriver tuned the CMA so as to prevent memory fragmentation considering TZASC's limited memory protection capability.

6 Case Study

We believe the protection provided by PrOS has practical impacts in running various trusted applications. To clearly demonstrate the feasibility of supporting these applications, we implemented and further thoroughly tested following three realistic applications heavily leveraging TrustZone's security guarantees: DRM player, Bitcoin wallet and safe vault. For the more realistic case study, these applications are developed to use trusted I/O devices (i.e., screen). Note that, the screen is also used in the normal world. In other words, when the TA is terminated, the secure OS returns a control for the screen to the normal OS. These applications contain security-critical parts, which handles sensitive data, such as copyrighted contents, financial data, and personal information. As compromises of these applications can lead to critical security issues, the app developers strive for strong security protection by placing the security-critical part in the isolated execution environments provided by the secure OS. In applying this protection method, however, there are two issues that may raise app developers' security concerns. First, the secure OS beneath their TAs can be compromised by other developers' malicious TAs launching privilege escalation attacks. Second, in many cases, device vendors disallow dynamic TA installation in the secure world (mainly because of the first issue). Thus, it is difficult for the app developers to install and execute their TAs in the secure world. We note that PrOS can resolve the above-mentioned issues by allowing every app developer to privatize their own secure OS and run their TAs in fully isolated from each other at OS-level.

DRM Player. A DRM player is an application that is responsible for playing the copyright-protected contents for users. To implement a DRM player, as DRM contents are delivered as encrypted, we directly forwarded the contents to the TA (implementing the DRM player). Then we implemented the TA as follows: (1) It decrypts and decodes the DRM contents only in the secure memory, which is located within the privatized secure OS, such that other TAs cannot access this secure memory; and (2) It writes extracted raw video frames to the secure screen buffer to be displayed to the user through a trusted screen. Here, since PrOS supports trusted I/O (while ensuring the exclusive access), this screen buffer writing can be trusted in the same way as running the single secure OS.

Bitcoin Wallet. A Bitcoin wallet is an electronic payment application that supports peer-to-peer tradings. To harden this application, we developed the TA performing the following tasks: (1) authenticates the user through the password typed using trusted I/O; (2) signs transactions with a private key stored in a secure storage; and (3) reports the transaction results to the user through a trusted screen. We were able to successfully implement all of the aforementioned tasks, since PrOS executes TA on the privatized secure OS holding its own exclusive memory and supporting both trusted I/O and file I/O (i.e., secure storage).

Safe Vault. Mobile devices often store a variety of personal contents such as photos and documents. A safe vault is an application that protects these sensitive contents by only storing the encrypted contents in an isolated storage (such as TrustZone). For this application, we developed the TA that temporarily decrypts the contents only when there are authorized accesses (i.e., PIN checks) and displays it to the user through a trusted screen. Similar to the case of the DRM player, we were able to support this TA in the privatized secure OS using PrOS, augmenting the security level for the safe vault.

7 Evaluation

This section evaluates the prototype of PrOS by measuring performance overhead and analyzing security. Experiments have been conducted on the versatile express V2M-Juno r1 platform [23], which has Cortex-A57 1.15 GHz dual-core processor and Cortex-A53 650 MHz quad-core processor in a big.LITTLE architecture

and 6 GB of DRAM. We used Android 7.1.2 with Linux kernel 4.4.71 as the normal OS, and OP-TEE 2.5.0 [10] as the secure OS.

7.1 Achieving the Design Goals

We study how well PrOS satisfies its design principles listed in §4.1. First, PrOS achieves **P1** (i.e., isolated TrustZone service) as it strictly isolates each secure OS from others using software-based virtualization techniques. Second, PrOS satisfies **P2** (i.e., minimum changes in existing software), as it is compatible with the existing software. In the normal OS, PrOS only requires to add PDriver to it because PDriver is the device driver and can be co-located with the unmodified legacy TrustZone driver. In the case of the host application, PrOS only adds interface calls for loading and unloading the secure OS, but it does not affect the programming model for other API calls. In the secure world, PrOS implements Pvisor solely at EL3, so it is entirely transparent to the secure OS (running at EL1). In the case of the secure OS, PrOS inserts several SMC calls for trap-and-emulate in privilege instruction or to the exception handler, but does not make any significant code modifications. PrOS satisfies **P3** (i.e., minimize the TCB) as its TCB only includes PVisor where its implementation complexity is 3K LoC (§5). We note that this size is only 3% of OP-TEE OS. Such a small code base is attributed to the fact that unlike the secure OSes having the responsibility for implementing all functionality to run TAs (i.e., management for TA, dynamic memory, peripheral devices, file/network, etc), PrOS only focuses on providing security to secure OSes by separating execution environments. Compared to the commodity hypervisor, PrOS is still much smaller (i.e., Xen hypervisor is 8,477k LoC [24] and NOVA microhypervisor is 36k LoC [25]). We deem that it is due to the secure OS-aware virtualization of PrOS that reduces the implementation overhead for CPU/memory/IO virtualizations. It is worth noting that PDriver is not part of the TCB, as all of its runtime behaviors are strictly verified by PVisor. PrOS meets **P4** (i.e., low overhead): PrOS incurs near-zero overhead to the normal world execution as we further describe in the next subsection (§7.2). PrOS involves moderate performance impacts on the secure OS (1.18% on average) due to the virtualization.

7.2 Performance Evaluation

To measure the overhead of PrOS, we experimented with following two cases: (i) *Native*, which denotes the traditional case that a single secure OS runs within TrustZone; and (ii) *PrOS*, which denotes the case where two secure OSes run using PrOS. In both cases, we used OP-TEE OS as secure OS.

First, we examined how efficiently PrOS and OP-TEE OS can manage secure OSes and TAs. To understand this, we measured the execution time of the management APIs provided by PrOS and OP-TEE OS, respectively. Second, we investigated the performance impact of PrOS to the system. Specifically, we ran the the official test suits of OP-TEE and several synthetic application benchmarks to understand the performance degradation of the TAs and the normal OS. We repeated these experiments 20 times and presented averaged results.

PrOS APIs. As described in §4.3.4, PrOS newly provides two APIs for host applications and three APIs for PDriver. To understand the performance impact of these APIs, we implemented a host application that invokes these APIs, and measured the elapsed time of each API call in the application and PDriver, respectively. While these experiments were conducted using synthetic host applications, we believe the experiments are reasonable as the APIs are designed to perform the same task regardless of the host applications. The results are shown in Table 1. The elapsed time of `PD_LoadSecureOS`, which loads a secure OS, is only 1.5% of the real boot time of OP-TEE OS, showing the effectiveness of Zygote secure OS based installation as presented in §4.3. Compared to the APIs of OP-TEE OS (Table 2), PrOS's APIs in loading and unloading a secure OS are noticeably slower (about 100 times) than `InitializeContext` and `FinalizeContext` APIs that manage the connection of host application and TA in OPTEE. However, since these APIs are rarely invoked (so as to manage its secure OSes) and even very short compared to the startup time (2000ms on average) of most of normal applications [26], we believe this slowdown would not severely impact the practical use-cases of PrOS.

API name	time(μs)
PD_LoadSecureOS	11799.5
PD_UnloadSecureOS	10383.0
PV_LoadSecureOS	9676.2
PV_ActivateSecureOS	3.1
PV_UnloadSecureOS	9686.8

TABLE 1: Elapsed time of PrOS APIs

API Name	time(μs)		Overhead
	Native	PrOS	
InitializeContext	41.9	42.7	1.8%
FinalizedContext	12.2	12.1	-0.4%
OpenSession	15738.0	15620.4	-0.7%
CloseSession	5482.4	5508.1	0.5%
AllocateSharedMemory	21.9	21.9	0.0%
RegisterSharedMemory	21.2	21.5	1.4%
ReleaseSharedMemory	11.6	11.7	0.9%
InvokeCommand	96.2	109.2	13.5%

TABLE 2: Performance of GlobalPlatform Client APIs

OP-TEE APIs. OP-TEE OS adopts the standard API specification published by GlobalPlatform [27], [28]. The specification defines two sets of APIs: GlobalPlatform client APIs, which are invoked by host applications to manage TAs; and GlobalPlatform internal core APIs, which are invoked by TAs to use OS services. As PrOS runs secure OS on top of PVisor, it would incur overheads for both of these API groups. We measured such overheads (shown in Table 2) in the same way as in the PrOS's APIs. We found that `InvokeCommand` showed noticeable overheads compared to other APIs. This is because, due to the implementation choices of OP-TEE OS, `InvokeCommand` involves the full page table creation, which in turn causes the sync operation of the shadow paging scheme in PrOS. We would like to point out that this slowdown would not severely impact the overall performance of TrustZone's service. The actual computation/processing time (such as encryption, decryption, and signing operations performed by TA) will be a dominating factor for the performance as it takes much longer than the communication time impacted by `InvokeCommand`.

The GlobalPlatform internal core APIs includes a various set of APIs, including trusted core framework APIs, trusted storage APIs, cryptographic operations APIs, time APIs, and TEE arithmetic APIs. As shown in Table 3, PrOS only introduces lightweight overhead (2.1% in the worst case), suggesting that PrOS would be efficient enough to handle such various tasks.

Performance Impacts on TA. As PrOS intervenes all operations of the secure OS, it may introduce a performance penalty on TAs.

Name	time(μs)		Overhead
	Native	PrOS	
Trusted Core Framework API	15912.4	16035.0	1.8%
Trusted Storage API	8615.2	8640.2	2.1%
Cryptographic Operations API	6953.0	7142.4	1.8%
TEE Arithmetic API	2950.0	3008.0	0.0%
Time API	3248.0	3248.0	1.6%

TABLE 3: Performance of GlobalPlatform Core APIs

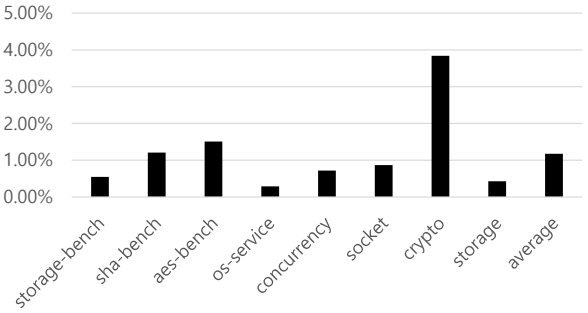

Fig. 10: Performance Overhead of `xtest`. The suffix "bench" means that the associated test group performs a stress test.

Benchmark	time(μs)		Overhead
	Native	PrOS	
GeekBench 3.4.1			
single core	412.3	408.0	-1.0%
muti core	1524.0	1524.3	0.0%
Vellamo 3.2			
productivity	3886.2	3919.8	0.9%
metal	1194.2	1199.8	0.4%
Antutu 6.0.1	28990.3	28941.2	-0.2%

TABLE 4: Performance overhead on the normal world

To understand this performance aspect, we ran official test suite of OP-TEE OS, `xtest` [29], which includes various tests. More specifically, `xtest` consists of a main host application and many TAs that are implemented to perform each test. To clearly represent the results, we grouped 39 tests into eight groups and presented the performance overhead per group (shown in Figure 10). Overall PrOS imposed 1.18% on average (ranging from 0.29% to 3.84%), implicating that PrOS would not introduce noticeable performance overheads to typical TA workloads.

Performance Impacts on Normal World. One of key advantages of PrOS is that it never causes performance degradation in the normal world, as its virtualization is carried out in the secure world. This advantage is especially important considering the fact that mobile devices are mostly operating in the normal world. To demonstrate this, we experimented with three synthetic benchmarks, such as GeekBnech, Vellamo, and Antutu, that measure the overall system performance. As shown in Table 4, PrOS imposes negligible overheads, demonstrating that PrOS does not slow down the normal world to privatize secure OS.

7.3 Case Study Evaluation

As described in §6, we performed a series of case studies on the DRM player, the Bitcoin wallet, and the safe vault. The case studies suggested that these applications can be hardened by partitioning their security critical parts into TAs and securely running in secure OSes fully supported by PrOS. In this subsection, we study the performance aspects of these case studies.

In the DRM player, we measured how fast the TA is to decode one frame of the video clip and copy to the secure frame buffer. In the experiment, we used the video clip of the resolution of 360p. The result showed that, in the big cores, the TA finished its task in 14.15 ms and 14.17 ms, respectively, without or with PrOS. In the little cores, the TA showed 30.29 ms and 30.36 ms, respectively, without or with PrOS. We believe this results advocates the efficiency of PrOS, as it showed near-zero overheads. We additionally noticed that, if the same task of TA is implemented in the normal world, it can run much faster: 9.43 ms in the big cores and 24.88 ms in the little cores. Based on our analysis, we found that this is because of the request-response model of the secure OS. While the TA is running in the secure world (including both before and after PrOS), the execution is often interrupted by unexpected interrupts, such as timer interrupts. To handle the interrupts, each of them should be transferred to the normal world, frequently incurring the costly world switches. We believe this is not the limitation of PrOS, but the limitation of TrustZone, which can be mitigated in the future by minimizing the world switch latency between the normal world and the secure world.

	From Normal world	From Secure world
Secure OS	12	70
PVisor	3	10

TABLE 5: The number of external interfaces of secure OS and PVisor.

In the Bitcoin wallet, we measured the execution time of the tasks of its TA: the transaction report task. According to our result, the execution time showed 100.1μs and 100.8μs (without or with PrOS, respectively). These measurement results also show the efficiency of PrOS as PrOS showed almost no different results from non-privatized secure OS.

In the case of the safe vault, we measured the time taken to decrypt an encrypted image file (4.1 MB) and further display through trusted I/O. The result showed 91.5μs and 92.1μs, respectively, when PrOS is deactivated and activated. This also confirms the efficiency of PrOS.

7.4 Security Evaluation

The primary goal of PrOS is to allow app developers to execute their TAs securely. The key idea of PrOS to achieve this goal is to privatize secure OS to let each app developer have their own isolated execution environment. In this subsection, we first discuss the possible attack vectors and the security guarantees provided by PrOS for each vector, and second perform a comparison with the legacy TrustZone in terms of security.

7.4.1 Possible attacks against PrOS

First, adversaries may attempt to attack PVisor from the normal world (i.e., PDriver). Since PrOS assumes that PDriver can be adversarial or compromised, PDriver can launch various attacks against PVisor. More specifically, attackers may attempt to exploit a vulnerability in PVisor (such as memory corruption or semantic vulnerabilities [30]) through invoking the SMC call with crafted parameters. We argue that spotting a vulnerability in PrOS would be difficult as PVisor is small (i.e., 3k LoC). We acknowledge that this may be a strong enough argument, and PrOS may leverage memory-safe languages or verified kernel approaches [31] to implement PVisor in the future.

Second, attackers may attempt to exploit PVisor from the secure world (i.e., TA). Because PrOS provides secure OSes for individual

application developers, attackers can install a TA inside the secure world via PrOS. If they know a vulnerability in the secure OS, they can compromise the secure OSes through a well-known privilege escalation attack. However, even though the attackers take control of a secure OS and have an ability to execute instructions at Secure EL1, it is difficult for them to negatively affect other secure OSes due to the CPU virtualization technique of PVisor. There remain no security-critical privileged instructions in the secure OS because all of them were replaced with SMC calls. Also adding new instructions to the secure OS and accessing the memory region of PVisor from the secure OS are also prevented completely. Therefore, after acquiring the execution context of the secure OS, the attackers should compromise PVisor, similar to the attack by PDriver, but their attempts would be blocked as we have mentioned before.

7.4.2 Comparison with the legacy TrustZone

We admit that the legacy TrustZone that runs all TAs in a single secure OS also can ensure such isolated executions of each TA as long as the secure OS remains intact and safe against attackers. However, protecting secure OS is much harder than protecting PVisor in terms of the size of attack surface. We reasonably assume that the attack surface of the secure OS and PVisor each will be expanded in proportion to the number of their external interface, such as system calls or SMC calls, that can be used to inject malicious payloads and the size of their code base that is directly connected to the possibility of vulnerabilities being existed. Table 5 summarizes how many external interfaces are implemented for PVisor and the secure OS, respectively. Consequently, we can observe that PVisor has 74.1% smaller number of external interfaces than the secure OS, and from this fact it would be difficult for the attackers to inject malicious payloads into PVisor than into the secure OS. In addition, the code size of PVisor (3k LoC) is only 3% of that of the secure OS, thus creating exploitable payloads in PVisor would be more difficult than in the secure OS.

7.5 Limitations and Discussion

PrOS focuses on providing TAs a separate secure OS to ensure isolated execution. To accomplish such a secure OS privatization, PrOS takes advantage of the Zygote mechanism and the virtualization mechanism. However, the current implementations of these mechanisms lead to some limitations PrOS is facing.

More Precise Trap-and-Emulation. To conduct the trap-and-emulation, PVisor replaces all privileged-instructions in the secure OS' code to SMC calls during the boot process. For correct replacement, PVisor needs to be able to recognize where the instructions are located. Luckily, because secure OS is implemented putting a priority on security, PVisor can easily find the code page by identifying the executable-bit from the page table of the secure OS. However, it may be less precise about certain code pages where instructions and constants coexist. Although problems related to such code-constant mixed pages have not been observed in our experiments, we may need to improve the precision of the instruction replacement with the help of well engineered dynamic binary instrumentation systems [32].

No Support for Dynamic Code Loading. Holding the exclusive control over the page table of secure OSes, PVisor constantly enforces the write-execute-never policy. After being forked, therefore, the secure OSes are not allowed to load a new module dynamically or to perform self-modification to their code. Fortunately, we may be able to relieve this limitation by thoroughly applying the trap-and-emulation technique on the newly modified or added code page.

Homogeneous Secure OS. Thanks to the Zygote mechanism, we greatly reduce the loading time of secure OS when a new TA is spawned. However, since all secure OSes are loaded by being forking from the same secure OS created in the boot time, this mechanism results in all secure OSes in PrOS becoming homogeneous. We believe that this fact would be acceptable if once TAs are developed based on the standard APIs, such as those of GlobalPlatform, rather than using unique APIs supported only in a specific secure OS.

With respect to the homogeneity of secure OS in PrOS, all security OSes will share the same vulnerability, which can lead to a security concern that an attacker might be able to compromise all secure OS instances in the same way. It is laborious, however, because with PrOS each TA runs on a separate secure OS so that the attacker has to launch tailored privilege escalation attacks as many as the number of TAs. Also, we will be able to improve the zygote mechanism through diversification techniques [33], thereby preventing secure OSes from having the same vulnerability.

Hardware-supported Virtualization on TrustZone. ARM recently announced ARMv8.4-a architecture which includes Secure EL2 for virtualization on TrustZone. Although the detailed specification about Secure EL2 is not yet available, it will contain similar hardware supports for memory/IO virtualizations as the existing EL2. Therefore, we predict that if a hypervisor based on Secure EL2 is implemented, it will facilitate more efficient and transparent implementation of TrustZone virtualization than PrOS. However, we believe that PrOS and its software-based TrustZone virtualization will be still worthy considering many existing devices and budget devices that are implemented on the conventional ARM architectures.

Communication between TAs. As mentioned in §3, we assume that all TAs are not trusted each other because different developers develop them. Thus, PrOS does not provide a mechanism for direct communication between secure OSs for each TA in the secure world. However, some TAs may need to communicate with other TAs (for example, a TA of a device vendor providing fingerprint services). At this point, TAs can communicate with different TAs indirectly through the normal world, and supporting direct communication between TAs will be our future work.

8 Related work

Isolated Execution Environment. Building isolated execution environments to ensure secure execution of TAs has long been a widely studied topic in security. TrustVisor [34], InkTag [35], Overshadow [36], Sego [37], OSP [38], PrivateZone [39], and Wimpy-Kernel [40] established isolated execution environments based on hypervisors. TrustShadow [41], ObC [11], [13], and TLR [42] achieved the same goal by using TrustZone. However, the security guarantees provided by all the aforementioned systems might be compromised by a malicious TA launching privilege escalation attacks because they assume the execution model that all TAs run on top of a single privileged software layer corresponding to the secure OS in TrustZone. We believe that such security concern these systems are facing can be mitigated by applying PrOS and by allowing each TA to run separately without sharing the same privileged software layer.

Formal Verification. If it were proved that a secure OS has no security vulnerability by applying the formal verification mechanism [31], a malicious TA running on the secure OS would not be able to compromise other TAs across the isolation boundary set by the OS. However, this mechanism requires certain constraints on kernel implementation and design. For example, to eliminate non-deterministic events, there are various restrictions such as disabling most interrupts or limiting the use of function pointers. As a result, these limitations make it difficult to apply the formally proven security mechanisms to an existing secure OS.

TrustZone Virtualization in the Normal World. Terra [43] provides strong protection to TAs by allowing each of them to run on top of different OSes by leveraging hypervisors. We can use this approach to achieve the goal of PrOS by (1) virtualizing TrustZone in the normal world and (2) installing secure OSes in the virtualized TrustZone environments. However, this approach causes problems regarding security and performance. In this approach, the hypervisor is the TCB of secure OSes, but historically the hypervisor suffered from various security vulnerabilities [44], as its implementation is complex by various functionality. This approach also can impose a non-negligible performance burden. As the hypervisor intervenes in all accesses to the underlying hardware resources, this approach introduces unavoidable overheads. We highlight that such overheads are impacting the entire software components running in the normal world (including normal OS and its running applications) because the hypervisor should always be turned on even if TrustZone services are not actively used.

TrustZone Virtualization in the Normal World with Security Module in the Secure World. TrustICE [1] has built isolated execution environments for secure OSes in the normal world. TrustICE dynamically sets TZASC not only to protect secure OSes from an untrustworthy normal OS in the normal world but also to isolate secure OSes from each other. One thing to note is that as TZASC is located between CPU cores and DRAM, any protection configuration of TZASC affects all CPU cores. It means that if one core runs a secure OS in a multi-core environment, the other cores with separate secure OSes or even the normal OS must be suspended, severely limiting the scalability of TrustICE. However, in PrOS, TZASC is used only to block access to secure OSes from the normal world, and PVisor provides isolation between secure OSes through TrustZone virtualization. Therefore, PrOS can avoid such a scalability issue, unlike TrustICE. vTZ [2] proposed another approach that can realize the goal of PrOS. In this approach, a hypervisor is still employed to virtualize TrustZone in the normal world, but the difference is that it is excluded from the TCB as follows: (1) a security module is installed in the secure world that is isolated by the real TrustZone from the hypervisor, (2) the security module supervises and controls the behaviors of the hypervisor (i.e., the execution flow and the resource management). As a result, secure OSes in the virtualized TrustZone are still secure even after the hypervisor is compromised because the TCB belongs to the security module in the secure world. However, this approach still causes the performance problem, especially in mobile devices that have almost turned off the hypervisor, because all software components in the normal world are subject to continuous performance degradation. Even worse, this approach suffers from a compatibility issue. As the hypervisor (in the normal world) must be monitored by the security module (in the secure world), there is no transparent way to enforce the non-bypassable supervision of the security module against the hypervisor. Therefore, this approach requires many modifications to the legacy hypervisor that is continuously evolving due to the changes in hardware/software requirements, arguably rendering it an impractical solution.

9 Conclusion

ARM TrustZone is a promising security technique providing hardware-assisted privilege isolation for computing resources. However, since all trusted applications are running on the same secure OS, the secure OS becomes the single point of security failure. This paper realized the light-weight secure OSes through virtualization, each of which runs a trusted application. As such, compromising one secure OS does not implicate that the entire TrustZone is subverted, augmenting the security level of TrustZone. According to the evaluation of the prototype of PrOS, PrOS is not only performance effective but also demonstrated that it can support various trusted applications, ranging from Bitcoin wallet to DRM player, without imposing compatibility issues.

Acknowledgement

This work was supported by Institute of Information Communications Technology Planning Evaluation (IITP), a grant funded by Korea government (Ministry of Science and ICT) (no. 2016-0-00078, Cloud Based Security Intelligence Technology Development for the Customized Security Service Provisioning). This work also was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2017R1A2A1A17069478, NRF-2018R1C1B5086364, NRF-2018R1D1A1B07049870).

References

[1] 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.

[2] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing arm trustzone," in *USENIX Security Symposium*, 2017.

[3] D. Shen, "Attacking your trusted core: Exploiting trustzone on android," in *Black Hat USA*, 2015.

[4] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices." in *USENIX Security Symposium*, 2016, pp. 549–564.

[5] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proceedings of the 17th Conference on Security Symposium*, 2008.

[6] E. Solutions, "Analysis tools for ddr1, ddr2, ddr3, embedded ddr and fully buffered dimm modules," 2014, http://www.epnsolutions.net/ddr.html.

[7] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1675–1689.

[8] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 1–12.

[9] S. H. Kim, D. Han, and D. H. Lee, "Predictability of android openssl's pseudo random number generator," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 659–668.

[10] Linaro, "Op-tee: Open source trusted execution environment." 2017, https://www.op-tee.org/.

[11] 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.

[12] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2006.

[13] J.-E. Ekberg, K. Kostiainen, and N. Asokan, "Trusted execution environments on mobile devices," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1497–1498.
[14] A. Fitzek, F. Achleitner, J. Winter, and D. Hein, "The andix research osâĂŤarm trustzone meets industrial control systems security," in *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*. IEEE, 2015, pp. 88–93.
[15] "Xen," https://www.xenproject.org/.
[16] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, pp. 412–421, 1974.
[17] C. A. Waldspurger, "Memory resource management in vmware esx server," in *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, ser. OSDI '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 181–194. [Online]. Available: http://dl.acm.org/citation.cfm?id=1060289.1060307
[18] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang, "Towards high-quality i/o virtualization," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009, p. 12.
[19] ARM, "System memory management unit (smmu)," http://www.arm.com/products/system-ip/controllers/system-mmu.php.
[20] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, 2013.
[21] "generic tee subsystem," https://lwn.net/Articles/674280/.
[22] "Contiguous memory allocation," https://lwn.net/Articles/396702/.
[23] ARM, "Versatile express juno r1 development platform," in *ARM 100122_-0100_00_en*, 2015.
[24] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 203–216.
[25] U. Steinberg and B. Kauer, "Nova: a microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 209–222.
[26] J. Yang, "Cold start times: An analysis of top apps," http://blog.nimbledroid.com/2016/02/17/cold-start-times-of-top-apps.html.
[27] GlobalPlatform, "Tee client api specification v1.0."
[28] ——, "Tee internal core api specification v1.1."
[29] "optee-test," https://github.com/OP-TEE/optee_test.
[30] 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," 2017.
[31] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.
[32] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 261–270.
[33] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 424–439.
[34] 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.
[35] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *ACM SIGARCH Computer Architecture News*, vol. 41. ACM, 2013, pp. 265–278.
[36] 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 SIGARCH Computer Architecture News*, vol. 36. ACM, 2008, pp. 2–13.
[37] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive trusted metadata for efficiently verified untrusted system services," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 51. ACM, 2016, pp. 277–290.
[38] Y. Cho, J.-B. 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 *USENIX Annual Technical Conference*, 2016, pp. 565–578.
[39] 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.
[40] Z. Zhou, M. Yu, and V. D. Gligor, "Dancing with giants: Wimpy kernels for on-demand isolated i/o," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 308–323.
[41] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," 2017.
[42] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *ACM SIGARCH Computer Architecture News*, vol. 42. ACM, 2014, pp. 67–80.
[43] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 193–206.
[44] "Xen: Vulnerability statistics," http://www.cvedetails.com/vendor/6276/XEN.html.

Donghyun Kwon received the BS degree in Electrical and Computer Engineering from the Seoul National University, Korea, in 2012. He is currently working toward the PhD degree in Electrical and Computing Engineering from the Seoul National University, Korea. His research interests include system security against various types of threats.

Jiwon Seo received the BS degree in Electrical and Computer Engineering from the Seoul Women's University, Korea, in 2016. She is currently working toward the PhD degree in Electrical and Computing Engineering from the Seoul National University, Korea. Her research interests include system security against various types of threats.

Yeongpil Cho received the BS degree in Electrical Engineering from the POSTECH, Korea, in 2010. He received the PhD degree in Electrical and Computer Engineering from the Seoul National University, Korea, in 2018. Currently, he is a professor in the School of Software at Soongsil University. His research interests include system security against various types of threats.

Byoungyoung Lee received the BS and MS degrees in Computer Science and Engineering from POSTECH, Korea in 2009 and 2011, respectively. He received the PhD degree in computer science from the Georgia Institute of Technology in 2016. Currently, he is a professor at the Department of Electrical and Computer Engineering, Seoul National University, Korea. He is interested in all computer security and privacy related problems in general. In particular, his research focus is in system security, e.g., designing and implementing secure systems through eliminating vulnerabilities and mitigating attacks.

Yunheung Paek received the BS and MS degrees in Computer Engineering from the Seoul National University, Korea in 1988 and 1990, respectively. He received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1997. Currently, he is a professor at the Department of Electrical and Computer Engineering, Seoul National University, Korea. His research interests include system security with hardware, secure processor design against various types of threats, and machine learning based security solution. He is a member of the IEEE.