3rdParTEE: Securing Third-party IoT Services using the Trusted Execution Environment

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

Abstract—Advancements in the Internet of Things (IoT) have resulted in the connection and deployment of numerous smart and embedded devices. Although such devices enable various services such as smart grids, they attract more attackers to the IoT world. A trusted execution environment (TEE), which can be created by using TrustZone technology, is a promising security artifact for protecting critical operations and sensitive data of IoT devices. Unfortunately, although TrustZone is available in most ARM architecture-based devices ranging from microcontrollers to high-end smart devices, it has not been widely adopted by third-party IoT service providers because of its limited access. That is because the TEE is maintained by the TEE platform vendors to preserve its security. Therefore, third parties must adhere to strict policies and procedures to ensure the deployment of trusted services in the TEE. This aspect hinders the fast development and deployment of IoT services. In this work, we propose 3rdParTEE to address this problem by enabling third-party IoT service providers to readily run their trusted services, thereby minimizing their dependency on the TEE maintainers. Specifically, 3rdParTEE facilitates the secure running of the third-party's native kernel driver in the TEE without hampering the security of the existing TEE components. To demonstrate the effectiveness of our approach, we ran three kernel drivers for maintaining the IoT services platform (e.g., kernel integrity check) in the TEE. Additionally, during the performance evaluation, we observed a reasonable performance overhead of up to 7% when running the kernel drivers in such a secure manner.

Index Terms—Trusted Execution Environment, IoT Security

1 Introduction

THE Internet of Things (IoT) industry continues to grow. According to Ericsson [1], the number of IoT devices is expected to increase to 24 billion by the year 2050. Supporting this expectation, IoT devices for everyday life, including in-home entertainment, critical infrastructure, and industry, continue to emerge [2], [3]. Unfortunately, this situation makes IoT devices attractive targets for attackers [4]. When it comes to IoT devices, the tendency that security is less prioritized than fast deployment and productivity might attract more attackers to the IoT world. It has been proven that the vulnerabilities of IoT devices can be abused to constitute IoT botnets [5], [6], [7], [8] and take such devices hostage after they have been released to the public [9], [10].

A trusted execution environment (TEE), which isolates the execution context from a rich execution environment (REE), can be a promising technique for securing IoT devices. In other words, security critical logic and operations, such as cryptographic operations and firmware recovery, can be effectively isolated, even if the operating system in the REE is compromised. ARM TrustZone is one such TEE technique, which is available on ARM architecture-based platforms ranging from small IoT devices to servers [11]. Researchers have proposed various security applications using TrustZone, including a virtualized TEE for cloud servers [12], a one-time password for mobile devices [13], and secure advertising [14]. Specifically, Samsung utilizes TrustZone to protect the operating system's (OS) kernel [15] in a manner that deprivileges the OS and enforces the emulation of its critical operations in the TEE, thereby ensuring the immutability of the OS. Apart from its application in mobile and server environments, TrustZone has also gained attention in the IoT field. As a result, researchers have designed TrustZone-based security measures for IoT devices [16], [17].

Although TrustZone enables the system to benefit from the TEE, the accessibility of third-party application developers and IoT service providers to TrustZone is limited. This is because the TEE belongs to the system on a chip (SoC) manufacturers [18], [19], [20] or TEE software platform providers [21], [22], [23]. The TEE is generally a paid service that is supported based on membership. TEE clients must develop trusted applications (TAs) using the TEE application programming interfaces (APIs) provided by a specific TEE vendor. Additionally, such TAs are strictly verified by TEE vendors to prevent the deployment of a new TA with an exploitable bug. These TEE features increase the time required for the development and deployment of new or updated trusted IoT software, and they conflict with the time-to-market attributes of IoT services. We argue that this is the major reason why third-party IoT service providers are reluctant to use TrustZone.

In this study, to minimize the disadvantages of TEEs, we propose 3rdParTEE, which is a TEE platform that enables third parties to readily use the TrustZone-based TEE. 3rdParTEE aims to (1) ensure the secure running of third-party software without bloating the existing TEE and without introducing a new attack surface to the TEE. It also aims to (2) minimize the use of separate APIs to develop trusted applications. Specifically, we propose an approach

- *B. Kang is with Korea Advanced Institute of Science and Technology. E-mail: brentkang@kaist.ac.kr*
- *J. Jang is with Chungnam National University. E-mail: jisjang@cnu.ac.kr*

Corresponding author: Brent Byunghoon Kang

for protecting kernel-privileged software, such as a loadable kernel module (LKM), using the TEE, which differentiates our work from previous studies that focus on protecting user-level applications [17], [24], [25], [26]. We expect third-party IoT service providers to benefit from 3rdParTEE in a manner that ensures the secure OS-privileged operations. For instance, any update to an OS whose integrity is protected by the TEE-based kernel integrity monitor [15], [27] generally requires the TEE vendor to verify the update and apply it to the OS image. However, 3rdParTEE enables the third party to serve urgent device management operations, such as the immediate fixing of OS bugs *in memory*, without waiting for the vendor to complete verification and apply the change to the OS image.

3rdParTEE consists of two parts: the REE component and the TEE component. In the REE side, we implemented a simple loader that communicates with the 3rdParTEE component in the TEE side to send a request for the secure execution of the kernel module provided by the third party. To handle this request, the TEE component of 3rdParTEE first verifies the signature and integrity of the provided kernel module based on an asymmetric cryptographic algorithm. It then shields the execution of the kernel module using the TEE. To this end, we designed 3rdParTEE to leverage the TrustZone address space controller (TZASC), which enables filtering access to a TEE memory region. Finally, although we assume that IoT service providers are not malicious, any programming errors that might affect the existing TEE platform's security, such as mispatching the TEE memory instead of the REE OS, must be considered. To address this issue, we ensure that any access to the TEE memory region by the shielded module is monitored by setting hardware debugging watchpoints, as demonstrated in previous studies [28].

We implemented our 3rdParTEE prototype using an Arndale board equipped with a 32-bit ARM dual-core processor and 2 GB of memory. To evaluate the performance of 3rdParTEE, we ran three Linux-based LKMs under the protection of 3rdParTEE. Each module conducts OS management operations, such as hashing the REE OS region, patching kernel memory, and traversing dynamic kernel data structures. Owing to additional operations, such as the integrity check and the communication between the REE and the TEE, running a kernel module protected by 3rdParTEE imposed a maximum overhead of 7%.

The contributions of this study are as follows:

- 3rdParTEE enables the isolation of privileged software, such as a LKM, from an untrusted REE OS. This differentiates our work from previous TrustZone-based work that is limited to protecting user applications.
- Developers can simply follow conventional kernel module development practices to create 3rdParTEE-protected software. This aspect exempts the need to use TEE-dedicated APIs, thereby reducing development effort.
- Various urgent device management operations that specifically require TEE privileges can be conducted by IoT service providers on the fly without waiting for the TEE vendor's intervention, which could inhibit the time-to-market requirements of IoT devices.

2 Background

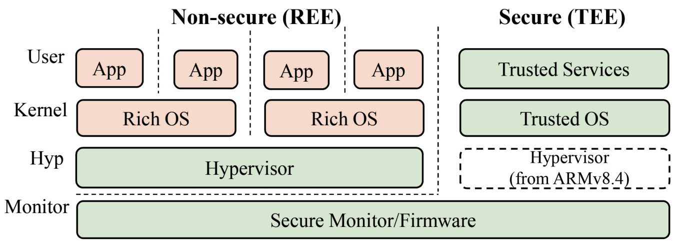

Fig. 1: ARM architecture with TrustZone.

2.1 ARM TrustZone

ARM TrustZone is a security extension of ARM processors. Through TrustZone technology, a system can be partitioned into two execution environments: the REE and the TEE. General software stacks, such as those based on Linux and Android, are hosted in the REE. On the other hand, the TEE protects security critical applications, such as cryptographic services and digital rights management (DRM). Various hardware components are supported to ensure isolation between these environments. First, ARM processors define different security states: secure and non-secure states. In non-secure states, the user, kernel, and hypervisor modes are supported. In secure states, the user, kernel, and monitor modes are available. Notably, as it pertains to the monitor mode, the running software manages the switching between the environments by saving and restoring the context of each environment. In addition to the processor's security state, the TZASC is provided to partition the DRAM for each environment. Using the TZASC, we can define several physical memory regions and their access permissions depending on the processors' security states. For example, a specific range of the physical memory can be configured as a TEE region by allowing its access from the TEE (secure state) but not from the REE (non-secure state). In addition to security state and memory access control, TrustZone also protects input/output (IO) peripherals, such as displays and keypads. The TrustZone protection controller (TZPC) enables the creation of a secure IO channel by dynamically connecting such peripherals to the TEE, thereby preventing interference resulting from the compromised REE.

2.2 Virtual Memory System on ARM

ARM processors support virtual memory systems to translate virtual addresses to physical addresses. Such translations are performed based on page tables whose base addresses are configured using translation table base registers (TTBRs). Generally, there are two architecture-supported TTBRs: TTBR0 and TTBR1. In 64-bit systems, TTBR0 and TTBR1 generally contain the page table base addresses for the user and the kernel space, respectively. In 32-bit systems, because of their small address range, i.e., 4 GB, only one page table can be used. For example, a 32-bit Linux-based system can be configured to use TTBR0 to map the entire address space covering the user and kernel spaces. In this case, the settings of the translation table base control register (TTBCR) determine whether one of the TTBRs is leveraged or whether both TTBRs are leveraged. Specifically, when the short-descriptor format [29] is used, the N flag in the TTBCR determines the size of the address' translation conducted through the page tables pointed to using TTBR0 and TTBR1. For example, if the N value is 0b000, only TTBR0 is used. Otherwise, both TTBR0 and TTBR1 are leveraged, and the address ranges translated using each TTBR are determined by the N value. For instance, if the N value is set to 0b001, the addresses that are under and over 2 GB of virtual memory are translated using TTBR0 and TTBR1, respectively. The TTBRs and TTBCR are banked registers for non-secure and secure states of the processor. Therefore, they can be independently configured in the REE and in the TEE.

2.3 IoT Device Protection using TrustZone

To ensure the security of the TEE, its access is restricted to and managed by a few stakeholders, such as SoC designers, device manufacturers, and secure OS providers, who have control over critical hardware (e.g., eFUSE and TZASC) and software (e.g., secure OS). Based on the dominion of TEE, various TEE-based IoT security frameworks and solutions have been introduced. For example, Samsung's TrustZone-based mobile device security technologies, known as KNOX and TIMA [15], are revisited and similarly applied to IoT devices, such as smart TVs [30]. Through such security measures, the conventional OS of such devices is deprivileged, and security critical system operations, such as page-table management, are undertaken by a security agent running in the TEE. Trustonic introduces a TEE OS named Kinibi [31] as well as its dedicated APIs for building TAs. Samsung and Qualcomm also provide their own TEE platforms, Teegris [32] and QTEE [33], respectively. However, those platforms are proprietary and require vendors' intervention to deploy the applications in the TEE. That is, applications are developed using TEE-specific SDKs and verified and signed by the vendors before their deployment in the TEE. To access such commercial TEEs, third parties need to sign up as partners of TEE vendors and also buy licenses. For instance, Samsung defines three types of commercial licenses for registered partners depending on the supported TEE APIs and services [34].

On the other hand, ARM provides the Mbed [22] development platform, which is an open source platform that supports secure and normal OSs for high-end devices and small microcontrollers. In particular, the open source TEE called OP-TEE is hosted as a secure OS. However, despite its open source support, IoT service providers cannot fully control the TEE unless they have access to the hardware-based root of trust (e.g., a device's secret key in eFuse [35]) that is managed in the TEE.

3 Motivation and Attack Model

3.1 Motivation

We assume the conventional TEE usage model in which a TEE vendor and a TEE client are separated. In this case, the client refers to an IoT service provider, i.e., an IoT device owner, who implements IoT services and manages the software running in the REE (e.g., Linux). On the other hand, the TEE vendor manages the TEE software stacks, such as the trusted applications, secure OS, and firmware. Although the client cannot manage the TEE directly, the REE software depends on the TEE to some extent to benefit from defensive measures, such as secure boot and runtime kernel protection. The secure boot mechanism verifies the integrity and signature of the REE OS at boot time, whereas the runtime kernel protection ensures that the kernel static region is immutable at runtime. Both techniques require the intervention of the TEE, such as the hash validation of the REE OS or the emulation of critical OS operations shown by TZ-RKP [15]. This dependency results in the intervention of the TEE vendors to reflect any update of the protected REE components in the TEE-based protection (verification) mechanism. For instance, TZ-RKP ensures that the OS text region is in read-only format and cannot be patched at runtime without TEE privileges. To achieve this, (1) a new TEE service that conducts in-memory patching should be deployed or (2) a new version of the REE OS image must be redeployed and loaded by rebooting the device.

The downside of the conventional TEE model is that it increases the complexity of managing an IoT device. As previously mentioned, maintaining the software stack protected by a TEE requires the TEE vendor's intervention. 3rdParTEE attempts to address this issue by enabling IoT service providers to securely run their code with TEE privileges without introducing a new attack vector to the TEE. Additionally, the existing TEE management model can be preserved. Simultaneously, IoT service providers can perform their IoT device management operations in parallel without waiting for the TEE vendor's intervention. We expect such operations to imperatively fix critical OS bugs in memory or reliably dump and investigate REE memory without interference from untrusted OSs.

3.2 Attack Model

We assume that both the TEE vendor and the client are not malicious and that they trust one another. In addition, only the applications created by an authorized client can be loaded onto the TEE. This can be achieved through application signing and verification. As a result, malware cannot be deployed in the TEE through an ordinary TEE application deployment mechanism. However, the TEE application might contain vulnerabilities. Therefore, an attacker can compromise a TA through a maliciously crafted input. Once the TA is compromised, the attacker can further attempt to take control over higher-privileged TEE software, such as a secure OS [31]. On the other hand, TEE attacks that exploit hardware vulnerabilities, such as Rowhammer [36], Spectre [37], and VoltJockey [38], are not considered. In addition, we assume that the device is equipped with an input-output memory management unit (IOMMU) [39] and is properly configured. Therefore, direct memory access (DMA) attacks [40] also fall outside the scope of our attack model.

4 System Design

4.1 Overview

4.1.1 *Usage Model*

To ensure that only the trusted TEE clients use 3rdParTEE, we depend on the public key infrastructure (PKI), which is prevalently used to secure mobile device services, such as the digital right management (DRM). In other words, we assume that the client registers their public key to the TEE vendor. The vendor then saves the hash of the public key in the TEE. This is the sole involvement of the TEE vendor in using 3rdParTEE. To run the client-provided kernel module, the module developer first signs the hash of the module using a private key, which is a pair of registered public keys. The public key is also delivered to the device along with the signed hash. Before loading the kernel module, 3rdParTEE first verifies the integrity of the public key using the stored hash in the TEE. It then checks the integrity of the kernel module by comparing the newly calculated hash with the decrypted hash using the public key.

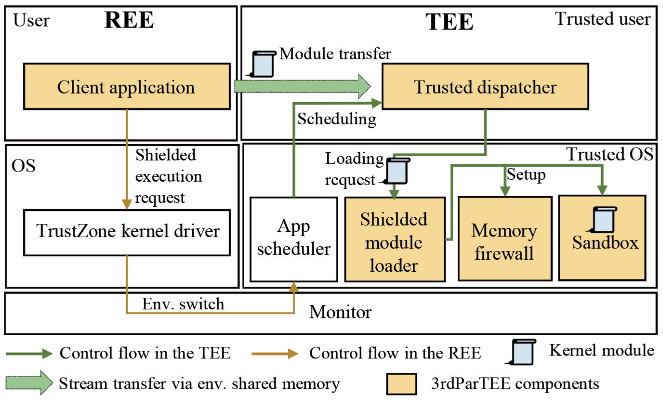

Fig. 2: Design of 3rdParTEE. In the REE, the client application transfers a kernel module to the TEE and requests a shielded execution. In the TEE, the trusted dispatcher verifies the integrity of the module and invokes a shielded module loader. The loader configures the memory firewall and the sandbox to securely execute the module and preserve the TEE security, respectively.

4.1.2 *Design*

3rdParTEE consists of two parts: the REE and the TEE. For 3rdParTEE, the REE part has three components: a loader, a client's kernel module, and a TrustZone kernel driver. The loader simply reads the entire kernel module as a file stream and writes it to the memory. Next, it communicates with the TrustZone kernel driver to send a request to the TEE part of 3rdParTEE to verify and execute the module. In the TEE part, 3rdParTEE first verifies the integrity of the kernel module, as previously mentioned. It then maps the module-loaded region to the TEE and configures some defensive facilities to ensure the security of the module as well as that of the existing TEE components. Finally, it starts running the kernel module in an isolated manner, which is referred to as a shielded kernel module. Further details regarding the design of 3rdParTEE are presented in the following sections.

4.2 REE Components

We aim to enable IoT service providers (TEE clients) to deploy their kernel modules on-demand and securely run them under TrustZone protection without hampering the security of the TEE. Additionally, 3rdParTEE attempts to minimize developers' efforts when building TAs that generally require the verification and separation of the applications' security critical logic. It also attempts to implement such applications without using specific TEE vendor-provided APIs. The REE components required to accomplish this goal are described in the following sections.

4.2.1 *Shielded Kernel Module*

Developers can create their kernel modules running as shielded kernel modules for IoT service and device maintenance purposes, thereby benefiting from 3rdParTEE. For example, a module can urgently patch the kernel text region in memory to fix a bug without replacing the entire kernel image and rebooting the device. Generally, under the condition that the text region is enforced to be immutable owing to the protection provided by the TEE-based memory protection solutions [15], [19], this patching operation may require TEE-vendor intervention to update the protected memory with TEE privileges. However, because 3rdParTEE enables running the client's module with TEE privileges, the TEE vendor's intervention is not required. In addition, the module is isolated and shielded from untrusted OSs during its execution. Therefore, memory inspection operations, such as dumping REE memory for forensic analysis, can be conducted reliably without an attacker's interference.

The development practice of the shielded kernel module is similar to that of a normal Linux LKM, which also contributes to minimizing the development effort and complexity that stem from the fact that the REE and the TEE use different APIs. In other words, the shielded kernel module is developed using normal Linux kernel symbols that allow for the shielded kernel module's tasks, such as inspecting kernel data structures. One minor difference between the general device driver and the shielded kernel module is that the shielded kernel module is self-contained and runs without interacting with the user process. Thus, the shielded kernel module does not use kernel APIs, such as copy_from_user and copy_to_user. To check the integrity of the shielded kernel module, a checksum is generated. We obtain a hash of the module based on SHA256 and sign it using the developer's private key. This signed hash is delivered to the TEE part of 3rdParTEE together with the request for executing the shielded kernel module.

4.2.2 *3rdParTEE Client Application*

The 3rdPartEE client application (CA) is a 3rdParTEE component that triggers the execution of a shielded kernel module. The CA handles three input arguments: the path of the shielded kernel module image, the entry point (function name), and the signed hash of the module. It first loads the shielded kernel module image as a file stream. The starting address of the loaded kernel module, the size of the module, the entry point, and the signed hash are packed as a message to the TEE using the TEE APIs. It then invokes an inter-processor interrupt (IPI) to turn off the other cores. By doing so, we can prevent the time-of-check-to-time-of-use (TOCTTOU) attack that abuses race conditions between multiple cores. In particular, this approach aims to reliably conduct the module's tasks, such as REE memory inspection, by preventing potentially malicious cores from hiding the attacker's footprint (e.g., the timely restoration of manipulated kernel objects before inspection). Finally, the CA opens the TrustZone kernel driver to deliver the message to 3rdParTEE on the TEE side.

4.2.3 TrustZone Kernel Driver

The packed message created by the CA is copied by the TrustZone kernel driver and sent to the TEE part of 3rdParTEE. Specifically, the TrustZone kernel driver executes the SMC instruction that switches the CPU mode to the monitor mode. The handler code running in the monitor mode verifies the request and delivers it to 3rdParTEE. Although the OS kernel is untrusted, using the TrustZone kernel driver is unavoidable because the CPU's mode switch to the TEE is conducted by executing the privileged SMC instruction. Instead of implementing the TrustZone kernel driver from scratch, we reuse the driver provided by the TEE vendor. In our PoC, we leverage the Sierraware TEE and its TrustZone kernel driver. For production devices, the TrustZone kernel driver provided by a specific TEE vendor will be utilized. As mentioned previously, because the TrustZone kernel driver resides in the REE, it can be compromised. Therefore, the request for the execution of the shielded kernel module can be ignored, i.e., denial of service (DoS) attack. In addition, a fabricated value can be returned instead of the actual result of the execution of the shielded kernel module. For instance, the attacker can attempt to deceive the remote administrator of an IoT service, i.e., the remote attester, by always returning the normal state of the device instead of the actual inspection result of the shielded kernel module. However, such attacks can be easily hampered by signing the result with a device secret key, which can only be accessed in the TEE, and thus, by 3rdParTEE as well.

4.3 TEE Components

The request for the execution of the shielded kernel module is handled by the TEE components of 3rdParTEE. The main roles of the TEE components include shielded kernel module initialization, memory firewall configuration, sandbox activation, and module dispatching.

4.3.1 Trusted Service Dispatcher

The trusted service dispatcher is a TA, which is a counterpart of the 3rdParTEE CA. It receives and handles the request, i.e., the packed message from the CA, for the execution of the shielded kernel module. The dispatcher first parses the message to obtain the kernel module's address and size. It then allocates memory in the sandboxed TEE region and copies the third-party kernel module image to that region based on the size information of the module. The sandbox is introduced to protect the TEE from the third-party kernel module (Section 4.3.4). The hash of the copied module is verified using a signed hash from the CA. After verifying the hash, the dispatcher invokes the shielded module loader (Section 4.3.2) to initialize and execute the shielded kernel module. The shielded kernel module's address in the sandbox and entry point is delivered as an argument for the shield module loader.

4.3.2 Shield Module Loader

The shielded module loader, which is a modified version of the ELF loader, performs general operations, such as parsing the ELF header, linking kernel symbols, and loading each memory segment, to appropriately execute the shielded kernel module. Note that the kernel symbol information is needed because the shielded kernel module is compiled using the symbols, and it refers to them during execution. The symbol information is transferred to the TEE as part of a secure boot chain. Therefore, its integrity is preserved. However, simply loading the kernel symbol information in the TEE is not sufficient to run the shielded kernel module. This is because the shielded kernel module may invoke the REE OS kernel's APIs whose addresses are yet to be mapped in the TEE page tables.

Therefore, memory mapping to OS APIs should be created afresh in the TEE. Instead of traversing and copying every entry in the OS page tables to the TEE page tables, we revisit the ARM virtual memory system features, specifically, the TTBRs and the TTBCR. The key idea is to use one of the two TTBRs to reference the base address of the REE OS page table. We configure TTBR1 to map the REE OS region to the address space of the TEE. As a result, the shielded kernel module mapped in the TEE can seamlessly access the OS APIs without redundant memory mapping procedures that may incur complex page table management and performance overhead. On the other hand, TTBR0 is utilized for the virtual memory mapping of the TEE as its original usage.

Figure 3 depicts the setting for memory translation when this approach is applied. In the REE, TTBR0 is used to access the entire virtual address space for the REE. Each time the monitor mode is entered for the execution of the shielded kernel module, it saves the value of TTBR0 of the REE. After that, TTBR1 of the TEE is newly configured with the saved value, which is the copy of TTBR0 for the REE. Additionally, the virtual memory setting for the TEE is reconfigured using the TTBCR. Specifically, the N bits of the TTBCR in the TEE, which are used to configure how TTBRs are used, are set to 001_b, i.e., 001 in binary. This configures the virtual memory addresses that range above 0x80000000 to be translated by referencing TTBR1 and those below using TTBR0. As mentioned previously, because TTBR1 currently contains the value of TTBR0 of the REE, the TEE virtual addresses above 0x80000000 are mapped to the REE. To this end, we also recompile the TEE software using a new configuration that enforces the TEE to explicitly be mapped in the virtual addresses that range from 0x00000000 to 0x7fffffff.

This approach has several advantages. First, we can minimize the performance overhead because the need to traverse and manage the page table is exempted. Further discussion on the performance benefits is presented in Section 6.2.2. Additionally, it allows the shielded kernel module that obtains TEE privileges for invoking OS APIs with their original symbol addresses. Finally, switching the TTBR on every entry to the TEE enables a fresh view of the current kernel context.

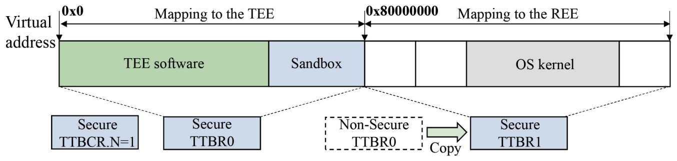

Fig. 3: The non-secure TTBR0 for the OS is copied to the secure TTBR1 to map the OS region. The TEE and sandbox are mapped using the secure TTBR0. For the use of two TTBRs, TTBCR.N is set.

4.3.3 Memory Firewall

The attacker cannot directly influence module execution because the shielded kernel module runs in the TEE. For example, the module text as well as the execution context in the stack and heap cannot be manipulated during module execution. However, the attacker can still affect the result of the execution of the shielded kernel module by manipulating the status of target inspection objects in a timely manner. For instance, a malicious core in the non-secure state can conceal the attack footprint in the kernel immediately before the shielded kernel module runs in the secure state (TEE). To address this issue, as shown in Section 4.2.2, we first temporarily stop any core except the one that runs the shielded kernel module. However, because the IPI for stopping other cores depends on the execution of the kernel API, which can be ignored by the attacker, this approach does not reliably prevent the threat.

Therefore, the reliable measurement of the shielded kernel module should not solely depend on this approach, which simply freezes other cores. Therefore, we utilize the TZASC to prevent malicious cores from disturbing the result of the shielded kernel module. As described in Section 2.1, the TZASC performs access control in the physical memory regions based on the CPU's security state. For example, the access permissions of the physical memory region allocated to the TEE are accessible only when the CPU is in the secure state. On the other hand, the REE region can be set to be accessible regardless of CPU states to allow the software running in the TEE to access the REE, which enables the creation of a shared memory for a communication channel between the environments or emulates security critical operations of the OS in the TEE [15].

In our design of 3rdParTEE, we dynamically reconfigure the TZASC before executing the shielded kernel module such that the entire physical region allocated to the REE is not accessible by cores in the non-secure state. By doing so, we can ensure that the shielded kernel module and kernel APIs are not accessible by the malicious cores running in the non-secure state, even if the attacker blocks the IPI requests for stopping them. Note that because the core running the shielded kernel driver is in a secure state, it can still access the entire physical memory, and thus, seamlessly execute the module. This dynamic configuration of the TZASC is straightforward because the physical memory regions for the REE and the TEE are already defined and configured at boot time. Thus, redefining the regions is not required. We simply reset the REGION_ID_ACCESS_<n> register of the TZASC to block the read and write accesses to the REE region from the cores in a non-secure state.

4.3.4 Sandbox for Shielded Execution

As discussed in Section 3.2, we assume that the client (the IoT service provider) is not malicious and is trusted by the TEE vendor. Therefore, the shielded kernel module developed by the client does not explicitly perform malicious operations. Nevertheless, the shielded driver's vulnerability or unintended access to the TEE owing to a programming error should be considered because these aspects can be abused by the attacker to compromise the TEE. For instance, the attacker might exploit the shielded kernel module to learn the internal workings of the TEE. This cannot be generally fulfilled through static analysis, because the TEE OS and firmware images are generally provided as encrypted binaries. However, because they are decrypted in the TEE at runtime, the attacker can abuse the shielded kernel module privileges to infer the fruitful information of the TEE.

```
//...(Omitted)...
 LOOP:
 mov x1,#TEEADDR          //Get the TEE addr.
 msr DBGWVR0_EL1,x1       //Set the watchpoint
 mov x2,#TEEADDR          //Get the TEE addr.
 cmp x1,x2                //Validate the set value
 b.ne LOOP                //If invalid, loop back
// ...(Omitted)...
```

Listing 1: Validation of watchpoint setting. Note that the size of watchpoint monitoring is configured in the watchpoint control register (DBGWCR), which is omitted in the code snippet.

To address this issue, 3rdParTEE adopts a watchpoint-based memory isolation technique [28] to build a sandbox for the shielded kernel module. We first reserve part of the TEE memory, which is 1 GB in our implementation, as the sandbox region. Next, in the shielded kernel module initialization phase, this reserved region is leveraged for module allocation. Before executing the module in the sandbox, we configure the watchpoint to monitor any access outside the sandbox where other TEE software resides. In other words, watchpoint-based read and write access monitoring is activated so that escaping to the TEE generates an exception. Additionally, the memory allocation for the stack and the heap is conducted using the memory in the sandbox. Once the module completes its execution, the watchpoint is reconfigured to make the TEE region accessible again. Because the watchpoint configuration instructions are accessible in the sandbox, the attacker might try to abuse them to manipulate the configuration. To prevent this, we ensure that the watchpoint configuration instructions are adhered to through verification logic that checks the correctness of the configuration. As shown in Listing 1, this verification is straightforward because the watchpoint should always be configured using constant values, i.e., the non-sandboxed region's start address and size. Note that despite the watchpoint-based protection, the outside of the sandbox region is still executable. Thus, the attacker might attempt to divert the control flow to the non-sandboxed region to leverage useful TEE libraries and achieve a successful attack. However, we expect this to be difficult because the internal layout of the TEE cannot be dynamically or statically analyzed. In particular, the encrypted TEE software prevents static analysis, as mentioned previously. In addition, the watchpoint-based sandbox hampers runtime information leakage in the TEE layout. As a result, the attacker cannot pinpoint the location of useful code chunks in the TEE.

5 Implementation

We implemented our PoC using an Arndale board equipped with Cortex-A15 dual-core processors and 2 GB RAM. Linux v4.4 and SierraTEE [41] were used as the building blocks of the REE and the TEE, respectively. Note that SierraTEE provides the TEE software components, such as secure OS, as well as the TrustZone kernel driver and TEE libraries that help the REE applications interact with the TEE. We attempt to maximize the use of such TEE vendor-provided software stacks to implement the REE components of 3rdParTEE instead of implementing them from scratch. This is because, as discussed in Section 3.2, we assume that the TEE vendor and client trust one another, and the vendor is willing to open part of the TEE to allow the client to maintain their IoT services with safely escalated privileges. Additionally, reusing the existing components contributes to minimizing the unexpected vulnerabilities introduced by adding new software components to the TEE.

In this regard, the change in the REE is minimal. For the secure transmission of kernel symbols to the shielded kernel module loader in the TEE, we added a TrustZone driver invocation during the booting of the device. More specifically, we lead the init process to create a TEE message that contains the kernel symbol information, such as the $symsearch$ data structure, which consists of *__start_-__ksymtab* and *__stop___ksymtab*. This information is then linked to the shielded kernel module through the module loader in the TEE. The 3rdParTEE CA is implemented using SierraTEE libraries for opening and closing a session with the trusted dispatcher, which packs a message and sends it to the TEE.

On the other hand, the trusted dispatcher is added, which is a new TA for communicating with the 3rdParTEE CA, to the TEE. As described in Section 4.3.1, it receives and parses the message from the CA, and it maps the module image in the TEE sandbox using SierraTEE OS APIs. The shielded kernel module loader is implemented as one of the TEE OS services. Therefore, we create a new system call for invoking the loader. The loader is implemented by modifying the Linux ELF loader. In addition to loading the shielded kernel module, it sets up the debug watchpoint to restrict any illegitimate access to the TEE by escaping from the sandbox. Because the starting address of watchpoint monitoring should be aligned with the size of monitoring [29], we simply divide the virtual memory for the TEE into two regions with the same size, and we assign the higher half to the sandbox. Specifically, virtual addresses ranging from 0x0 to 0x3fffffff and from 0x40000000 to 0x7fffffff are assigned to the TEE and the sandbox, respectively (note that virtual memory from 0x80000000 is assigned to link the REE kernel symbols). The TZASC must also be set up by the loader for the memory firewall before executing the shielded kernel module. Unfortunately, our low-end development board does not fully provide all the TrustZone components. Therefore, we omit this configuration. Note that this is not the case for commercial devices that implement TrustZone-based TEE because the TZASC is an essential component for memory isolation. Thus, applying 3rdParTEE to commercial devices is still feasible. Finally, similar to other architectures, ARM separates the page permissions between the user and the kernel. Therefore, the mapped REE kernel region for linking the kernel APIs to the shielded kernel module is intrinsically only accessible (executable) through kernel privileges. As a result, the shielded kernel module is scheduled to run with a secure kernel privilege.

6 Evaluation

We first perform a security analysis on 3rdParTEE in terms of preserving the TEE security as well as protecting the shielded kernel module. The performance of 3rdParTEE is also evaluated by running three example shielded kernel modules, which check the kernel integrity, patch the system call table, and traverse kernel data structures.

6.1 Security Analysis

The primary goal of 3rdParTEE is to ensure the secure execution of third-party kernel modules. Thus, we first discuss how several attack vectors for shielded execution are accomplished. In addition, because the shielded kernel module runs with escalated privileges, we show how the security of existing TEE components is protected from the exploitation of vulnerabilities in the shielded kernel module.

6.1.1 Attack Against Shielded Module Execution

In this section, we assume that the attacker's goal is to hinder the IoT device owner's management of the device. Examples of such management operations include the urgent patching of the kernel memory and measuring the kernel object integrity. Such operations are conducted through the shielded kernel module. Therefore, compromising the shielded kernel module is a viable approach for achieving the attacker's goal. On the other hand, the attacker can control the factors that affect the results of module execution without directly compromising the shielded kernel module. For example, the system status can be transiently restored before being verified by a shielded kernel module.

Compromising the shielded kernel module execution. The binary in the device storage or loaded module image in memory can be manipulated. This can be prevented by comparing the hash of the loaded kernel module with the precomputed one that is signed using the shielded module provider's key. By considering the time-of-check-to-time-of-use (TOCTTOU) attack that attempts to manipulate the image between verification and execution, 3rdParTEE copies and verifies it in the TEE.

Because the shielded kernel module is linked to the kernel APIs, compromising the APIs can result in the shielded kernel module's malfunction. However, this also requires the attacker to bypass additional security facilities, such as real-time kernel protection, which is widely deployed in commercial devices and is also adopted in our work. The kernel text and data are essentially enforced to be read-only through the RKP. Therefore, manipulating the kernel

APIs is not achievable. Diverting the control flow (e.g., the kernel ROP) can possibly bypass the RKP [15]. To address this attack, numerous approaches have been proposed to ensure control flow integrity [42], [43], [44], [45], [46], [47], [48], [49], [50]. We expect such defenses to complement the operations of 3rdParTEE.

The shielded kernel module may contain vulnerabilities that can be exploited by an attacker. Considering the fact that the crafted input exploits vulnerabilities, the kernel module can be developed as a self-contained task that fulfills the required operations without taking arguments. In contrast, enabling the argument-taking feature of the shielded kernel module may increase the reusability of the deployed module. For example, the urgent patching of the kernel text might require updating different memory regions depending on the vulnerability. To prevent the attacker from abusing this aspect to bypass the kernel integrity protection approaches [51], the invocation of the module should be restricted to the module owner or remote attester. To achieve this, signing and verifying the arguments using a secret key can be a viable option, similar to verifying the shielded kernel module.

Interfering communication channel. 3rdParTEE aims to provide an approach for securely running the IoT device owner-provided kernel module that performs device management operations. For instance, remote attestation can be performed, with the dumping of device memory as the measurement result. Even if the measurement is securely created through the shielded kernel module, additional operations, such as loading the module and sending the measurement results, are mediated by the untrusted REE OS. Thus, the attacker may attempt to compromise such operations in the REE. For instance, the measurement result can be manipulated before being sent to the remote administrator to conceal the attack footprint. However, preventing this attack is straightforward. Because the measurement is generated in the TEE, it can be safely signed using a device secret key, which is only accessible in the TEE.

Manipulating system status. If the aim of the shielded kernel module is to measure the system status, the attacker might transiently restore the system status to hide any footprint before invocation of the shielded kernel module. For example, the privilege of a malicious process that was escalated to the root can be deprivileged before the shielded kernel module detects it. In our current prototype of 3rdParTEE, the module is synchronously invoked by the client application that runs in the untrusted REE. Thus, if the attacker already compromises part of the control flow for entering the TEE from the client application, the status can be restored in a timely manner. Therefore, the measurement result will not reflect actual system states with the presence of a malicious process. This problem can be resolved by hardening the mechanism of the shielded kernel module invocation. For instance, once the shielded kernel module is successfully loaded in the TEE, it can be scheduled to run periodically at random intervals. The attacker might abuse the multi-core environment to manipulate the system status by winning the race condition between the cores. This attack is prevented by activating the memory firewall, which blocks any memory access from malicious cores.

6.1.2 Undermining the TEE Security

Because the shielded kernel module runs with the TEE privilege, it can result in breaking the TEE security once the module is compromised. As discussed in Section 3.2, the shielded kernel module possibly contains a vulnerability that can be exploited by the attacker. If the write-what-where vulnerability of the shielded module is exploited, the attacker may attempt to tamper with the TEE as well as the shielded kernel module. Additionally, the attacker may also attempt to execute any TEE functions by hijacking the control flow. Fortunately, the attacker's ability is limited to compromising the module in the presence of 3rdParTEE. In other words, owing to the sandboxing approach, the attacker cannot read or write the TEE region. Any attempt to access the TEE from the compromised module generates a watchpoint exception that is trapped by the predefined exception handler. The remaining option for an attacker is disabling the watchpoint or remapping the exception handler. These operations require the execution of privileged instructions that configure the watchpoint or the vector base address register (VBAR). Because we assume the deployment of RKP [15] that sanitizes the OS to remove all the privileged instructions from the dynamically loaded kernel modules and kernel text, the attacker cannot find desirable instructions for a successful attack. Because the TEE region is still executable, the attacker diverts the control flow to execute such instructions in the TEE. We expect that this is not readily achievable because commercial TEE software is not open to the public, and thus, it cannot be statically analyzed. In addition, the watchpoint-based sandbox naturally ensures that the TEE region is execute-only [52], [53], which prevents the attacker from dynamically finding useful instructions at runtime.

Finally, for performant introspection from the shielded kernel module, we reuse the OS page table (Section 4.3.2). This can be an attack vector that compromises the TEE. In particular, the attacker manipulates the page table to map malicious kernel APIs instead of normal ones. The shielded module may then execute malicious kernel APIs that disable the protection enforced by 3rdParTEE. Although it is a crucial attack vector, it is also essentially prevented by the presence of the RKP because the page table is a critical kernel object that is strictly protected by the RKP. In other words, any update of the page table is verified and emulated by the RKP. Therefore, even the compromised OS cannot manipulate it.

6.2 Performance

6.2.1 Inter world communication

We first measure the round-trip time between the CA and the shielded kernel module. Because this aims to evaluate the pure latency required for communication, the shielded kernel module does nothing (executes a dummy function). Additionally, the module is already loaded and mapped in the TEE. As shown in Table 1, the round-trip time for calling a shielded kernel module takes 258.72 μs. In our analysis, 201.71 μs and 57.01 μs were taken in the REE and the TEE, respectively. Specifically, the TEE API execution that uses the ioctl system call to send a request message to the TEE is the major reason behind the latency in the REE. Note

TABLE 1: Latency of inter-world communication. The REE latency includes the time for the ioctl system call and its handling. The latency for the TEE is incurred by switching CPU modes and invoking the shielded module with no operation.

REE	TEE	Overall
201.71μs (77.9%)	57.01μs (22.1%)	258.72μs

that the message contains the ID of the preloaded shielded kernel module, and the time required to create the message is not included in the measurement. On the other hand, the latency for the TEE (57.01 μs) is incurred from switching environments and dispatching a shielded kernel module that simply runs a dummy function.

6.2.2 Kernel Memory Mapping and Introspection

.

Improving the performance by reusing the OS page table, as illustrated in Section 4.3.2, is also evaluated. This is conducted by accessing the kernel object from the shielded kernel module. In particular, we compare the latencies for accessing the kernel region using two different approaches: (1) reusing the page table (2) and creating the mapping to the kernel in the TEE page table. We ran five tests with access sizes ranging from 4 bytes to 20 bytes. Additionally, the granularity of the memory access is 4 bytes. We managed to ensure that each 4 bytes reside in different kernel pages. Therefore, the access to each 4 bytes requires the creation of new page table entries. We ran each case 100 times, and the latency was measured using the performance monitor cycle count register (PMCCNTR). Figure 3 shows the results. Reusing the OS page table took 87 cycles regardless of the memory access size. However, the latency for the approach with page table update linearly increases depending on the accessed memory size, from 324 cycles with 4 bytes to 1788 cycles with 20 bytes. This is because creating page table entries requires more operations, such as traversing the kernel page tables for virtual to physical memory translation of the accessed memory, creating new page table entries, and flushing the TLB cache after updating the page tables. In contrast, our approach solely requires reconfiguring the TTBCR and the TTBR1 to reuse the OS page table. It is also worth reiterating that the experiment only accesses the maximum 20 bytes of kernel text. Hence, we expect that if the sparse memory region is accessed, for example, by traversing the dynamic kernel objects with various sizes, the timing gap between the two approaches will significantly increase as a result of the overhead of creating new page mappings.

6.2.3 Shielded Third-party Module Execution

We evaluated the performance of shielded kernel modules that are protected by 3rdParTEE. To this end, three modules were created, and their performance was compared to that of normal kernel modules that conduct same tasks. In particular, each module performs kernel hash calculation, system call table updating, and kernel object traversing.

Kernel integrity check. Hash measurement is a fundamental operation for the remote management of IoT devices. For instance, the integrity of running tasks or important system files can be checked using their hash. We created a kernel module that generates the hash of the kernel text using SHA256. The module obtains the start address of the kernel and its size using the *_etext* and *_stext* kernel symbols. It is then loaded in the TEE and runs as a shielded kernel module. Additionally, we loaded the same module in the REE as the normal kernel driver to evaluate the overhead resulting from the adoption of 3rdParTEE. As shown in Figure 4, the overhead of running the shielded kernel module was 1.8%. This overhead is mainly incurred by the additional operations of 3rdParTEE, such as copying the module to the TEE, validating the hash of the module, and switching between the environments for execution. According to our analysis (Table 2, copying and validating the module with a size of 109 KB was the major reason behind this overhead, which took 4751 μs and 149 μs, respectively. On the other hand, the primary task of the module, which is verifying the kernel integrity using SHA256, took approximately 68 ms in execution time.

System call table update. A shielded kernel module that updates a system call table entry is created, and its performance is evaluated. The system call table is a static object whose integrity is protected by kernel integrity monitors, such as the RKP. Thus, updating such objects is not allowed using software with lower privileges than the RKP. However, because our aim is to enable the IoT device owner to urgently manage the device before the TEE vendor takes action, the privilege of the module should be escalated. This is essentially achieved by 3rdParTEE that runs the module with TEE privilege. As mentioned previously, this escalation requires additional operations, such as loading the module to the TEE and validating its integrity. We observed an overhead of 2.7% compared to that of the normal kernel driver. In this experiment, the size of the module was 51 KB, and it took 2242 μs to prepare the loading and validation of the shielded execution.

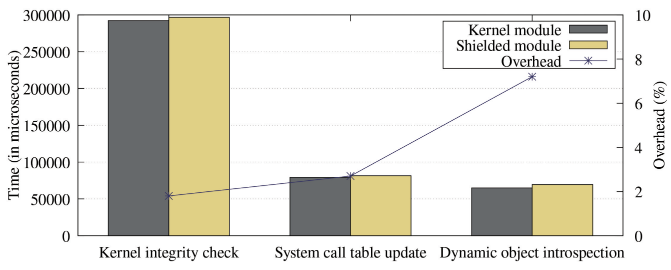

Fig. 4: Shielded module performance compared to that of kernel modules. A maximum overhead of 7.2% was observed through dynamic object introspection.

Dynamic object inspection. An attacker who compromises the OS can manipulate dynamic kernel objects to hide the attack footprint. For instance, the pointer for reading the function of the virtual file system (VFS), which is placed in a kernel object, can be hooked to redirect it to a malicious function that hides specific information, such as the network connection to a malicious server. Thus, traversing the kernel objects is a crucial feature for managing device security because it enables checking the validity of the expected data in the object. To measure the performance of such a task, we

TABLE 2: Time breakdown of three shielded modules and ratio to overall time.

	Module size	Loading	Validation	Execution	Overall
Kernel integrity check	109 KB	4751μs (1.6%)	149μs (0.2%)	291274μs (98.2%)	296615μs
System call table hooking	51 KB	1562μs (1.9%)	680μs (0.9%)	79164μs (97.2%)	81406μs
Kernel object traversing	143 KB	3483μs (5.0%)	1893μs (2.8%)	64103μs (92.2%)	69479μs

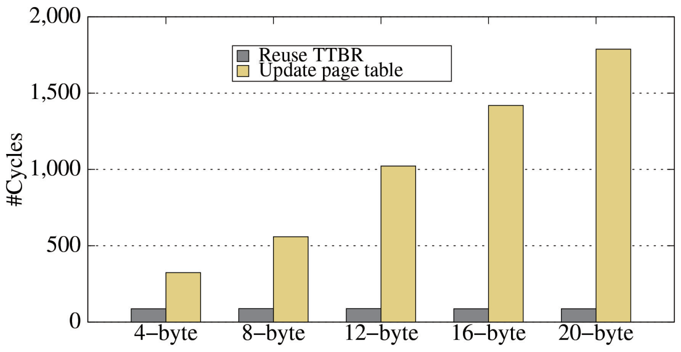

Fig. 5: Owing to the exemption of traversing and configuring page tables, and flushing the TLB, reusing the TTBR outperforms updating the page table, specifically when the number of accessed pages increases (accesses with 4-byte granularity in different pages).

created a shielded kernel module that traverses the *proc_-dir_entry* data structures, which are linked to each other and enables the searching of files in the */proc* directory. Generally, system information, such as networks, file systems, and devices can be retrieved by referring to the files in the */proc* directory. We specifically find the object for the */proc/net/tcp* file and check its function pointer in the VFS read function to determine whether its value is within the valid range of the kernel text. Compared to the normal kernel driver that performs the same task, the shielded module introduces an overhead of 7.2%. Similar to the other two experiments, the overhead was incurred owing to the additional operations for protecting the module. As shown in Figure 4, this test introduced the highest overhead among the three tests. This is because the proportion of additional time required for shielding the module is significantly higher than those of other two tests. In particular, the size of the shielded kernel module (143 KB) is the largest compared to other tests, but the execution time is the shortest (64103μs). This results in 3rdParTEE's additional operations occupying a larger portion (7.8%) in the module's entire lifetime, and thus, it introduces most of the overhead among the three tests (Table 2).

7 Related Work

In this section, we introduce a line of work relevant to TrustZone technology, including TrustZone application, attack and defense, and opening the TrustZone for 3rd-party usage.

7.1 Application of TrustZone

In academia, TrustZone-based TEE has been widely adopted to build trustworthy services. For example, TrustOTP [13] and fTPM [54] build a software-based one-time password (OTP) and a trusted platform module (TPM) in the TEE, respectively. TZ-RKP [15] and Sprobes [27] use TrustZone to isolate kernel-integrity monitors from the untrusted kernel. Ninja [55] utilizes TrustZone for stealthy malware analysis. Finally, TrustZone is leveraged for the reliable control of peripherals [56], [57], securing the IO [14], [58], [59], and the reliable acquisition of memory [60]. Similarly, 3rdParTEE also benefits from TrustZone in shielding the IoT device owner-provided kernel module.

On the other hand, as an industry solution, Samsung KNOX [19] implements TZ-RKP as part of its platform stack to protect OS kernel integrity on 32-bit ARM architecture-based mobile devices. In addition to protecting the kernel static regions, the LKMs are verified to ensure that the TZ-RKP is not bypassed. In other words, as illustrated in Section 2.3, security-critical instructions are removed from the kernel text to ensure the immutability of static regions, and thus they should never be reintroduced in any kernel region. Consequently, TZ-RKP verifies and sanitizes the LKMs to ensure that any removed critical instructions do not exist in the LKM. Note that this verification of the kernel module aims to ensure the kernel integrity, not to protect the module itself. We expect 3rdParTEE and TZ-RKP to be complimentary. The verification mechanism of TZ-RKP can be employed to minimize the vulnerabilities in the modules that are shielded by 3rdParTEE.

7.2 TrustZone attack and defense

The security of TrustZone technology has been explored. Researchers have shown that cache side channels can be exploited to compromise the TrustZone-based TEE [61], [62], [63], [64], [65]. Specifically, CITM illustrates how the security of isolated execution environments can be broken when they leverage memory that is shared with untrusted-OS instead of being built in the TrustZone-protected memory. Because the shielded kernel module is deployed and run in the TrustZone-protected memory, 3rdParTEE is immune to the CITM attack. CacheKit [66] utilizes the cache incoherence between the REE and TEE to hide malware from the introspection conducted in the TEE. VoltJockey [38] and CLKScrew [67] showed that fault injection attacks can exfiltrate a secret from the TEE. Finally, the Boomerang attack [51] and horizontal privilege escalation (HPE) [68] are types of confused deputy attacks that abuse TrustZone to attack REE OS and CAs. On the other hand, previous work has also attempted to harden the TEE. SeCReT [69] proposed a method to secure the communication channel between the REE and TEE. Pager [70] enables the TEE OS and TAs to run in TrustZone-protected SRAM. To secure the TEE from the cache side channel attacks, secTEE [71] separates caches between different TAs and cleans up the caches when the CPU switches to the REE. Finally, PARTEMU [72] provides an emulation environment for conducting a dynamic analysis of TAs and determining their vulnerabilities.

7.3 Openness of TrustZone

Approaches for improving third-party accessibility to the TEE have been explored. PrivateZone [24], TFence [73], and OSP [74] utilize hardware-assisted virtualization available on ARM architecture to shield third-party-provided security critical applications in the REE. TrustICE [26] and Sanctuary [75] also protect applications running in the REE, but they utilize the TrustZone component, TZASC, for their isolation. Ginseng [76] protects secret data by ensuring that they are placed only in registers, which is achieved through the compiler technique and privileged services running in the TEE. Contrary to the aforementioned systems, TrustShadow [17] and on-board credentials (ObC) [25] directly run the critical applications in the TEE by employing a lightweight runtime system and emulation platform in the TEE, respectively. Trusted language runtime (TLR) [77] ports the .NET framework in the TEE to ensure the feasibility of mobile application development.

Furthermore, several open TEE platforms have been released, contributing to the prosperity of TEE-relevant research. OP-TEE [78] is an open-source TEE platform that can be ported to various development boards. SierraTEE [41], wherein 3rdParTEE is implemented, supports not only the commercial version but also the publicly available version of TEE for developers. Open-TEE [79] is a software-based virtual TEE environment that aims to improve efficiency in developing and testing trusted applications. Those open TEE platforms are built to satisfy the de-facto standard of TEE, which is specified by GlobalPlatform (GP) [80]. Therefore, general TEE functionalities defined by GP specifications (e.g., isolating applications, creating secure storage) are implemented in the open TEE platforms. However, the core technique proposed in 3rdParTEE, which is shielding a third-party kernel module by leveraging TEE, has never been introduced by such open platforms and can be adopted by them to improve the accessibility of the TEE. We further discuss the adoptability and scalability of 3rdParTEE in Section 8.

8 Discussion and Future Work

Scalability. The PoC of 3rdParTEE is implemented on SierraTEE. However, because the core techniques of 3rdParTEE leverage general hardware features available on ARM architecture, we expect them to be readily employed by other TEE platforms as well. For instance, the TZASC leveraged for building the memory firewall (Section 4.3.3) is generally available on devices that properly implement the TrustZone-based TEEs. The debug watchpoint for the sandbox implementation (Section 4.3.4) is also the general hardware feature defined in 64-bit as well as 32-bit ARM architectures. In the trusted service dispatcher (Section 4.3.1), one of the TTBRs–the TTBR1 in a secure state–is used for linking kernel symbols. Although the same set of TTBRs is available regardless of whether the architecture is 32-bit or 64-bit, both TTBR0 and TTBR1 are generally leveraged to map the user and kernel spaces, respectively, on 64-bit architecture. Therefore, additional engineering effort is expected to port 3rdParTEE to the TEE platforms with 64-bit ARM architecture. For example, we can reorganize the virtual memory layout of the TEE so that both the user and kernel spaces are mapped by using only the TTBR0. Finally, as long as the mandatory hardware features are available, minimal effort for software migration is expected because contemporary TEE platforms follow the TEE standard specifications. Improving the scalability of 3rdParTEE for various TEE platforms has been set aside for our future work.

Hardening the TEE. 3rdParTEE provides a way to shield a kernel module by hosting it in the TEE. Considering the vulnerability of the module and its exploitation, watchpoint-based memory sandboxing is also proposed. As illustrated in Section 4.3.4, this approach only enforces the outside of the sandbox to be non-readable and non-writable; unfortunately, it is still executable. Hence, preserving the effectiveness of sandboxing requires that the layout of the TEE platform not be exposed. By doing so, we can prevent the attacker who hijacks the control flow from abusing the useful instructions in the TEE. Satisfying this requirement is feasible for the closed TEE platforms as long as their internal layout is not revealed by accident [81]. However, open TEE platforms that publicly open their source code might be vulnerable to this attack. To address this, conventional defensive measures such as address space layout randomization (ASLR) can be applied to the TEE. Temporarily unmapping the TEE region or making it inexecutable by configuring page table entries and hardware features [82] before the execution of the shielded kernel module can be adopted as well. We will further explore ways to harden the TEE so that it is more immune to any attack that may abuse 3rdParTEE.

9 Conclusion

We designed and implemented 3rdParTEE to shield IoT service providers' kernel modules by benefiting from the TEE but minimizing its impact on the TEE security. For the reliable and secure execution of the shielded module, 3rdParTEE leverages the TZASC, which controls the memory region's access permissions. To prevent the shielded module from opening a new attack surface on the TEE, a debug watchpoint-based sandbox was applied to the shielded module execution. Owing to additional operations for enabling such protection facilities, the shielded module introduced more overhead compared to the conventional kernel module execution. However, we observed only negligible and moderate overhead, with a maximum overhead of 7% for traversing dynamic kernel objects. We expect the design of 3rdParTEE to inspire further research on security measures for increasing the safety of IoT devices.

Acknowledgments

The authors would like to thank all the anonymous reviewers for their valuable insights and feedback that helped us to improve our paper. This work was supported by the Office of Naval Research (ONR) through Award N00014-18-1-2661. This work was also supported by Institute for Information & communications Technology Promotion (IITP) Grant funded by the Korea Government (MSIT) (No. IITP-2019-0-01343, IITP-2019-0-01570, and IITP-2020-0-01840), National Research Foundation (NRF) Grant funded by Korea Government (MSIT) (No. NRF-2020R1F1A1058305) and research fund of Chungnam National University.

References

[1] (2021) Internet of things. [Online]. Available: https://www.ericsson.com/en/internet-of-things

[2] (2020) 50 sensor applications for a smarter world. [Online]. Available: https://www.libelium.com/libeliumworld/top-50-iot-sensor-applications-ranking/

[3] (2014) Increasing vending profitability with more intelligent machines. [Online]. Available: https://www.intel.co.kr/content/www/kr/ko/intelligent-systems/retail/ref-design-for-intelligent-vending-product-brief.html

[4] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "Sok: Security evaluation of home-based iot deployments," in *2019 IEEE symposium on security and privacy (sp)*. IEEE, 2019, pp. 1362–1380.

[5] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} security symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[6] O. Çetin, C. Ganán, L. Altena, T. Kasama, D. Inoue, K. Tamiya, Y. Tie, K. Yoshioka, and M. van Eeten, "Cleaning up the internet of evil things: Real-world evidence on isp and consumer efforts to remove mirai." in *NDSS*, 2019.

[7] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin, "Measurement and analysis of hajime, a peer-to-peer iot botnet," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[8] S. Soltan, P. Mittal, and H. V. Poor, "Blackiot: Iot botnet of high wattage devices can disrupt the power grid," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 15–32.

[9] (2018) "hide and seek" becomes first iot botnet capable of surviving device reboots. [Online]. Available: https://www.bleepingcomputer.com/news/security/hide-and-seek-becomes-first-iot-botnet-capable-of-surviving-device-reboots/

[10] (2018) The 7 craziest iot device hacks. [Online]. Available: https://securityboulevard.com/2018/05/the-7-craziest-iot-device-hacks/

[11] (2021) "setting new standards for cloud computing. [Online]. Available: https://www.arm.com/solutions/infrastructure/cloud-computing

[12] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing arm trustzone," in *In Proc. of the 26th USENIX Security Symposium*, 2017.

[13] 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.

[14] 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*, 2015, pp. 75–88.

[15] 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.

[16] F. Abdi, C.-Y. Chen, M. Hasan, S. Liu, S. Mohan, and M. Caccamo, "Preserving physical safety under cyber attacks," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6285–6300, 2018.

[17] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 488–501.

[18] (2019) Guard your data with the qualcomm snapdragon mobile platform. [Online]. Available: https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf

[19] (2018, May) Knox and arm trustzone. [Online]. Available: https://kp-cdn.samsungknox.com/c887e0a066f5598a1e9ecea2d68edbe1.pdf

[20] (2018) Your smartphone has a special security chip. here¡¯s how it works. [Online]. Available: https://www.howtogeek.com/387934/your-smartphone-has-a-special-security-chip.-heres-how-it-works/

[21] (2021) Secure platform. [Online]. Available: http://www.trustonic.com/secure-platform/

[22] (2020) Arm mbed linux os. [Online]. Available: https://github.com/PelionIoT/mbl-docs/blob/v0.10/Docs/introduction/introduction.md

[23] (2021) Mocana. [Online]. Available: https://www.mocana.com/

[24] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "Privatezone: Providing a private execution environment using arm trustzone," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 5, pp. 797–810, 2016.

[25] 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.

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

[27] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *Proceedings of the Third Workshop on Mobile Security Technologies (MoST)*, 2014.

[28] J. Jang and B. B. Kang, "In-process memory isolation using hardware watchpoint," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[29] (2018, May) Arm architecture reference manual armv8, for armv8-a architecture profile. [Online]. Available: https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[30] (2016) Samsung smart tv security solution gaia v1.0. [Online]. Available: https://www.commoncriteriaportal.org/files/epfiles/%5BKECS-CR-16-08%5D%20Samsung%20Smart%20TV%20Security%20Solution%20GAIA%20V1.0%20Certification%20Report.pdf

[31] (2020) Secure iot development with kinibi-m. [Online]. Available: https://www.trustonic.com/technical-articles/kinibi-m/

[32] (2020) Samsung teegris. [Online]. Available: https://developer.samsung.com/teegris/overview.html

[33] (2021) Qualcomm® trusted execution environment (tee) v5.8 on qualcomm® snapdragon™ 865 security target lite. [Online]. Available: https://www.tuv-nederland.nl/assets/files/cerfiticaten/2021/08/nscib-cc-0244671-stlite.pdf

[34] (2020) Knox licenses. [Online]. Available: https://docs.samsungknox.com/dev/common/knox-licenses.htm

[35] (2021) Using encryption and authentication to secure an ultrascale/ultrascale+ fpga bitstream. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1267-encryp-efuse-program.pdf

[36] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[37] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[38] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 195–209.

[39] (2016) Arm system memory management unit architecture specification. [Online]. Available: https://developer.arm.com/documentation/ihi0062/dc/

[40] E.-O. Blass and W. Robertson, "Tresor-hunt: attacking cpu-bound encryption," in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 71–78.

[41] (2017, May) Sierraware. [Online]. Available: https://www.sierraware.com/open-source-ARM-TrustZone.html

[42] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[43] M. Zhang and R. Sekar, "Control flow integrity for {COTS} binaries," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 337–352.

[44] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 292–307.

[45] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *NDSS*, vol. 26, 2015, pp. 27–30.

[46] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 941–951.

[47] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[48] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1470–1486.

[49] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 195–211.

[50] D. Jung, M. Kim, J. Jang, and B. B. Kang, "Value-based constraint control flow integrity," *IEEE Access*, vol. 8, pp. 50 531–50 542, 2020.

[51] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17), San Diego, CA*, 2017.

[52] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "Exoshim: Preventing memory disclosure using execute-only kernel code," in *Proceedings of the 11th International Conference on Cyber Warfare and Security*, 2016, pp. 56–66.

[53] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, "Norax: Enabling execute-only memory for cots binaries on aarch64," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 304–319.

[54] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a {TPM} chip," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 841–856.

[55] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on arm," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[56] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, 2018, pp. 1–13.

[57] F. Brasser, D. Kim, C. Liebchen, V. Ganapathy, L. Iftode, and A.-R. Sadeghi, "Regulating arm trustzone devices in restricted spaces," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, 2016, pp. 413–425.

[58] D. J. Sebastian, U. Agrawal, A. Tamimi, and A. Hahn, "Der-tee: Secure distributed energy resource operations through trusted execution environments," *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6476–6486, 2019.

[59] K. Ying, A. Ahlawat, B. Alsharifi, Y. Jiang, P. Thavai, and W. Du, "Truz-droid: Integrating trustzone with mobile operating system," in *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, 2018, pp. 14–27.

[60] 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.

[61] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn, "Prime+ count: Novel cross-world covert channels on arm trustzone," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 441–452.

[62] R. Guanciale, H. Nemati, C. Baumann, and M. Dam, "Cache storage channels: Alias-driven attacks and verified countermeasures," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 38–55.

[63] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 549–564.

[64] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 980, 2016.

[65] J. Wang, K. Sun, L. Lei, S. Wan, Y. Wang, and J. Jing, "Cache-in-the-middle (citm) attacks: Manipulating sensitive data in isolated execution environments," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1001–1015.

[66] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "Cachekit: Evading memory introspection using cache incoherence," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 337–352.

[67] A. Tang, S. Sethumadhavan, and S. Stolfo, "{CLKSCREW}: exposing the perils of security-oblivious energy management," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1057–1074.

[68] D. Suciu, S. McLaughlin, L. Simon, and R. Sion, "Horizontal privilege escalation in trusted applications," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[69] 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*, 2015.

[70] (2021) Pager. [Online]. Available: https://optee.readthedocs.io/en/latest/architecture/core.html#pager

[71] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1723–1740.

[72] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "{PARTEMU}: Enabling dynamic analysis of real-world trustzone software using emulation," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 789–806.

[73] J. Jang and B. Byunghoon Kang, "Retrofitting the partially privileged mode for tee communication channel protection," *IEEE Transactions on Dependable and Secure Computing*, 05 2018.

[74] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek, "Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[75] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves." in *NDSS*, 2019.

[76] M. H. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system." in *NDSS*, 2019.

[77] 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.

[78] (2017, May) Linaro: Op-tee. [Online]. Available: https://www.op-tee.org/

[79] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, "Open-tee – an open virtual trusted execution environment," *2015 IEEE Trustcom/BigDataSE/ISPA*, Aug 2015. [Online]. Available: http://dx.doi.org/10.1109/Trustcom.2015.400

[80] (2017, May) Technology document library. [Online]. Available: https://globalplatform.org/specs-library/?filter-committee=tee

[81] (2018, Feb.) Key apple iphone source code exposed on github in 'biggest leak in history'. [Online]. Available: https://beebom.com/apple-iboot-iphone-source-code-leaked-github/

[82] J. Jang and B. B. Kang, "Selmon: reinforcing mobile device security with self-protected trust anchor," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 135–147.

Jinsoo Jang received the B.S. degree from Ajou University and the M.S. and Ph.D. degrees in information security from the Korea Advanced Institute of Science and Technology (KAIST). He is currently an Assistant Professor with the Department of Computer Science and Engineering, Chungnam National University (CNU). He has been working on systems security areas, particularly in hardening the trusted execution environment (TEE) and leveraging general hardware features to build various defensive measures.

Brent Byunghoon Kang (Member, IEEE) received the BS degree from Seoul National University, Seoul, South Korea, the MS degree from the University of Maryland, College Park, Maryland, and the PhD degree in computer science from the University of California at Berkeley, California. He is currently a professor with the Graduate School of Information Security, Korea Advanced Institute of Science and Technology (KAIST). Before KAIST, he has been with George Mason University as an associate professor. He has been working on systems security area including botnet defense, OS kernel integrity monitors, trusted execution environment, and hardware assisted security. He is currently a member of the USENIX and ACM.