Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

*Digital Object Identifier 10.1109/ACCESS.2020.19013*

# SofTEE: Software-based Trusted Execution Environment for User Applications

**Unsung Lee[1] and Chanik Park[1]**

[1]Departmernt of Computer Science and Engineering, Pohang University of Science and Technology, Pohang 37673, Republic of Korea

Corresponding author: Chanik Park (e-mail: cipark@ postech.ac.kr).

This work was supported by Institute for Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (2020-0-00936, Core Technologies for 5G-Aware Blockchain Networks).

**ABSTRACT** Commodity operating systems are considered vulnerable. Therefore, when an application handles security-sensitive data, it is highly recommended to run the application in a trusted execution environment. In response to this demand, hardware-based trusted execution environments such as Intel SGX and ARM TrustZone have been developed in commodity computers. However, hardware-based approaches cannot be quickly upgraded to address design vulnerabilities or to reflect customer feedback. In this paper, we propose SofTEE, a software framework to support a trusted execution environment for user applications. For a trusted execution environment, SofTEE should support memory isolation and attestation. For memory isolation, SofTEE relies on kernel deprivileging which delegates the execution of privileged operations such as memory management, from a kernel to a special module called a security monitor. To reduce the overhead of switching between the deprivileged kernel and the security monitor, SofTEE proposes an efficient management mechanism of the address space identifier. SofTEE supports attestation by assuming minimal hardware functionalities of random entropy and root of trust. The main challenge of SofTEE is to guarantee security properties like confidentiality and integrity of security-sensitive applications. For security analysis, we have identified security invariants that SofTEE should meet for confidentiality and integrity guarantees. Based on the security invariants, we have designed and prototyped each component of SofTEE on a Raspberry Pi 3 board. SofTEE produces about 3% overhead in case of a security-sensitive application with long execution time and 23% overhead in case of a security-sensitive application with short execution time.

**INDEX TERMS** Address space identifier, kernel deprivileging, trusted execution environment.

## I. INTRODUCTION

Trusted execution environment (TEE) is an isolated environment that protects user code and data from a malicious kernel. It is highly recommended to execute a user application in TEE when the application handles security-sensitive information such as passwords [1], [3], [4], or when the application preprocesses some privacy-sensitive data such as blood pressure and glucose level collected by sensors [2].

There are two fundamental requirements for TEEs: 1) memory isolation; and 2) attestation.

For memory isolation, the address space of TEEs is isolated and protected from a kernel. There are hardware-based solutions, e.g., Intel SGX [6] and ARM TrustZone [5], and software-only approaches, e.g., Virtual Ghost [18] and SKEE [34]. Software-based approaches apply compiler instrumentation or kernel deprivileging to isolate the TEE memory from the kernel memory.

Attestation is a proof to verify that an application is running in a trusted execution environment. To create a proof of attestation, we need to establish a chain of trust in the system. The chain of trust is typically built by using cryptographic hash chains from the beginning of the system boot. Thus, it is critical to create the root key of a hash chain (i.e., root-of-trust (RoT)) in a secure way for correct attestation. The root key of a hash chain is created either by an early bootloader in Komodo [17] or hardware in SGX [6] and trusted platform module (TPM).

Software-based TEEs have several advantages over hardware-based TEEs. First, software-based TEEs are more suitable to be applied in various machine environments because software-based TEEs do not have any dependency on special hardware features, e.g. ARM TrustZone [5], Intel SGX [6]. Second, in software-based TEEs, the update is much easier (and cheaper). The fast upgrade is important to fight against vulnerabilities found later. Third, it does not expand the attack surface of the hardware-based TEE. Thus, software-based TEEs can be used together with hardware-based TEEs.

In this paper, we propose SofTEE, a software framework to implement a TEE for user applications without relying on special hardware features or complicated compiler techniques. For address space isolation, SofTEE applies a technique called kernel deprivileging to delegate some privileged operations, such as memory management, to a special software module called a 'security monitor'. With kernel deprivileging, a CPU privilege mode in SofTEE is logically divided into a normal mode and a secure mode. Based on these two virtual CPU modes, it is possible to support TEEs by software in SofTEE. For attestation, SofTEE assumes Root-of-Trust built by hardware, e.g. TPM.

In SofTEE, the main challenge is how to meet security properties such as confidentiality and integrity of security-sensitive applications. The careful analysis leads us to identify seven security invariants that SofTEE should meet. Then, we have designed each component of SofTEE to satisfy the seven security invariants.

In summary, the contributions of this paper are as follows:

- SofTEE provides a trusted execution environment for secure-sensitive applications, denoted as TAs, without special hardware features. 1) five APIs are defined for TAs (see TABLE I); 2) efficient ASID management without violating security properties is proposed for the performance of TAs; 3) TAs execute with hardware interrupts enabled; 4) TAs are protected from each other, that is, the non-interference property among TAs is guaranteed.
- We defined seven security invariants (I - VII) which are required to support the confidentiality and integrity of security-sensitive applications.
- We prototyped SofTEE based on seven security invariants on a multicore ARM-based Raspberry Pi 3 [8].

The rest of this paper is organized as follows: Section II describes the background, and Section III discusses the threat model and assumptions. Design and implementation are described in Section IV. In Section V, we analyze the security invariants and discuss the confidentiality and integrity of our framework. We evaluate our system in Section VI. We discuss future works and security problems in Section VII. Finally, we conclude our idea in Section VIII.

## II. BACKGROUND

### A. HARDWARE-BASED TEE

CPU vendors have developed hardware techniques to support TEEs (e.g., Intel SGX [6] and ARM TrustZone [5]). Intel SGX protects applications in the enclave memory from the kernel. To reach this goal, Intel SGX isolates and encrypts enclave memory. However, it is not available on other architectures and legacy devices. In addition, SGX-equipped Intel CPU chips cannot be quickly updated to address the vulnerability [37], [54], [55], and to extend features such as the dynamic memory management provided by SGXv2 [6].

ARM TrustZone supports two different worlds: normal and secure. Memory space in the secure world is not accessible from the normal world software, such as the Linux kernel. Thus, ARM TrustZone is generally used as the underlying hardware technology for previous TEE solutions [9], [17], [50], [56]. In these past studies, a trusted software, commonly called a security monitor on ARM TrustZone, detects and blocks kernel accesses to trusted user code and data. However, it also has some limitations. First, ARM TrustZone-based TEEs rely on architecture. Therefore, they are valid only on ARM machines. Moreover, the security monitor in the secure world bloats the trusted computing base (TCB) size of the entire system. Finally, programs running on ARM TrustZone can be targeted by attackers [47]. Therefore, system on chip (SoC) vendors, especially mobile device manufacturers, are reluctant to open ARM TrustZone freely to third-party developers [14], [15]. Instead, some original equipment manufacturer (OEM) applications are exceptionally allowed. This limitation is a significant obstacle when developing a system that leverages ARM TrustZone.

Sanctuary [53] mitigates this limitation of ARM TrustZone by performing a trusted part of an application called trusted application (TA) in the normal world, not in the secure world. In Sanctuary, the system exploits dedicated CPU cores and the TrustZone address space controller (TZASC) to protect the TAs. This approach reduces attack surfaces of the secure world, but Sanctuary still depends on ARM TrustZone and requires privileged software in the secure world at runtime. Besides, the applications and kernel in the normal world cannot use the dedicated CPU cores until the TAs are completed.

Some studies [10]-[14], [46] use hardware virtualization instead of ARM TrustZone to protect trusted applications from the kernel. In the research, hardware virtualization supports memory isolation using a hypervisor. Like a security monitor in ARM TrustZone, a special software called a hypervisor detects and blocks kernel accesses to trusted applications. Thus, hardware virtualization is widely used for address space isolation. However, hardware virtualization has some limitations. First, many devices cannot support hardware virtualization. For example, many legacy internet of things (IoT) devices do not include hardware virtualization, and IoT-oriented Cortex-M series cannot take advantage of hardware virtualization. Besides, hardware virtualization severely degrades performance due to nested paging [14].

Some studies [7], [51] involve architecture modification to replace an existing hardware TEE such as ARM TrustZone. These approaches address the limitations of existing hardware-based TEEs. However, architecture modification is hard in the real world, so the industry rarely adopts such solutions. Moreover, legacy devices cannot take advantage of these solutions.

### B. SOFTWARE-BASED TEE

Virtual Ghost [18] is considered the first study to enforce application security using compiler instrumentation and runtime checks. The structure of this system is similar to hardware virtualization-based TEE in that it runs the operating system (OS) and applications on a virtual machine (VM). However, Virtual Ghost supports VMs without hardware virtualization. Instead, the system uses sophisticated compiler analysis and instrumentation to compile the OS and implement virtualization. Virtual Ghost offers better performance than a hardware virtualization-based approach [10], but Virtual Ghost is much slower than an ARM TrustZone based approach, TrustShadow [9].

Some studies [16], [34] provide trusted kernel execution environments without special hardware features. For example, in SKEE [34], A. M. Azab *et al.* proposed a framework to support the trusted kernel execution environment. In SKEE, an address space separation was used to isolate the trusted kernel execution environment from the compromised kernel execution environment. However, SKEE does not consider the trusted execution environment for user applications. On the other hand, SofTEE provides TEE for user applications, including the trusted kernel execution environment. Therefore, the security monitor of SofTEE tackles security problems not considered by SKEE.

### C. KERNEL DEPRIVILEGING

Currently, most commodity OSes are based on monolithic kernels, which can be easily compromised, and attackers gain complete control over all kernel functionalities, including memory management. Therefore, there have been many attempts to reduce kernel authorities. For example, microkernels [19]-[21] delegate some kernel functionalities, such as device drivers and filesystem, to user processes. By doing so, microkernels are generally lighter than monolithic kernels. However, microkernels require redesign and extensive code modifications from commodity OSes.

Another approach is to delegate certain kernel functionalities, such as memory management and privileged instructions (e.g., memory management unit (MMU) configuration updates) to a kernel module, and separate the module from the kernel. In this solution, there are two main ways to isolate the module from the kernel.

The first way is to put the module and the kernel in the same address space. Instead, the module is hidden from the kernel via address space randomization (ASR) [32] or protected by write protection (WP) in x86 [16]. The system with ASR is less likely to be attacked, so the module is expected to be safe, but in practice, it is not secure. Some studies [22]-[27] have shown that rerandomization or runtime code randomization is necessary to mitigate memory disclosure attacks [49]. Meanwhile, the system using WP isolates the module from the kernel, but this hardware technique is not supported on some architectures such as ARM. Besides, some security monitor memory in SofTEE is not mapped to the kernel to ensure confidentiality. Therefore, we cannot use these methods to isolate the module.

The other way is to use the shadow page table to separate the module address space from kernel address spaces [28], [34]. In general, address space separation using shadow page table is widely used, but significant overhead is inevitable due to the memory management unit (MMU) configurations and TLB maintenance [29]. In some research [34], [69], the overhead of context switching between a normal kernel execution environment and a trusted kernel execution environment is reduced by reserving an ASID for the trusted kernel execution environment. Unlike the previous research, however, we considered ASID management for trusted applications. In SofTEE, ASID management is considered a privileged operation. Thus, the security monitor handles ASID management directly. In Section IV.A, we describe the detailed design of the security monitor.

## III. THREAT MODEL AND ASSUMPTIONS

We consider that attackers may completely compromise the kernel. Once the kernel is compromised, the attacker can access and modify any data and registers in the system. Therefore, the attacker can arbitrarily control memory address space by manipulating page tables as well as MMU configurations. In addition, kernel control paths may also be changed by manipulating kernel control flow data.

We assume that our binary scanner for kernel deprivileging is functionally correct. The binary scanner scans the kernel image and replaces all the targeted privileged instructions with explicit calls to the security monitor. We also assume that the security monitor, the main component of the trusted computing base, is bug-free.

Direct memory access (DMA) attacks, other hardware attacks such as bus monitoring [38], denial-of-service (DoS) attacks, and side channel attacks using shared hardware resources are out of scope. Note that previous works [44], [56] can be easily applied to address these hardware attacks.

We assume that root-of-trust (RoT) and random entropy are provided by hardware. In SofTEE, RoT is required for attestation, and random entropy is critical for cryptographic key creation.

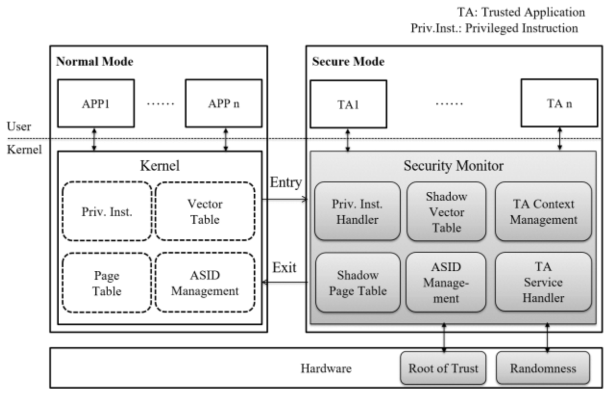


**FIGURE 1.** SofTEE architecture. The system has two modes: normal mode and secure mode. Normal mode memory includes applications and kernel memory. Secure mode, on the other hand, consists of TAs and the security monitor. Normal mode and secure mode are converted through the dedicated Entry and Exit Gates. Note that shaded boxes show the TCB and dotted boxes show deprivileged functionalities.

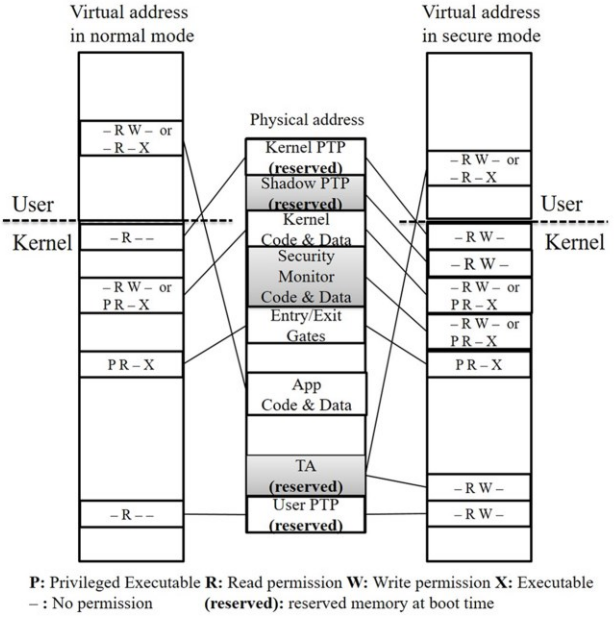


**FIGURE 2.** Process address space in SofTEE. In SofTEE, each process address space consists of virtual address spaces for both normal and secure modes. In the normal mode, its virtual address consists of normal user application and the kernel memory. In the secure mode, the virtual address consists of secure user application called TA and the security monitor memory. Note that a certain region (shaded box) of physical memory called secure page is exclusively accessed by the security monitor. In addition, some physical memory is reserved to store page-table pages (PTPs) and TA.

## IV. DESIGN AND IMPLEMENTATION

### A. DESIGN

Fig. 1 shows the SofTEE architecture consisting of a (deprivileged) kernel and a (privileged) security monitor. To build the deprivileged kernel, the first step is to apply an offline binary scanner to the kernel binary image. Given the list of privileged instructions for kernel deprivileging, the scanner detects the targeted instructions in the kernel binary image and replaces them with an explicit call to the security monitor (i.e., the entry gate in Fig. 1). Next, we manually add an explicit call to the entry gate before some of the privileged operations, such as page table updates, and an explicit call to the exit gate after completing the privileged operations.

With the help of kernel deprivileging, SofTEE logically separates a physical CPU mode into two modes: normal and secure. The deprivileged kernel runs in the normal mode whereas the security monitor runs in the secure mode. In Fig. 1, dotted boxes in the kernel represent deprivileged operations that are handled by the security monitor. Switching between normal and secure modes is handled by the entry gate and the exit gate.

Each application consists of two parts: a security-insensitive normal part and a security-sensitive trusted part. In Fig. 1, the normal part is denoted as APP and the trusted part is denoted as TA. The interactions between the APP and TA are specified by application programming interfaces (APIs) provided in SofTEE. We will discuss which operations are provided for the APP and TA later.

**Privileged instruction handler.** In SofTEE, several privileged instructions are removed from the kernel. These instructions include: 1) the instruction that updates registers containing the page table base address and the vector table base address; 2) the instruction that handles ASID; and 3) the instruction that changes the control register that manages the MMU. These instructions are handled by the privileged instruction (Priv. Inst.) handler in the security monitor. This approach is similar to the one in previous research [16], [32], [34].

**Shadow page table.** For address space isolation, the security monitor manages its own page table, the shadow page table. The shadow page table concept was originally developed in the hypervisor to support efficient memory virtualization [12], [33].

Fig. 2 shows how the security monitor manages the address spaces for a process. Because a user application consists of two parts: a security-insensitive APP and a security-sensitive TA, the address space of the process is defined by two address spaces: a virtual address space for APP in the normal mode and a virtual address space for TA in the secure mode. Note that in the virtual address map of the normal mode, accesses to both the kernel page-table pages (Kernel PTP) and the user page-table pages (User PTP) are restricted to read-only. On the other hand, both read and write permissions are allowed in the virtual address map of the secure mode.

Some physical memory spaces are reserved for the security monitor to store TA contexts and all page tables. Any information stored in the reserved areas is safe from the malicious kernel. This is because the reserved memory spaces are isolated from the memory spaces managed by the kernel. In SofTEE, there is no mapping with access permission of both write and execute. Thus, attackers cannot inject new code into the preloaded kernel code and data areas, known as code injection attack [35]. However, malicious users can collude with the kernel to inject privileged instructions. For example, the malicious user attempts to load applications or kernel modules that contain privileged instructions. Therefore, user application memory should be mapped without privileged executable permission, and the kernel modules should be scanned as if the kernel image was scanned. Even worse, some kernel modules, such as BSD packet filtering [48], generate code dynamically at runtime. SofTEE should make sure that the generated code does not contain any of the privileged instructions. (We have not implemented it yet.)

Privileged instructions can easily be detected by the binary scanner, but it is not easy to detect privileged operations such as memory management by the binary scanner. Therefore, privileged operations such as memory management may be included stealthily in the kernel modules. SofTEE does not enforce that kernel modules cannot contain these operations. Instead, the kernel modules are executed in the normal mode. Basically, in the normal mode, the kernel has read-only permission to all page table pages and cannot access the security monitor code and data. This is because SofTEE ensures memory isolation through a security monitor page table called the shadow page table. Therefore, like the kernel, the modules are not authorized to perform the privileged operations (e.g., memory management).

SofTEE prevents code injection attacks, but malicious users attempt code reuse attacks, such as return-oriented programming (ROP) attacks. We consider two types of attacks (return-to-user (ret2usr) [63] and return-to-direct-mapped memory (ret2dir) [64]) to discuss about ROP attacks. In these attacks, an attacker uses multi-stage shellcode by grouping user and kernel shellcode. The attacker can inject kernel code or connect several code fragments (gadgets) to construct the kernel shellcode. In particular, the attacker can construct the kernel shellcode via sophisticated rootkits such as return-oriented rootkits [71].

In ret2usr, the attacker exploits memory corruption vulnerabilities to force the kernel to redirect to user code and data. To prevent this attack, Intel introduced supervisor mode execution prevention (SMEP) [65] and supervisor mode access prevention (SMAP) [66]. Similarly, ARM introduced privileged execute never (PXN) [67] and privileged access never (PAN) [68]. In SofTEE, we assume SMEP (Intel) or PXN (ARM) enabled to prevent the kernel from executing privileged instructions in user code. On the other hand, the ret2dir attack can bypass these hardware features because the ret2dir attack exploits physical memory mapping of the kernel to user code and data. The paper [64] proposed a method of exclusive page frame ownership (XPFO) to handle the ret2dir attack. XPFO explicitly controls the kernel memory mapping to user processes. In SofTEE, it is assumed that SofTEE applied the technique XPFO to handle the ret2dir attack.

**TA context management.** The security monitor maintains TA context information consisting of an identifier, an interrupt status flag, an assigned ASID information, general-purpose registers, and some secure pages. Additionally, SofTEE allows a shared memory page set up for efficient communication between APP and TA. SofTEE enforces any secure operations of user application (e.g., operations accessing private information) to run inside TAs. Only the results of secure operations are exposed to APPs. By doing so, the security monitor of SofTEE can enforce any private information accessed only inside TAs. Therefore, the attack on shared memory between APP and TA cannot violate the security properties enforced by SofTEE. However, such an attack scenario should be considered in practical environments. SofTEE can handle this attack by upgrading the security monitor to manage communication channels between APP and TA. The detailed steps are shown as follows: 1) reserving some physical address spaces (and virtual address spaces) for use as shared memory between APP and TA at kernel boot and 2) Preventing the kernel from accessing the shared memory between APP and TA. For example, privileged access never (PAN) of ARM or supervisor mode access prevention (SMAP) of Intel does not allow the kernel to access the user data field. It can also be prevented by software techniques that intentionally remove shared memory between the kernel and user, such as exclusive page frame ownership (XPFO) [64]. When applying XPFO, the security monitor may reject a kernel request asking a mapping to the shared memory between APP and TA. Note that all page table updates are handled by the security monitor in SofTEE.

**TA service handler.** Table I shows the APIs available for applications consisting of an APP and a TA in SofTEE. Each API is served by the TA service handler. Among the nine APIs, four APIs are for the APP and five APIs are for the TA.

For noninterference between TAs, SofTEE does not allow shared memory between TAs.

1) **Create_TA**: This API is provided to create a TA specified in the argument. The security monitor is responsible for loading the code and data of the TA into the main memory and then initializing the TA context.

2) **Remove_TA**: This API is provided to remove a TA context. This operation deallocates the secure pages assigned to the TA.

3) **Enter_TA and Reenter_TA**: These APIs are required to run a TA. 'Enter_TA' is invoked by the APP when the TA executes for the first time. Note that the security monitor invokes the execution of the TA with interrupts enabled. There may be

TABLE I
SofTEE APIs PROVIDED TO APPLICATIONS (APP AND TA)

| **APIs for APP** | |
|---|---|
| Create_TA(char *name, void *shared_mem) -> u32 result | Create a TA context |
| Remove_TA() | Remove a TA context |
| Enter_TA(u32 arg0, u32 arg1, u32 arg2, u32 arg3) -> u32 result | Execute a TA context |
| Reenter_TA() -> u32 results | Execute a TA after the TA stops due to an interrupt |
| **APIs for TA** | |
| Attest(u32 *data, u32 *sig) -> u32 result | Make the attestation of a TA identity |
| Verify(u32 *data, u32 *sig) -> u32 result | Verify the attestation of a TA |
| GetRandom(void *random_mem, int random_bytes) -> u32 result | Get the hardware random numbers |
| GetHash(void *hash_mem, void *input_mem, u32 input_size, int hash_alg) -> u32 result | Get the hash value of an input |
| Exit(u32 ret0) | Stop a TA |

interrupts generated during the execution of the TA. These interrupts are handled by the interrupt handler in the deprivileged kernel. When the interrupt handling is completed, the control returns to the APP in the normal mode, not to the TA, even though the execution of the TA was interrupted. Later, the APP needs to invoke 'Reenter_TA' to resume the execution of the TA.

4) **Attest and Verify**: These APIs are fundamental for attestation. Like the local attestation protocol provided by Intel SGX, SofTEE provides local attestation (verifier in the same machine). In this process, a TA requests the 'Attest' operation and the handler deals with this operation. For this operation, the security monitor has a key pair. During the 'Attest' operation, the monitor signs the current TA identity and TA-provided data (256 bits) using a private key, then the handler returns the result (a signature) to the requesting TA. After receiving the result, the requesting TA invokes the 'Verify' operation to verify that the trusted code is running on the SofTEE platform. In this procedure, a public key is used.

5) **GetRandom and GetHash**: These APIs are great help for cryptographic operations. When the TA calls these operations, the handler returns a hardware random seed and a hash value, respectively, to the requesting TA. Note that, SofTEE should support hardware randomness. This is because the security level of the created key definitely depends on the randomness of the seed. In addition, by providing hash algorithms for TA developers, they can build their software without relying on external libraries.

6) **Exit**: This API is provided to return to the APP after completing the execution of the TA.

**Shadow vector table.** Note that TAs are executed with interrupts enabled by the security monitor. Thus, interrupts or exceptions may be generated during the execution of TA in the secure mode. Since current CPU context information, e.g. general-purpose registers, is exposed to interrupt or exception handlers, the security monitor should maintain its own vector table, called shadow vector table. The shadow vector table saves the context information of the currently running TA.

Since a page fault exception of a TA can be directly handled by the security monitor, secure page usage information is not disclosed to the deprivileged kernel. Therefore, SofTEE prevents page-fault based side channel attacks [58]. In SofTEE, other faults like a hardware interrupt are handled by the interrupt handlers available in the deprivileged kernel. For dispatching interrupt handling, we need to switch the privilege mode from secure to normal. This is done by the exit gate. For security assurance, SofTEE cleans up general-purpose registers and cleans all the data cache entries accessed by the TA before mode switching (see more details in Section VII).

**ASID management.** Context switching overhead is typically high due to the invalidation of TLB entries. Traditionally, the address space identifier tag (e.g., address space identifier (ASID) in ARM and process context identifier (PCID) in Intel) is attached to each TLB entry to reduce context switching overhead.

Since stale TLB entries may be abused by the attacker [36], SofTEE has to flush all TLB entries before returning from secure mode to normal mode. SofTEE mitigates the mode

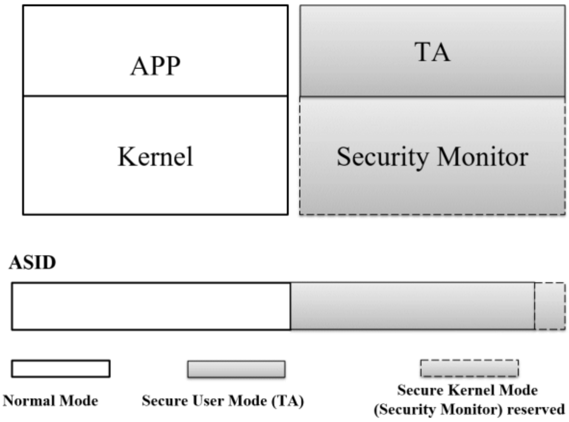


**FIGURE 3.** **ASID management. Each address space has each range of ASID. In SofTEE, the last ASID is reserved for the security monitor. On the other hand, the kernel uses the first ASID.**

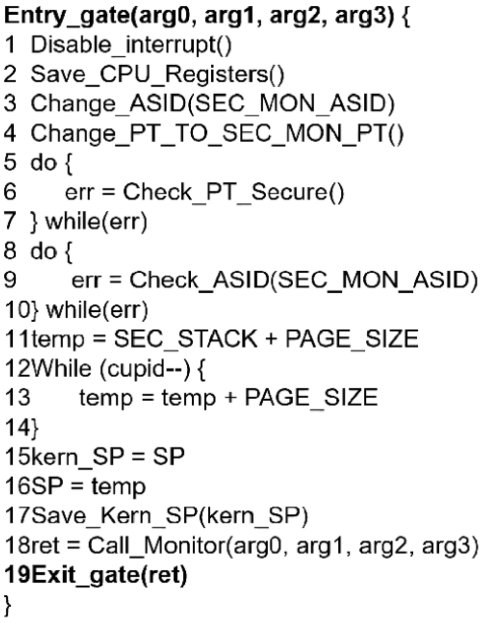
switching overhead by efficient management of address space identifiers.

ASID management is considered a privileged operation and is handled by the security monitor. In SofTEE, the entire ASID space is partitioned into two spaces (see Fig. 3). Note that some ranges of ASIDs are exclusively handled by the security monitor to safely assign the ASID to each TA context. In addition, memory in the secure mode is mapped as non-global, so cached TLB entries in the secure mode address space are only available to ASIDs that are exclusively assigned to the secure mode. With this ASID management, SofTEE does not have to flush TLB entries at every mode switch.

In SofTEE, the key point of ASID management is non-interference property while reducing switching overhead. In particular, SofTEE decides how many ASID entries are assigned to TAs. In addition, SofTEE selectively flushes only TLB entries corresponding to the specified ASIDs in the secure mode. It prevents redundant TLB flush operations. These ASID entries are carefully managed by the security monitor in order to meet the non-interference property of TAs.

**Entry and exit gates.** If the deprivileged kernel in the normal mode wants to execute privileged operations of the security monitor, it should be done through the entry gate. The entry gate takes care of mode switching from the normal mode to the secure mode by updating address space, and then the security monitor is invoked to execute the requested operation. After the operation is complete, the control is returned to the deprivileged kernel through the exit gate. The exit gate is responsible for switching the address space from secure mode to normal mode.

Fig. 4 and Fig. 5 show the pseudo-codes for the entry and exit gate. Note that it is possible for a compromised kernel to jump into the middle of the entry or exit gate. Therefore, the entry gate and exit gate include security validation codes.



```
Entry_gate(arg0, arg1, arg2, arg3) {
1  Disable_interrupt()
2  Save_CPU_Registers()
3  Change_ASID(SEC_MON_ASID)
4  Change_PT_TO_SEC_MON_PT()
5  do {
6      err = Check_PT_Secure()
7  } while(err)
8  do {
9      err = Check_ASID(SEC_MON_ASID)
10} while(err)
11temp = SEC_STACK + PAGE_SIZE
12While (cupid--) {
13      temp = temp + PAGE_SIZE
14}
15kern_SP = SP
16SP = temp
17Save_Kern_SP(kern_SP)
18ret = Call_Monitor(arg0, arg1, arg2, arg3)
19Exit_gate(ret)
}
```

**FIGURE 4.** **Entry Gate pseudo code.**

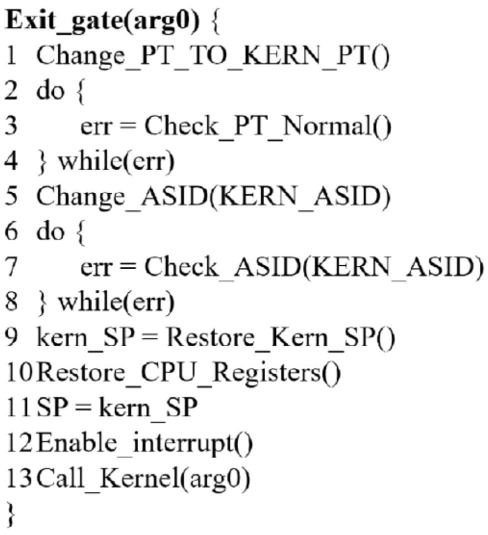
The entry gate works as follows:

**Line 1.** Disable interrupt. Note that the security monitor reenables interrupts just before the TA executes.

**Line 2.** Save kernel's general-purpose registers except for the stack pointer (SP). The saved registers will be used to restore the kernel's general-purpose registers while the exit gate is executing. Therefore, the kernel cannot obtain information about the secure mode through the general-purpose registers. This step is vital to ensure confidentiality.

**Line 3.** Change the ASID from the kernel ASID to the security monitor ASID (i.e., SEC_MON_ASID).

**Line 4.** Change the page table from the normal mode page table to the secure mode page table. The entry gate should be deterministic despite the compromised kernel attempts to jump to the middle of the entry gate. For the entry gate, in SKEE [34], the authors suggested two solutions: 1) setting the kernel and secure kernel to use different translation table base registers (TTBR0 and TTBR1) respectively, and 2) using a zero register to update the TTBR register. In addition, Hilps [69] adjusted the virtual address range by updating a control register called the translation control register (TCR) at the entry and exit gates. In this case, the gates need to check whether the control register value is valid immediately after updating the register. In SofTEE, we can use these solutions to switch address space from normal mode to secure mode. The Change_PT_TO_SEC_MON_PT() switches page table without input argument. In means that the page table is switched regardless of the value of general-purpose registers.



```
Exit_gate(arg0) {
1  Change_PT_TO_KERN_PT()
2  do {
3      err = Check_PT_Normal()
4  } while(err)
5  Change_ASID(KERN_ASID)
6  do {
7      err = Check_ASID(KERN_ASID)
8  } while(err)
9  kern_SP = Restore_Kern_SP()
10 Restore_CPU_Registers()
11 SP = kern_SP
12 Enable_interrupt()
13 Call_Kernel(arg0)
}
```

**FIGURE 5.** Exit Gate pseudo code.

**Lines 5-7.** Check_PT_Secure() instruction of the entry gate verifies that the page table is switched from normal mode to secure mode.

**Lines 8-10.** Check the validity of the ASID range. In the normal case, ASID is changed from the kernel ASID to the security monitor ASID in Line 3, but attackers can skip Line 3 (e.g., jumping to Line 4). This attack is handled by checking the validity of ASID in Line 8-10. This code checks whether current ASID is the security monitor ASID. If the ASID is not valid, the entry gate prevents the kernel from proceeding further.

**Lines 11-14.** Calculate the stack address to be used while the security monitor is running. SofTEE provides the security monitor with as many special stacks (called secure stacks) as the number of CPUs. Note that the secure stacks (i.e., SEC_STACK) are not accessible in the normal mode.

**Lines 15-17.** Save the current kernel SP (i.e., kern_SP) and change the hardware stack to the secure stack of this CPU.

**Line 18.** Jump to the security monitor code.

**Line 19.** Call the exit gate to switch mode from secure to normal.

The exit gate restores all the information saved by the entry gate and returns back to the kernel. Under normal circumstances, the exit gate is only called by the security monitor. However, attackers can invoke the exit gate directly. In order to handle this attack, the exit gate (Lines 6-8) verifies that an ASID matches the ASID of the normal mode kernel (i.e., KERN_ASID). In addition, Check_PT_Normal() instruction (Lines 2-4) of the exit gate verifies that the page table is switched from secure mode to normal mode. For the exit gate, in SKEE, the authors suggested two solutions: 1) if the kernel and secure kernel are set to use TTBR0 and TTBR1, respectively, the exit gate checks whether a control register value (i.e., translation table base control register (TTBCR)) is valid immediately after switching the page table, and 2) Placing the page table update instruction (e.g., Line 1 in Fig. 5) at the end of the physical page that is isolated from the kernel. In SofTEE, we can use these solutions to switch address space from secure mode to normal mode.

The first step (Line 1) of the entry gate is to disable local interrupts, but attackers can skip this step by jumping to Line 2. In order to handle this attack, we add a security check code in the kernel interrupt handler. The interrupt handler in the normal mode checks the current address space such as the page table and ASID range. If the current address space is not the kernel address space, then the system stops. Even if an attacker attempts to delete (or modify) or bypass the security check code in the interrupt handler, the attacker will not be able to skip the security check code for two reasons. First, SofTEE does not allow code injection attacks by setting all memory to w xor x (W^X). It enforces non-writable code and non-executable data. Next, in SofTEE, updating the vector base address register is considered a privileged instruction, so SofTEE monitors the vector table base address register updates.

### B. IMPLEMENTATION

This section describes the detailed implementation of each component of the security monitor. We prototyped our framework on a Raspberry Pi 3 board [8]. This development board is widely used because of its low cost and small size. This machine includes a quad-core 1.2GHz 32-bit CPU and 1 GB RAM. (The recent Raspberry Pi 3 model includes 64-bit CPU, but the initial version of this model contains ARMv7 [42], which is based on 32-bit CPU.) In addition, we installed a hardware TPM on the board. In terms of software, we downloaded and installed Ubuntu 16.04.5 LTS with the Linux kernel version 4.14.95 on the machine. We mainly followed the kernel configuration set by the hardware manufacturer but modified some kernel configurations to support hardware TPM and large physical address extension (LPAE).

**Security monitor.** SofTEE removes the memory management and ASID management privileges from the kernel. Besides, SofTEE changes the exception and interrupt vectors before a TA runs to safely manage the TA context. Thus, the kernel is deprivileged. Instead, the security monitor performs these privileged operations on behalf of the kernel.

SofTEE follows two main steps for the kernel deprivileging: 1) the binary scanner replaces all target privileged instructions with explicit calls to the entry gate; 2) the shadow page table separates the security monitor address space from the kernel address space. First, for binary scanning, we configure the scanner to read and replace fixedsize instructions. The scanner looks for instructions which update certain coprocessor registers such as TTBR0, TTBR1, translation table base control register (TTBCR), and system control register (SCTLR).

Next, for address space isolation, we configure read-only page tables in the normal mode. Thus, the deprivileged kernel can only update the page table through the security monitor. When the security monitor receives a request from the deprivileged kernel, the monitor verifies that the request is valid. For example, the malicious kernel may try to run user code. In this case, the security monitor would check the privileged execute never (PXN) bit in the page table entry. This hardware technique prevents the malicious kernel from executing code in user memory. The security monitor always sets the PXN bit when creating mappings for user memory.

**Minimum hardware support.** In the prototype, SofTEE uses a hardware TPM that supports 2.0 specification. This hardware module includes a permanently embedded encryption key called an endorsement key. Besides, this hardware module provides the hardware random number generator. Note that the TPM 2.0 specification is known to prevent side channel attacks such as timing attacks [30].

The following is performed to include the TPM device in SofTEE. First, we need to modify the software running in the secure kernel-mode: 1) remove some mappings from the normal mode page table to prevent the kernel from accessing the TPM data area; 2) assign dynamically allocated variables in the TPM device driver to memory area protected by the security monitor; 3) change interrupt-driven input/output (I/O) to polling-based I/O; and 4) add memory mappings of the device registers in the shadow page table.

Second, we need to port some TPM software stack (TSS) source code [43] to the secure user mode. With the help of the TSS, SofTEE is able to send TPM commands to the TPM device driver. The TSS in the secure user mode handles all the requests from the TA. The operation sequence of the TSS in SofTEE is as follows: 1) the TA service handler forwards a TA request to the TSS; 2) The TSS handles the request by sending TPM commands to the hardware TPM; 3) the TPM returns a result to the TSS; 4) the TSS forwards the result to the handler; and 5) the handler sends the result to the requesting TA.

In the prototype, the TSS handles only one TA request at a time for simplicity, so the security monitor returns an error (Device is busy) if another TA is waiting for the result of the service. To accomplish this, the security monitor provides a global flag to check that the TSS is free.

## V. SECURITY ANALYSIS

### A. SECURITY INVARIANTS

SofTEE meets seven security invariants as follows. Invariants I - IV are for the relationship between the normal mode and secure mode whereas Invariants V - VII are for the relationship between software in the secure mode.

**Invariant I.** Memory management and ASID management are handled only by the security monitor in the secure mode.

**Invariant II.** The address space of the normal mode and the secure mode is separated.

**Invariant III.** The kernel cannot modify the security monitor code and data.

**Invariant IV.** Mode switching between the normal mode and the secure mode is done only via the dedicated entry and exit gates.

**Invariant V.** The address space of each TA is separated.

**Invariant VI.** Each TA context information including general-purpose registers is correctly saved and restored by the security monitor.

**Invariant VII.** The TA cannot access the security monitor code and data.

### B. SUPPORTING INVARIANTS

**Supporting invariant I.** The privileged operations such as memory management and ASID management are removed from the kernel by the binary scanner and shadow page table. Thus, the privileged operations are only handled by the security monitor.

**Supporting invariant II.** The security monitor supports an independent address space in the secure mode, isolated from the normal mode. To build the independent address space, SofTEE provides shadow page table and reserved ASIDs in the secure mode. Thus, the address space of the normal mode and the secure mode is separated.

**Supporting invariant III.** The address space of both modes is separated by Invariant II. Besides, the kernel cannot access the security monitor code and data, because the security monitor checks the address and access permissions each time the page tables in the normal mode are updated. Therefore, the integrity of the security monitor code and data is protected.

**Supporting invariant IV.** The monitor code and data are protected by address space separation (Invariant II). In addition, the kernel is deprivileged, so the kernel should call the entry gate to execute privileged operations. Likewise, the monitor returns to the kernel using the exit gate. As described in Section IV-A, our framework detects and blocks malicious behaviors such as jumping in the middle of the entry or exit gate. Therefore, the mode switching is done only via the dedicated entry and exit gates.

**Supporting invariant V.** The security monitor handles the secure pages and reserved ASIDs to manage each TA context. To maintain isolation between the TAs, the security monitor follows some rules: 1) the security monitor allocates secure pages that have not been previously assigned to other TA contexts; 2) the security monitor does not allow shared pages between the TAs; 3) the security monitor directly manages the ASID and assigns different hardware ASIDs for each TA context; and 4) the ASID in the deleted TA context is not immediately reused. Instead, the security monitor initializes all ASIDs in the secure mode at once when no ASID is available in the secure mode. Thus, the address space of each TA is separated.

**Supporting invariant VI.** The shadow vector table directly traps interrupts or exceptions generated during the execution of a TA. If a trap is from a valid cause, the shadow vector table saves the general-purpose registers in the current TA context (except for the 'Exit' operation). Similarly, the shadow vector table restores all general-purpose registers from the current TA context before running the TA. Therefore, each TA context is correctly saved and restored by the security monitor.

**Supporting invariant VII.** The security monitor creates page table mappings of each TA, and the security monitor does not allow the TA to access the monitor's code and data.

### C. CONFIDENTIALITY AND INTEGRITY

According to our threat model, attackers can compromise the kernel and install malicious TAs in the secure mode. Therefore, we should meet the confidentiality and integrity conditions of the security monitor and TAs during their lifetime.

Malicious attackers may try to access memory or general-purpose registers that contain security-sensitive information.

**Security monitor confidentiality.** The confidentiality of the security monitor is proven as follows:

1) We need to show that the kernel cannot get any unpermitted information from the security monitor. First, the kernel cannot execute privileged operations or access secure pages that are protected by the security monitor (Invariant II). Second, the only way for the malicious kernel to execute privileged operations or access secure pages is mode switching from normal to secure through the entry and the exit gate (Invariant IV). When the exit gate switches the mode from secure to normal, the gate flushes the general-purpose registers of the security monitor. Thus, the kernel cannot get any unpermitted information of the security monitor.
2) When the malicious attacker is a TA, the malicious TA cannot access the security monitor code and data directly by the Invariant VII. Moreover, malicious TA cannot get the security monitor information via general-purpose registers by the Invariant VI.

**Security monitor integrity.** SofTEE protects the integrity of the security monitor: 1) when the malicious attacker is a kernel, the kernel cannot modify the security monitor code and data by the Invariant III; and 2) when the malicious attacker is a TA, the malicious TA cannot modify the security monitor code and data by the Invariant VII.

**TA confidentiality.** The Confidentiality of the TA is proved as follows:

1) When the malicious attacker is a kernel, it cannot access secure pages directly by the Invariant II. In addition, the kernel cannot get TA context information via general-purpose registers by the Invariant IV. Therefore, the malicious kernel cannot get any unpermitted information from the TA.
2) When the malicious attacker is a TA, the malicious TA cannot access secure pages of other TA contexts by the Invariant V. Moreover, the malicious TA cannot obtain other TA context information via general-purpose registers by the Invariant VI.

**TA integrity.** SofTEE protects the integrity of the TA: 1) when the malicious attacker is a kernel, the kernel cannot directly modify secure pages by Invariant II; and 2) when the malicious attacker is a TA, it is not possible for the malicious TA to change secure pages assigned to other TA contexts (Invariant V).

## VI. PERFORMANCE EVALUATION

In SofTEE, an application consists of two parts, an APP and a TA. The address space of the TA is isolated from the malicious kernel. Thus, the TA can handle security-sensitive data securely regardless of kernel hacking. This is possible due to the software technique called kernel deprivileging. However, kernel deprivileging incurs high overhead due to the delegation of privileged operations to the security monitor. To mitigate the performance overhead required by kernel deprivileging, several numbers of ASIDs are reserved for TA execution in SofTEE.Since the number of TLB flushes increases as the number of available ASIDs decreases, we need to determine how many ASIDs are reserved for TA execution. For this purpose, we have conducted experiments using the fork+exit benchmark of the LMBench suite [45], and the repetition option (-N) was modified to 2,500 for the stress test. The default repetition in the LMBench version 3.0-a9 is 11. The experiment was repeated 100 times and the average was calculated to get accurate results. Besides, we conducted the same experiment on SofTEE and Linux to get relative performance. We tested the benchmark by changing the number of ASIDs in the normal mode. The security monitor reserves one ASID for itself, so when testing the benchmark on SofTEE, at least one ASID should be assigned to the secure mode.

TABLE II
FORK+EXIT (WITH 2,500 REPETITIONS) RESULTS. NOTE THAT ARMV7-A SUPPORTS 256 ASIDS.

| Mode | ASID | Time of normal user mode (Relative to Linux kernel) |
|---|---|---|
| Normal | 255 | 1.76 x |
| Secure | 1 | |
| Normal | 239 | 1.87 x |
| Secure | 17 | |
| Normal | 223 | 1.91 x |
| Secure | 33 | |
| Normal | 191 | 1.89 x |
| Secure | 65 | |
| Normal | 127 | 2.00 x |
| Secure | 129 | |

Table II shows SofTEE's APP latency according to the number of ASIDs in the normal mode. As expected, the overhead of APP increased as the number of ASIDs in the normal mode decreases. However, when the number of ASIDs in the normal mode was 239, 223, and 191, there was little performance difference. Thus, 65 ASIDs are proper for the secure mode, so we reserved and assigned 65 ASIDs to the secure mode.

## A. MICRO BENCHMARKS

In order to evaluate the overhead of SofTEE, we executed LMBench benchmarks in the normal mode and measured the execution time relative to the Linux kernel. In addition, we also measured the execution time of SofTEE without ASID management to show how effective the ASID management works. Fig. 6 shows the results obtained from the average over 100 experiments.

The main overhead of SofTEE comes from memory isolation by kernel deprivileging. We have measured that SofTEE consumes 388 CPU cycles for memory isolation. When ASID management is not applied, it increases to 1,784 CPU cycles.

In Fig. 6, SofTEE is up to 1.5 times slower than the Linux kernel in most cases except pagefault, signal handler install, and fork+exit. The worst case is the fork+exit benchmark. SofTEE is 1.89 times slower than the Linux kernel. This is because fork+exit benchmark requires much more frequent mode switching operations caused by kernel deprivileging. In the case of the null benchmark, significant performance degradation is observed. This is because the execution time of the null benchmark is very short.

It is also shown in Fig. 6 that ASID management is quite effective to reduce the mode switching overhead by avoiding TLB flush operations. Specifically, in cases of mmap, pagefault, fork+exit, and fork+exec benchmarks, the performances degrade approximately 2.2 to 3.2 times compared to the Linux kernel when ASID management is turned off in SofTEE. On the other hand, SofTEE is only about 1.3 to 1.89 times slower than the Linux kernel when ASID management is turned on. This is because we can avoid frequent TLB cache flushing by ASID management.

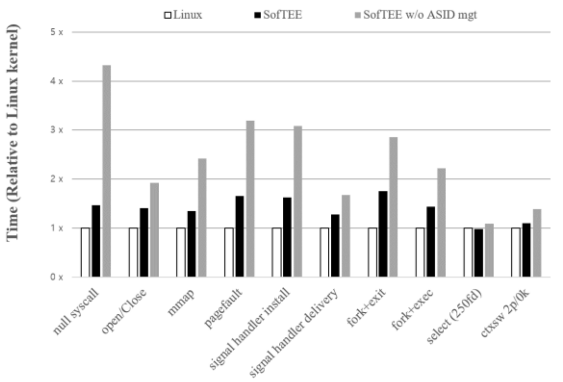


FIGURE 6. LMBench results.

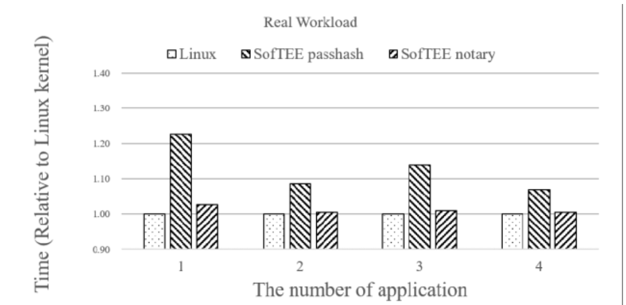
## B. REAL WORKLOAD

For real world applications, we consider PassHash and Notary applications described in Ironclad [31]. These applications consist of client and server parts where the server part handles security-sensitive data. For experiments, we reimplemented the server parts of Passhash and Notary as standalone C programs (281 LOC and 384 LOC, respectively).

The Passhash client and server work as follows: 1) when the server receives a buffer from the client, the server parses this buffer to gain the client's password and salt; 2) the server then combines random numbers with the client's password and salt to compute a hash value; and 3) finally, the server sends this hash value to the client.

The Notary client and server work as follows: 1) server application creates an RSA key pair and initializes a monotonic counter; 2) when the server receives a buffer from the client, the server parses this buffer to gain the client's document; 3) the server increments the counter; 4) the server hashes this document with the current counter value and signs it using the server RSA key; and 5) the server returns the result to the client. RSA key creation and signing can be implemented in software using the provided APIs in Table Ⅰ. However, for simplicity, we tested the Notary application by adding some APIs (i.e., RSA key creation and hash & sign operations).

In order to evaluate the overhead of SofTEE, we measured the execution time of Passhash and Notary applications on Linux and SofTEE. Fig. 7 shows how the performance changes when the number of security-sensitive applications, that is, the number of TAs, increases. We measured the application execution time both on Linux and SofTEE. For a fair comparison, we installed TPM in Linux so that all cryptographic operations are supported by hardware TPM.



| The number of Application | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| **Passhash** (ms)<br>SofTEE/Linux | 42/34 | 64/59 | 88/77 | 105/98 |
| **Notary** (ms)<br>SofTEE/Linux | 543/529 | 1073/1066 | 1634/1620 | 2225/2213 |

**FIGURE 7.** PassHash and Notary results in Linux and SofTEE.

It is shown in Fig. 7 that SofTEE incurs about 1-3% overhead in case of Notary whereas it incurs 6-23% overhead in case of Passhash. Because the execution time of Notary is much longer than the execution time of Passhash, the overhead of Notary is much lower.

Note that the overhead of SofTEE comes mainly from cache flush and invalidation caused by mode switching between application and TA. This overhead cannot be avoided simply by ASID management because SofTEE should guarantee the confidentiality and integrity of TAs.

The overhead shown in Fig. 7 also includes the overhead of the TA service handler which initializes TPM software stack (TSS) and handles system calls issued by TAs. Because TSS should coordinate the concurrent accesses to TPM from multiple TAs, the execution time of a TA increases as the number of TAs increases. That is why the overhead is reduced when multiple TAs are executed simultaneously.

In order to compare SofTEE with hardware-based TEE, we referenced the paper, called Komodo [17]. Komodo is considered as a hardware-based TEE because it is based on ARM TrustZone. Based on the performance of the Notary application given in Komodo [17], we are able to compare SofTEE and Komodo indirectly. In the case of Notary, the overhead of SofTEE is about 3% whereas the overhead of Komodo is almost 0% overhead. We think SofTEE incurs higher overhead than Komodo due to the limited performance of hardware TPM. There is much room to improve the performance in this aspect.

## VII. DISCUSSION

From a security perspective, protecting the trusted application is more complex than building a trusted kernel execution environment for the following reasons: 1) it should protect the confidentiality of the security monitor against the compromised kernel because the security monitor includes trusted application context; 2) it should ensure the confidentiality and integrity of the security monitor against trusted applications (TAs) launched by the malicious attacker. This is because malicious users may try to launch trusted applications in order to attack the security monitor; 3) it should guarantee the confidentiality and integrity of trusted applications against the compromised kernel; and 4) it should guarantee noninterference between TAs.

**TCB Size.** In SofTEE, TCB includes the binary scanner tool and the security monitor. Ultimately, all security monitor codes, including TPM software stack (TSS) and TPM device driver, should be trusted in the prototype. However, in this paper, the security analysis of the TPM is out of scope because it is too complex and enormous to deal with in this study. Instead, we measured the TCB size of the binary scanner and the core of the security monitor.

Our binary scanner and core of the security monitor have only 187 LOC and 3.7K LOC, respectively. Therefore, the TCB size of the main SofTEE is approximately 3.9K. This TCB size is reasonable compared to the TCB of previous studies [9], [18].

**Formal verification.** In a trusted execution environment (TEE) for user applications, security properties such as confidentiality and integrity of trusted application (TA) are essential. First, we have defined seven invariants that SofTEE should meet. Second, we have designed each component of SofTEE to satisfy the seven invariants. Our future works include how to verify each component of SofTEE meets such security invariants more formally by formal verification to improve the assurance of SofTEE correctness.

Based on TCB analysis, the security monitor in SofTEE is small enough for formal verification.

**Security Problems.** The attack surface of SofTEE is larger than the hardware-based TEEs. In particular, SofTEE may be more vulnerable to physical attacks, such as bus monitoring [38] and DMA attacks [39], [40]. However, we can apply some previous research to mitigate physical attacks. To address a bus monitoring attack, we can apply an idea discussed in previous research [44] to prevent this attack. The research attempted to use iRAM within the SoC to store cryptographic keys. Also, they encrypted privacy-sensitive data before the data were transferred.

DMA attacks can bypass kernel deprivileging. Therefore, we have to prevent the kernel from using DMA to access the security monitor memory. The solution for DMA attacks is straightforward in this model: 1) the monitor sets the system memory management unit (SMMU) [52] or IOMMU registers as read-only to the kernel using shadow page table (discussed in Section IV); 2) the security monitor sets I/O page table as read-only; and 3) the security monitor uses a shadow I/O page table to handle the read-only I/O page table.

In SofTEE, all CPUs can switch between the normal mode and the secure mode. In addition, the mode switching overhead is low due to the ASID management. Therefore, SofTEE uses CPU resources efficiently. However, CPU sharing between two modes can be exposed to attacks such as side channel attacks due to data cache sharing. To deal with cache side channel attacks, in a single-core machine, it is enough to simply clean and invalidate all data cache entries before returning to the normal mode. However, in a multicore environment, cleaning and invalidating cache entries is not enough because some cache side channel attacks [59], [60] are based on cross-core attacks via last level cache (LLC). The fundamental solution for preventing these attacks is to partition the LLC entries among cores. For example, some previous studies [41], [51] used hardware techniques for cache partitioning. On the other hand, SecTEE [56] proposed software techniques to partition the LLC entries. In SecTEE, the authors applied the page coloring technique [61], [62] to partition cache entries into some number of cache partitions, and used ARM cache locking mechanisms [57] to give exclusive control of a cache partition to an enclave.

It is worth noting that SofTEE does not expand the attack surface of hardware-based TEEs. For example, in an environment where ARM TrustZone and SofTEE coexist, SofTEE does not expand the attack surface of the vendor software running on ARM TrustZone.

**Performance.** SofTEE proposed the ASID management technique as an optimization to reduce the overall overhead. Nevertheless, the overhead of running trusted applications (TAs) is still high in real applications (especially when executing TAs with short computation time), limiting the practical applicability of SofTEE. We consider further optimizations for SofTEE to improve its practical applicability as future works.

For simple prototyping of cryptographic operations and attestation, SofTEE is using hardware TPM. In order to eliminate the hardware dependency on TPM, SofTEE may take advantage of software TPM such as fTPM [15], where most of the cryptographic functions are provided by the software. Due to the limited performance of hardware TPM, software-based cryptographic operations can further improve the performance of SofTEE. We consider software-based cryptographic operations in SofTEE to improve its practical applicability as future works.

**Comparison with Hardware-based TEEs.** There are many hardware-based TEE solutions [6], [9], [17], [50], [51], [56], [70], [72]-[75] available on ARM, Intel, and RISC-V processors. In particular, we compare SofTEE with ARM TrustZone which is widely used on mobile devices. SofTEE and ARM TrustZone require installing privileged monitor software in the isolated execution environment to manage trusted applications. The functionalities of the privileged monitor software are similar in both SofTEE and ARM TrustZone. However, SofTEE and ARM TrustZone are different in the following ways: 1) SofTEE supports the isolated execution environment through a software method called kernel deprivileging, whereas ARM TrustZone supports the isolated execution environment by an additional CPU state called secure world; and 2) SofTEE invokes the security monitor by function calls to the entry gate, whereas ARM TrustZone applies a hardware instruction called secure monitor call (SMC) to invoke the privileged monitor in the secure world.

Generally speaking, SofTEE has several advantages over hardware-based TEEs. First, SofTEE is more suitable to be applied in various machine environments because SofTEE does not have any dependency on special hardware features and higher privileges. Thus, applying SofTEE to various environments requires little effort. Second, it is important to fix any design flaws or deal with newly found vulnerabilities in TEE specifications as soon as possible. In SofTEE, faster (and cheaper) updates are possible. Third, it does not expand the attack surface of hardware-based TEE. Thus, SofTEE can be used together with hardware-based TEE on the same platform. For example, due to security reasons, most mobile device manufacturers strictly control the installation of security-sensitive applications on ARM TrustZone. In this environment, SofTEE can be used to provide TEE for security-sensitive applications, without affecting the security of ARM TrustZone.

**Cost of manual efforts.** We manually added an explicit call to the entry gate before some of the privileged operations, such as page table updates. Thus, we needed to modify the Linux source code. To prototype SofTEE in the Linux kernel, we totally modified 45 files and added 10 files. 8 out of 45 files were modified to support hardware TPM in the secure mode. We modified 708 LOC and added 4.8K LOC. 314 out of 708 LOC in Linux source code was modified to support hardware TPM in SofTEE.

**Hardware dependency.** SofTEE needs basic hardware supports such as root-of-trust (RoT) and random entropy. In addition, we assumed that SMEP (x86) or PXN (ARM) is enabled to prevent the kernel from injecting or reusing user code. SofTEE on SMEP or PXN may reduce the degree of hardware independence of SofTEE. However, because of the following reasons, we think the assumption of SMEP (x86) or PXN (ARM) does not violate our claim of SofTEE as software-based TEE. First, SMEP is considered general because AMD CPUs, as well as Intel CPUs, support SMEP. Thus, SMEP or PXN is a common technique regardless of architecture type. Second, SMEP and PXN are to be widely adopted in most x86 or ARM platforms.

## VIII. CONCLUSION

This paper presents a software-based TEE solution, SofTEE, which protects the security-sensitive part of an application from attackers, such as the compromised kernel. Using the kernel deprivileging, SofTEE delegates privileged operations such as memory-management-related operations from the kernel to a software module called a security monitor. For performance enhancement, SofTEE manages the address space identifier (ASID) so that the security monitor does not have to flush the TLB entries each time when switching the address spaces between the normal kernel and the security monitor.

We analyzed seven security invariants of SofTEE. Four invariants define the relationship between normal kernel-managed address space (called normal mode) and security monitor-managed address space (called secure mode), and the other three invariants define the relationship between software in the security monitor-managed address space (a.k.a. the secure mode).

In conclusion, SofTEE protects trusted applications from malicious attackers, such as the compromised kernel or security-sensitive user applications launched by malicious users. Unlike previous TEE solutions for user applications, SofTEE does not rely on special hardware technologies, such as Intel SGX and ARM TrustZone. Thus, it is applicable to any machine environment. Note that SofTEE needs basic hardware supports such as root-of-trust (RoT) and random entropy. This assumption is considered reasonable because these hardware requirements are fundamental for trusted computing.

## REFERENCE

[1] Samsung Developers. Device-side Security: Samsung Pay, TrustZone, and the TEE. [Online]. Available: https://developer.samsung.com/tech-insights/pay/device-side-security.

[2] SECURE TECHNOLOGY ALLIANCE. Trusted Execution Environment (TEE) 101: A Primer.

[Online]. Available:

https://www.securetechalliance.org/wpcontent/uploads/TEE-101-White-Paper-FINAL2-April-2018.pdf.

[3] WiDEVINE DRM Architecture Overview.

[Online]. Available:

https://storage.googleapis.com/wvdocs/Widevine_DRM_Architecture_Overview.pdf.

[4] HYPR. [Online]. Available: https://www.hypr.com/

[5] ARM Ltd. Security technology building a secure system using TrustZone technology (white paper). 2009.

[Online]. Available:

http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC 009492C_trustzone_security_whitepaper.pdf.

[6] Software Guard Extensions Programming Reference. Intel Corp., Oct. 2014. Ref. #329298-002. [Online]. Available: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[7] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution," in 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Cambridge, UK, 2014, pp. 190-202.

[8] Raspberry Pi 3 board reference. [Online]. Available:

https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[9] 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 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys), Niagara Falls, NY, USA, 2017, pp. 488-501.

[10] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, TX, USA, 2013, pp. 265-278.

[11] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Seattle, WA, USA, 2008, pp. 71-80.

[12] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Port, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS), Seattle, WA, USA, 2008, pp. 2-13.

[13] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P. Yew, and W. Mao, "Tamper-resistant execution in an untrusted operating system using a virtual machine monitor," Technical Report FDUPPITR-2007-0801, Parallel Processing Institute, 2007.

[14] 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 Proceedings of the USENIX Annual Technical Conference (ATC), Denver, CO, USA, 2016, pp. 565-578.

[15] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten, "fTPM: A software-only implementation of a TPM chip," in 25th USENIX Conference on Security Symposium (USENIX Security), Austin, TX, USA, 2016, pp. 841-856.

[16] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Istanbul, Turkey, 2015, pp. 191-206.

[17] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure enclave hardware from software," in Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), Shanghai, China, 2017, pp. 287-305.

[18] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual Ghost: Protecting Applications from Hostile Operating Systems," in Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Salt Lake City, Utah, USA, 2014, pp. 81-96.

[19] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility safety and performance in the SPIN operating system," in Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, USA, 1995, pp. 267-283.

[20] J. Liedtke, "On micro-kernel construction," In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, CO, USA, 1995, pp. 237-250.

[21] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development," in Proceedings of the USENIX Conference, 1986.

[22] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W. C. Hsu, and Y. Cheng, "RERANZ: A Light-Weight Virtual Machine to Mitigate Memory Disclosure Attacks," in Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Xi'an, China, 2017, pp. 143-156.

[23] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely Rerandomization for Mitigating Memory Disclosures," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), Denver, CO, USA, 2015, pp. 268-279.

[24] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY), New Orleans, Louisiana, USA, 2016, pp. 50-61.

[25] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization," in Proceedings of the 21st USENIX Conference on Security Symposium (USENIX Security), Bellevue, WA, USA, 2012, pp. 40-40.

[26] K.Lu, S. Nurnberger, M. Backes, and W. Lee, "How to Make ASLR Win the Clone Wars: Runtime Re-Randomization," in 23rd Annual Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 2016.

[27] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J.Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, 2016, pp. 367-382.

[28] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity," in 23rd Annual Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 2016.

[29] Y. Cho, D. Kwon, and Y. Paek, "Instruction-Level Data Isolation for the Kernel on ARM," in 54th Design Automation Conference (DAC), Austin, TX, USA, 2017.

[30] S. BANESCU, "Cache timing attacks," 2011.

[31] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill, "Ironclad apps: End-to-end security via automated fullsystem verification," in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA, 2014, pp. 165-181.

[32] L. Deng, P. Liu, J. Xu, P. Chen, and Q. Zeng, "Dancing with Wolves: Towards Practical Event-driven VMM Monitoring," in Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Xi'an, China, 2017, pp. 83-96.

[33] Z. Wang, J. Li, C. Wu, D. Yang, Z. Wang, W. Hsu, B. Li, and Y. Guan, "HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines," in Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Istanbul, Turkey, 2015, pp. 53-64.

[34] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM," In 23rd Annual Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 2016.

[35] D. Ray and J. Ligatti, "Defining code-injection attacks," in Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL), Philadelphia, PA, USA, 2012, pp. 179-190.

[36] S. Zhao and X. Ding, "On the effectiveness of virtualization-based memory isolation on multicore platforms," in 2nd IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France, 2017, pp. 546-560.

[37] MITRE. CVE-2017-5691, July 2017. [Online]. Available: https://nvd.nist.gov/vuln/ detail/CVE-2017-5691.

[38] G. Gogniat, T. Wolf, W. Burleson, J.-P. Diguet, L. Bossuet, and R. Vaslin, "Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective," Very Large Scale Integration (VLSI) Systems, vol. 16, no. 2, 2008, pp. 144-154.

[39] D. Aumaitre and C. Devine, "Subverting Windows 7 x64 Kernel with DMA attacks," in Hack In The Box Security Conference (HITB), 2010. [Online]. Available: http://esec-lab.sogeti.com/static/publications/10-hitbamsterdam-dmaattacks.pdf.

[40] P. Stewin, and I. Bystrov, "Understanding DMA Malware," in Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Heraklion, Crete, Greece, 2012, pp. 21-41.

[41] X. Dong, Z. Shen, J. Criswell, Alan L. Cox, and S. Dwarkadas, "Shielding Software From Privileged Side-Channel Attacks," in 27th USENIX Security Symposium (USENIX Security), Baltimore, MD, USA, 2018, pp. 1441-1458.

[42] ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition. [Online] Available:

https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf.

[43] Linux TPM2 & TSS Software. [Online]. Available: https://github.com/tpm2-software.

[44] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. D. Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Istanbul, Turkey, 2015, pp. 177-189.

[45] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC), San Diego, CA, USA, 1996, pp. 23-23.

[46] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARM TrustZone," in 26th USENIX Conference on Security Symposium (USENIX Security), Vancouver, BC, Canada, 2017, pp. 541-556.

[47] B. Lapid and A. Wool, "Cache-Attacks on the ARM TrustZone implementations of AES-256 and AES-256-GCM via GPU-based analysis," Selected Areas in Cryptography (SAC), Calgary, AB, Canada, 2018.

[48] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in Proceedings of the USENIX Winter 1993 Conference, San Diego, CA, USA, 1993, pp. 2-2.

[49] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the Memory Secrecy Assumption," in Proceedings of the 2nd European Workshop on System Security, (EUROSEC), Nuremburg, Germany, 2009, pp. 1-8.

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

[51] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in 25th USENIX Conference on Security Symposium (USENIX Security), Austin, TX, USA, 2016, pp. 857-874.

[52] ARM System Memory Management Unit. Architecture Specification, ARM Ltd, 2012.

[53] F. Brasser, D. Gens, P. Jauernig, A. R. Sadeghi, and E. Stapf, "Sanctuary: ARMing TrustZone with User-space Enclaves," in 26th Annual Symposium on Network and Distributed System Security (NDSS), 2019.

[54] M. Schwarz, M. Lipp, A. Moghimi, J. V. Bulck. J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK, 2019, pp. 753-768.

[55] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. V. Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK, 2019, pp. 769-784.

[56] S. Zhao, Q. Zhang, Q. Yu, W. Feng, and D. Feng, "SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE," in Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS), London, UK, 2019, pp. 1723-1740.

[57] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "CacheKit: Evading Memory Introspection Using Cache Incoherence," in 1st IEEE European Symposium on Security and Privacy (EuroS&P), Saarbrücken, Germany, 2016, pp. 337-352.

[58] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing Page Faults from Telling Your Secrets," in 11th ACM on Asia Conference on Computer and Communications Security (Asia CCS), Xi'an, China, 2016, pp. 317-328.

[59] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in 25th USENIX Security Symposium (USENIX Security), Austin, TX, USA, 2016, pp. 549-564.

[60] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in 36th IEEE Symposium on Security and Privacy (S&P), San Jose, CA, USA, 2015, pp. 605-622.

[61] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud," in 21st USENIX Security Symposium (USENIX Security), Bellevue, WA, USA, 2012, pp. 189-204.

[62] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource Management for Isolation Enhanced Cloud Services," in 1st ACM workshop on Cloud computing security (CCSW), Chicago, Illinois, USA, 2009, pp. 77-84.

[63] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kGuard: Lightweight Kernel Protection against Returnto-user Attacks," in 21st USENIX Security Symposium (USENIX Security), Bellevue, WA, USA, 2012, pp. 459-474.

[64] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking Kernel Isolation," in 23rd USENIX Security Symposium (USENIX Security), San Diego, CA, USA, 2014, pp. 957-972.

[65] V. George, T. Piazza, and H. Jiang, "Technology Insight: Intel©Next Generation Microarchitecture Codename Ivy Bridge," September 2011. [Online]. Available: http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf.

[66] Intel® 64 and IA-32 Architectures Software Developer's Manual. Instruction Set Extensions Programming Reference. Tech. rep., Intel Corporation, January 2013.

[67] ARM® Architecture Reference Manual. ARM® v7-A and ARM®v7-R edition. Tech. rep., Advanced RISC Machine (ARM), July 2012.

[68] ARM® Architecture Reference Manual Supplement. ARM® v8.1, for ARM®v8-A architecture profile, June 2016.

[69] Y. Cho, D. Kwon, H. Yi, and Y. Paek, "Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM," in 24th Annual Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 2017.

[70] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: an open framework for architecting trusted execution environments," in Proceedings of the 15th European Conference on Computer Systems (Eurosys), Heraklion, Greece, 2020.

[71] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in Proceedings of the 18th USENIX Security Symposium (USENIX Security), Montreal, Canada, 2009.

[72] C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in Proceedings of the USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, 2017, pp. 645-658.

[73] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA, 2014, pp. 267-283.

[74] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, 2016, pp. 689-703.

[75] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, A. Sadeghi, "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V," in 26th Annual Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA, 2019.