IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 9, 2020, accepted December 23, 2020, date of publication December 28, 2020, date of current version January 8, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3047813

SOTPM: Software One-Time Programmable Memory to Protect Shared Memory on ARM Trustzone

DONGWOOK SHIM AND DONG HOON LEE, (Member, IEEE)
Graduate School of Information Security, Korea University, Seoul 02841, South Korea
Corresponding author: Dong Hoon Lee (donghlee@korea.ac.kr)

This work was supported by the Korea University Grant.

ABSTRACT In ARM TrustZone-based architecture, shared memory is one of the most useful schemes to enable isolated execution environments supported by TrustZone to communicate between environments. However, it is already known that shared memory is vulnerable to man-in-the-middle attacks since mechanisms to check integrity or authenticate callers for the shared memory payload are not supported in TrustZone. While an encryption-based method that resolves this limitation does exist, there are some architectural limitations. Indeed, even with key protection countermeasures applied, there is a risk that encryption keys may be leaked, as they are placed in insecure user memory during communication. Moreover, countermeasures for key leakage cause system performance overhead. In this paper, we propose a lightweight and secure scheme for shared memory, called *Software One-Time Programmable Memory (SOTPM)*. SOTPM is a software-implemented, one-time programmable shared memory. It is based on the idea that payload encryption in the shared memory layer is unnecessary because sensitive data is already encrypted in the application layer before being written to the shared memory. SOTPM is set to read-only after data is written into SOTPM due to the one-time programmable characteristic. Therefore, attackers are unable to manipulate content in SOTPM during communication. Since it is not necessary for SOTPM to encrypt the payload in order to prevent malicious payload manipulation, it is possible to remove the risk of key leakage posed in previous studies. Additionally, in contrast with the existing method, our method can dramatically reduce system performance overhead. We implemented our prototype on an open-source hardware board with an Armv8-A processor and performed a security analysis and performance evaluation. The results show that SOTPM provides a sufficient level of security and less than 1% performance overhead, implying that SOTPM is a reasonable solution for current commercial products.

INDEX TERMS ARM TrustZone, secure communication, shared memory.

I. INTRODUCTION

ARM TrustZone [1], a security hardware extension for ARM architecture, has been widely adopted for use in various smart or IoT devices. This technology divides an execution environment into two isolated domains: a rich execution environment (REE) for hosting a rich OS and a trusted execution environment (TEE) for a secure OS. Isolated from the REE, the TEE executes security-critical services such as payment, banking, authentication, and signing without disclosing sensitive data to the REE. Since the two domains are isolated, a communication scheme is necessary to send large volumes of data from one domain to the other. The shared memory is one of the most useful schemes to enable the isolated domains to communicate with each other on TrustZone-based architecture, owing to its performance and usability. Many studies have widely adopted a shared memory scheme on TrustZone-based TEE to provide trusted computing features like security-critical services, end-to-end secure communication, and TEE virtualization.

TrustZone-related studies assume that TEE is secure and an attacker is unable to infiltrate a TEE. However, these assumptions can be dismantled since an attacker can infiltrate a TEE by exploiting weaknesses of the shared memory.

The associate editor coordinating the review of this manuscript and approving it for publication was Shadi Aljawarneh.

Indeed, attackers frequently exploit the various weaknesses of the shared memory scheme to infiltrate TEE, and there are already many reports on a variety of common vulnerabilities and exposures (CVE) (e.g. CVE2013-3051, CVE2015-4421, CVE2015-6639, CVE2015-6647, CVE2016-0825, and CVE2016-2431). Among the critical weaknesses of the shared memory scheme, a noteworthy point is that the ARM TrustZone does not support integrity-checking mechanisms and caller authentication for shared memory as related to communication between the REE and the TEE. In other words, an invoked trusted application (TA) in TEE cannot know whether or not the content in the shared memory has been manipulated or which client application (CA) in an REE invokes the corresponding TA. In more detail, an attacker can analyze the message structure of an objective TA with several methods, including hijacking the TEE client library or TEE driver, using a custom kernel, or reverse engineering, given that each TA may have own message structure for communication. After this analysis, the attacker creates an arbitrary process or hijacks the TEE client library or TEE driver in order to send messages. The attacker then sends falsified messages to the objective TA to locate vulnerabilities in the objective TA. If the attacker succeeds in finding effective vulnerabilities after analyzing the responses for falsified messages, they can invoke a TA with messages exploiting identified vulnerabilities and gain access to sensitive data in the TEE or manipulate TEE functions.

SeCReT [2] is the first proposed framework that aimed to protect shared memory from these vulnerabilities, and it introduced symmetric encryption for payloads in the shared memory and active process contexts for session key management. However, SeCReT has two problems: a security risk due to encryption key leakage and system performance overhead. Since a symmetric key for encryption is located in the user memory space of an REE, it is possible that the key may be leaked by attacks out of the system protection boundary, such as a zero-day attack. In addition to this security risk, the key protection countermeasures that are added to most exception handlers and the hash verification on every step cause performance overhead on the exception handler. SeCReT optimization [3] largely mitigates these two problems, but there is still the possibility of key leakage via various attacks, since the REE is an insecure execution environment.

Through analyzing payloads, we discovered that a payload at the shared memory layer is not necessarily handled as confidential data. Considering generic-use cases of TEE services such as payments, all sensitive data is first encrypted at the application layer in external service provider and TEE, and then placed in the shared memory. Additionally, the payload structure can be readily disclosed through reverse engineering or API documents. These characteristics are similar to payload wrapping on the general network communication model, and given this fact, we know that one main objective of channel encryption in previous research has prioritized checking communication channel integrity over confidentiality. Based on our discovery, we suggest a lightweight and secure scheme that is different from existing technologies, called *Software One-Time Programmable Memory (SOTPM)*. A one-time programmable (OTP) memory is a special type of non-volatile memory (NVM) that permits data to be written only once. SOTPM implements this characteristic of OTP memory at the software level through precise permission control of the Memory Management Unit (MMU) for pages of physical memory.

SOTPM has two distinct features: channel protection and caller process authentication. Channel protection switches the SOTPM property between read-writable and read-only. SOTPM is initially set to read-writable like random access memory, but after data is written into SOTPM, the property is switched to read-only to ensure data integrity. This switching procedure is implemented in software and called *activation*. We also introduce *Triggering memory (TM)*, which acts as a toggle switch that can perform activation securely without kernel interference. Caller process authentication, the second feature of SOTPM, checks whether or not the invoking CA is allowed to send messages to the TA by verifying the cryptographic hash value of the caller process. Specifically, hash verification is performed against a static region of the caller process by 4KB granularity during both registration and activation phases. Authentication is successful if the compile time-generated hash value matches the one generated at runtime. As more operations within the exception handler have a more significant effect on system performance, SOTPM is designed to minimize operations related to SOTPM within the exception handler.

We implemented the prototype of SOTPM on an open-source hardware board based on Armv8-A and performed a security analysis and performance evaluation. In our model, we assume that a secure boot [4] and kernel integrity monitor [5]–[7] have already been applied to protect the static region of the REE kernel. Our results show that SOTPM is sufficiently secure compared with previously proposed technologies and produces less than 1% performance overhead.

In summary, the main contributions of this paper are:

- We introduce SOTPM, an innovative, yet lightweight and secure shared memory scheme designed to protect the communication channel between the REE and the TEE. SOTPM provides a level of security that is comparable to previous works, while removing the risk of key leakage. Moreover, SOTPM scheme prevents malicious attempts to communicate with the TA for analysis purposes.
- SOTPM may be easily integrated into commercial products thanks to its lower implementation burden and low performance overhead of less than 1%. In addition, SOTPM does not interfere with any other security processes in the system by minimizing the involvement to exceptions within exception handlers.
- SOTPM can be extended to a cheaper A-series ARM architecture-based system, such as small sensors or IoT

devices. Since SOTPM only utilizes TrustZone and MMU, additional hardware extensions are not required.

The remainder of this paper is organized as follows: Section II introduces the required background information. Section III describes our assumptions and attack model. Section IV presents the design of SOTPM in detail. Section V describes the implementation of SOTPM. Section VI details our security analysis and performance evaluation. Section VII discusses the limitations of SOTPM. Section VIII presents related work. Section IX concludes this paper.

II. BACKGROUND

In this section, we provide background information on ARM TrustZone and Armv8-A AArch64 page descriptor to aid with understanding of our model. We briefly introduce the TrustZone-based kernel integrity monitor, since we assume in our model that a kernel integrity monitor is already applied to protect the kernel static region in an REE.

Note that the Exception levels first introduced with the Armv8 architecture replaces execution modes of Armv7 and is represented as "ELx," where x is a digit that describes the privilege level from 0 to 3. EL0 is the lowest level of privilege, where applications are executed, while OS kernel runs in EL1, hypervisor in EL2, and secure monitor in EL3, which is the highest level of privilege.

A. ARM TrustZone

ARM TrustZone [1], [8] is a security hardware extension on ARM architecture that divides an execution environment logically into an REE and TEE. The Non-secure (NS) bit of the Secure Configuration Register (SCR_EL3) [9] indicates whether the access is Secure or Non-secure, and it is added to all memory system transactions including cache tags and access to system memory and peripherals. TrustZone Address Space Controller (TZASC), TrustZone Memory Adapter (TZMA), and TrustZone Protection Controller (TZPC) are hardware libraries that control access to DRAM, SRAM, and peripherals. These libraries ensure that each TEE asset can be accessed from the REE by checking the NS bit. As a result, ARM TrustZone effectively provides a physical address space for the TEE and a completely separate physical address space for the REE.

By invoking a trusted application (TA) in the TEE, a client application (CA) in the REE first opens the TEE device of kernel, and then requests shared memory allocation to the kernel. After shared memory is allocated, the CA writes payloads for command ID, length, and other data into the shared memory, and subsequently requests the kernel to invoke the Secure Monitor Call (SMC) instruction. When the REE kernel invokes the SMC instruction upon request, the Exception level changes to EL3. EL3 is the gatekeeper of each execution environment, and it saves and restores the context of each domain in terms of every context switch between the REE and the TEE. On invoking a TA, EL3 invokes the TEE kernel after saving the REE context and restoring the TEE context. The TEE kernel thus refers to payloads in the shared memory and invokes the dedicated TA with arguments from payloads. The TA performs the requested operation and returns after writing the result into the shared memory. Upon returning a result to CA, invoking procedures are executed in reverse. Figure 1 illustrates the generic communication scheme of ARM TrustZone.

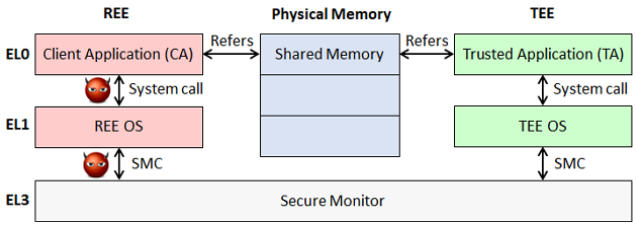

FIGURE 1. Generic ARM TrustZone communication scheme: REE EL0 and TEE EL0 pass payloads and results by referring to a shared memory. A man-in-the-middle attack is possible in an REE at the points indicated.

B. Armv8-A AArch64 MMU AND DESCRIPTOR

Armv8-A architecture supports Virtual Memory System Architecture (VMSA) [10], which uses multi-level paging in order to manage page tables efficiently. Each process has its own virtual address space and can access the physical memory through address translation. Address translation is the procedure in which a virtual address (VA) is translated to a physical address (PA) by referring to the page table on each level in multi-level paging [11]. The address of the page table (a.k.a. table base address), the start address to be translated, is held in the Translation Table Base Registers (TTBR0_EL1, TTBR1_EL1). TTBR0_EL1 is for applications in EL0, and it is selected when the upper bits of the VA are all set to 0. On the other hand, TTBR1_EL1 for kernel space is selected when the upper bits of the VA are all set to 1. EL2 and EL3 each have a TTBR0, but no TTBR1.

A Memory Management Unit (MMU) is dedicated hardware that translates a VA to a PA automatically in order to support VMSA. Note that any memory access is only possible via MMU if VMSA is supported and MMU is enabled. In addition to address translation, MMU decides whether or not to allow access to the physical address of the next level by referring to attributes of a descriptor on each level in multi-level paging. Next, there are two AArch64 descriptor types: table or block. A table descriptor contains the physical address of the next level page table, while a block descriptor contains the physical address of the actual block or page in the memory. Each descriptor has entry attributes, such as execution permission, access permission, and shareable and non-secure access. The MMU verifies that the access request is equivalent to the descriptor attributes, then executes address translation continuously or makes translation faults.

C. TrustZone-BASED KERNEL INTEGRITY MONITOR

The TrustZone-based kernel integrity monitor [5]–[7] resides in the TEE as a service that is aimed at protecting the static

region of the kernel in the REE. The main features of a kernel integrity monitor are kernel de-privileging, emulation of system control instructions, and trapping translation table updates. The kernel integrity monitor removes privilege from the kernel by configuring the properties of kernel static regions and page tables as read-only. A de-privileged kernel is unable to alter any content or property of a specified memory region, as this privilege is reserved for the kernel integrity monitor. In order to prevent any attempt at disabling the MMU or manipulating the TTBR register to force the MMU to refer to other falsified page tables, the kernel integrity monitor emulates all system control instructions. An emulation is implemented by replacing system control instructions, such as MRS and MSR, to SMC instruction. Since all system instructions are verified by the kernel integrity monitor through instruction emulation, an attacker is unable to manipulate an MMU into referring to falsified page tables. In addition, all page table updates are trapped within the kernel integrity monitor, such that only verified modifications are updated. Since trapping translation table updates are non-bypassable due to the fixed exception vectors, it is impossible for attackers to remap arbitrary page tables, double map kernel data, or manipulate memory properties. Moreover, the execution of malicious code with privilege is prevented by setting a Privileged eXecute Never (PXN) bit for all user-space memory besides the code section. By combining these features, the kernel integrity monitor ensures integrity of the kernel static region in the REE.

III. ASSUMPTIONS AND ATTACK MODEL

A. WEAKNESSES OF SHARED MEMORY SCHEME IN TrustZone

Shared memory can be allocated from free memory in user-space or an additionally allocated memory mapped from the kernel. The address of the shared memory should be aligned since that address is re-mapped to the address space of EL3 or TEE EL1. In order to obtain an easily aligned address, allocation through the kernel is mainly used, since the kernel manages memory as aligned page granularity.

The invocation procedure involving the shared memory scheme has two weaknesses. The first is that it is difficult to know whether an SMC invocation has been executed by an authenticated CA in a TA or TEE OS because anyone with privileges can invoke SMC instruction since there is no caller tracing mechanism in TrustZone. Due to this weakness, an attacker with kernel privileges can create an arbitrary process and send falsified messages with crafted parameters. The other is that a TA or TEE OS cannot know whether the shared memory address or content has been manipulated. An attacker with kernel privileges can hijack either an address or data in the shared memory via the TEE library or TEE driver. Moreover, the attacker can directly map the shared memory region to the kernel address space and therefore manipulate it. Figure 1 illustrates the parts vulnerable to attack in a communication scheme.

B. TRUSTED COMPUTING FEATURES

In our model, we assume that the secure boot and kernel integrity monitor, which are commercially available solutions, has already been applied. Secure boot [4] is a method that verifies the integrity and authenticity of a next boot stage image before handing over execution upon boot. Since the root of trust (RoT) should exist in read-only memory (ROM) to verify image authenticity, we also assume that RoT has already been inserted during factory processing and saved in ROM. In addition to secure boot, almost all platforms, including Android, iOS, and Tizen, ensure an application has not been compromised on initial load through verification of integrity and authenticity via signature. In other words, only a verified OS and legitimate applications in the REE and TEE are loadable through the secure boot. The static region of the REE kernel and page tables are protected on run-time by the kernel integrity monitor. Therefore, the kernel static region is always protected by secure boot and the kernel integrity monitor. Our assumptions for trusted computing features are realistic given that all mentioned solutions have been actually applied to commercial products such as Samsung Galaxy devices powered by KNOX [12].

C. ATTACK MODEL

In our model, we assume a realistic attacker and commercial device. Generally, the manufacturers of smart devices (Samsung Galaxy devices, Google Nexus, so on) release commercial products after disabling hardware-based debug features like JTAG, breakpoints, and watchpoints for security. Disabling hardware-based debug features makes it difficult to perform a single step execution of an application or direct manipulation against the application memory in commercial products. However, it does not mean that it is impossible to control-flow hijacking against the application or direct manipulation of the application memory. We discuss these limitations in section VII. Considering commercial device environments, we assume that an attacker can't hijack the control flow of an application by utilizing hardware-based debug features in our model.

We determine that the goal of an attack is to access, disclose, and misuse sensitive data like signing key, bio-metrics data, or encrypted contents in the TEE. To accomplish the goal, an active attacker is able to use debug bridge utilities supported a shell like adb (Android Debug Bridge), and they may have REE kernel privileges with exploiting kernel vulnerabilities in run-time. We consider the attacker with REE kernel privileges who can accomplish everything except manipulating the kernel static region or page tables that are protected by the secure boot and kernel integrity monitor or utilizing hardware-based debug features. The attacker is also able to bypass both authentication mechanisms that check the Universal Unique IDentifier (UUID) of the requester and access control mechanisms such as Security-Enhanced Linux (SELinux). These are realistic assumptions since there are bypassing cases [13], [14].

Direct access to TEE resources by the attacker can be prevented because ARM TrustZone protects the resources via TZASC, TZMA, and TZPC. Hence, the attacker tries to access TEE resources indirectly by exploiting the weaknesses of the shared memory scheme in TrustZone [15], [16]. In this scenario, the attacker with REE kernel privilege can passively observe payloads on shared memory via the kernel page table. The attacker analyzes the message structure of an objective TA by observation, hijacking the TEE libraries, using a custom kernel, or reverse engineering. The attacker can also observe data in shared memory via kernel page tables. After analysis, the attacker may create a malicious process that continuously invokes the SMC instructions with various crafted parameters to uncover TEE service or application vulnerabilities or access sensitive data in the TEE. Moreover, the attacker may manipulate the shared memory by double mapping to kernel space, or alternatively, sending falsified messages or shared memory addresses after hijacking the TEE library or TEE driver. If the attacker succeeds in finding effective vulnerabilities after analyzing the responses for falsified messages, they can invoke a TA with messages exploiting identified vulnerabilities and gain access to sensitive data in the TEE or manipulate the TEE function.

The MMU is at the core of our scheme, and therefore, we assume that the system supports VMSA (and has enabled MMU on boot. Additionally, our model rests on the assumption that the REE processes permitted to access the TEE and hash values of their corresponding static regions are predefined as a list, which is maintained in EL3. The list is created in compile-time and it may be included in the EL3 image on build-time or may be delivered to EL3 on boot with signature. We exclude Denial-of-Service (DoS) attacks aiming to disrupt TEE services as these are not in line with the goal of an attack that we present in this paper. Finally, we do not consider Direct Memory Access (DMA) attacks in this paper, as we assume that the Input-Output Memory Management Unit (IOMMU) is secure. Note that side-channel attacks are outside the scope of this paper since they are beyond the coverage of ARM TrustZone protection.

IV. SOTPM DESIGN

A. OVERVIEW

In previous works, there was a remaining issue involving the risk of encryption key leakage on communication and performance overhead caused by key protection countermeasures. In response to this, we specifically designed our model to match or surpass the security level of previous models and minimize performance overhead.

First, we considered the most efficient method for channel protection. Channel encryption is a satisfactory choice to preserve confidentiality and integrity simultaneously, but it also has security risk of encryption key leakage. In order to minimize this security risk, we analyzed payloads and discovered that payload is not necessarily handled as confidential data. Looking at end-to-end communication for TEE services like payment or banking, all sensitive data such as session key or payment transactions are encrypted at the application layer in the external service provider and TEE, then written into the shared memory. A CA is just worked as a bridge on communication between an external service provider and a TA, so that encryption is not necessary in the CA. Meanwhile, the payload structure and its values such as command ID can be easily obtained from the binary codes of an application, library, or API documents. These payload properties are similar to payload wrapping on a generic network communication model. Given that payload property, we designed a channel protection process that minimizes overhead through precise permission control of the Memory Management Unit (MMU).

Next, we considered the caller process authentication. It is a difficult problem to define the scope of a target process for authentication. Generally, caller process authentication represents whether or not process integrity, which is classified into static or dynamic integrity, has been preserved on verification time. Static integrity verifies values of pre-defined sections on compile-time and mainly targets a code section, while dynamic integrity mainly verifies Control Flow Integrity (CFI) or variables on run-time. Dynamic integrity has the advantage of detecting advanced attacks such as return-oriented programming (ROP) or code injection attacks, but it also results in partial performance loss and requires more storage. Unfortunately, given the other supported trusted computing features in our system, we deemed it too expensive to apply simultaneous control flow tracing along with static region verification. As a result, we ultimately decided to check static integrity only for read-only segments of a targeted application, including the program header and codes, in order to enhance performance. Of course, this decision pertains specifically to our system, and the integrity scope can certainly be adjusted in view of security requirements for a dedicated embedded system.

Figure 2 shows a conceptual overview of our proposal, called *Software One-Time Programmable Memory (SOTPM)*. SOTPM consists of two features and four phases: the two features encompass channel protection and caller process authentication, and the four phases are registration, activation, invocation, and deregistration. We describe these features and phases in detail in the next section.

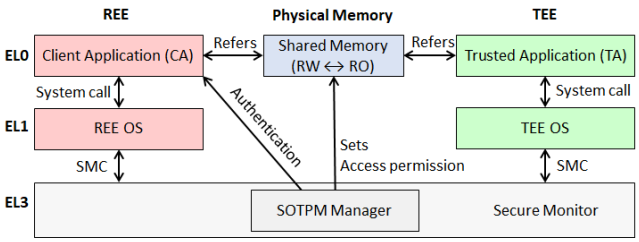

FIGURE 2. SOTPM conceptual overview: SOTPM manager in EL3 authenticates CA and sets the access permission for the shared memory to read-writable or read-only.

B. CHANNEL PROTECTION

An OTP memory property is automatically switched to read-only at the hardware level when data is written once. Unlike OTP memory, the SOTPM property switches to read-only when MMU alters the access permissions for page descriptors of the shared memory at the software level. This switching procedure is called *activation*. As shown in Figure 3, the AArch64 descriptor contains a physical address as well as upper and lower attributes. Access permissions for a memory page are managed at low attributes of blocks or page descriptors, which can be separately configured for unprivileged level and privileged level. In detail, access permission (AP) bits consist of two bits, which are placed at bits 6 and 7 of the descriptor lower address. Bit 7 indicates access permissions for privileged levels (EL1, EL2, EL3), which is represented as read-writable (0) or read-only (1). Bit 6 indicates access permissions for the unprivileged level (EL0), which is represented as no-access (0) or the value of privileged level (1) by the AP upper bit. When the value is 1, it means either read-writable or read-only. Configurable values of access permissions are summarized in Table 1. The referring access permission level is determined by the exception level on accessing to the memory page. For example, access permission for the unprivileged level is referred if page access is done by TTBR0_EL0. Channel protection in SOTPM is set by activation that alters access permission bits for page descriptors to 11, which indicates that all access to SOTPM at all exception levels is only permitted to read-only after activation. This means that an attacker is not able to indiscriminately manipulate data at all exception levels.

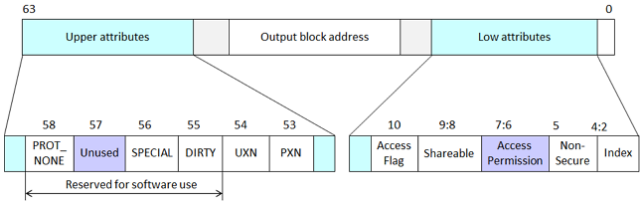

FIGURE 3. ARMv8 AArch64 page descriptor format: the descriptor has upper and lower attributes. An unused bit in upper attributes is used for caller process authentication, while access permission bits are used for channel protection.

Thus, activation is critical to the implementation of channel protection. Access permissions for the page descriptor are controlled by the kernel integrity monitor and the SOTPM manager in EL3. After a CA writes data into SOTPM, the CA should securely send an activation request to the SOTPM manager, prompting the activation of SOTPM. However, the activation request can be manipulated in EL1 by an attacker with kernel privileges. This is because all requests from EL0 to EL3 pass through EL1 since Exception levels increase stepwise. In order to prevent the request from attacker interference or manipulation in REE EL1 during activation, we introduce *Triggering Memory (TM)* to function as a switch that securely sends an activation request to EL3 in the form of a data abort exception.

TABLE 1. Configurable values of access permissions in page descriptors.

AP[7:6]	Unprivileged (EL0)	Privileged (EL1/2/3)
00	No-access	Read-writable
01	Read-writable	Read-writable
10	No-access	Read-only
11	Read-only	Read-only

The SOTPM property is switched to read-only when a CA writes data into SOTPM and touches (reads) the TM. In detail, the TM is additionally allocated during the registration phase, and its access permission is set to "no access" (10) in EL0. If a CA reads the TM, a reading permission fault occurs due to the EL0 access permission for TM. A reading permission fault is a type of data abort that is produced during address translation. A data abort handler is placed in the exception vector table of the REE EL1 static region, which contains codes that invoke an SMC instruction immediately for a reading permission fault. After the Exception level is changed to EL3 by SMC instruction, the SOTPM manager in EL3 is invoked. The SOTPM manager changes access permissions for TM and the shared memory for EL0 and EL1 to read-only after verifying data abort related registers. The reason that the access permission for EL1 changes additionally is because the shared memory can be double mapped in EL1 since EL1 has its own virtual memory space. As a result, both REE EL0 and REE EL1 are unable to manipulate shared memory values after activation. Each step of the changes in access permissions for the shared memory and TM is outlined in Table 2. Note that the reading permission fault for the TM, caused by no-access, only occurs in the unprivileged level (EL0), but it does not occur in any privileged level. Therefore, an attacker is neither able to activate SOTPM directly at the kernel level, even if they had kernel privileges, nor manipulate the exception vector table protected via the kernel integrity monitor. The attacker may attempt to activate SOTPM at the user level by creating an arbitrary process, but that attack can be prevented by applying caller process authentication.

TABLE 2. Access permission values for each memory region by procedure.

Memory	Access from	Initialization	Registration	Activation
Shared Memory	EL0	Read-writable (01)	Read-writable (01)	Read-only (11)
	EL1	Read-writable (00)	Read-writable (00)	Read-only (11)
Triggering Memory	EL0	Read-writable (01)	No-access (10)	Read-only (11)
	EL1	Don't care	Don't care	Don't care

In order to reduce SMC invocations caused by data abort exceptions, the data abort handler only invokes an SMC instruction if a level 3 reading permission fault occurs. Level 3 indicates the third level of address translation. Moreover, the SOTPM manager is placed in front of the TEE entry point as a gatekeeper, to avoid context switching. Through

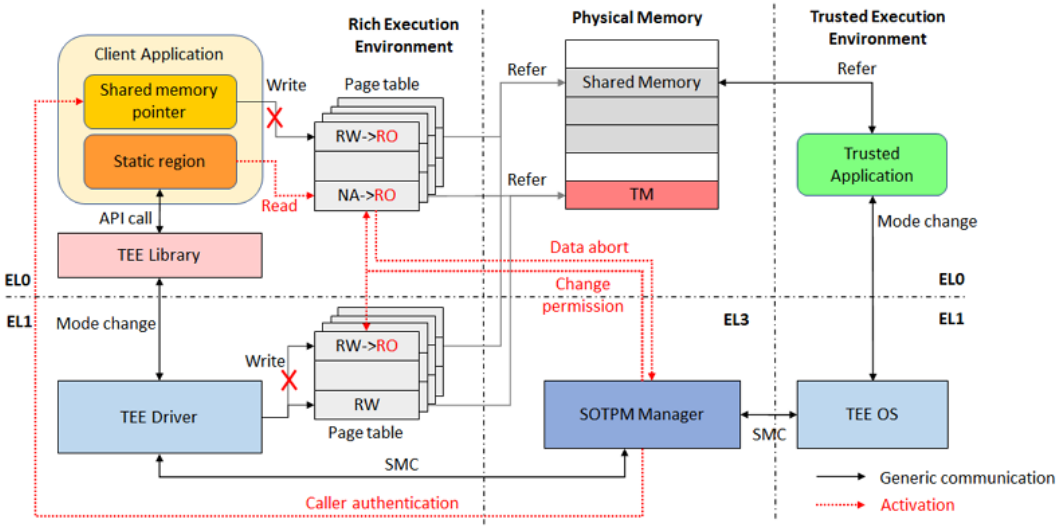

FIGURE 4. SOTPM procedure overview: SOTPM is made up of four phases. Here, generic communication and activation flows are shown. The flows of the registration, invocation, and deregistration are the same as the generic communication flow. SOTPM is activated via a data abort against the Triggering memory. Writing data into SOTPM is prohibited after the access permissions of SOTPM in EL0 and EL1 are changed from read-writable to read-only.

minimizing SMC invocation and context switching, performance overhead can be minimized for channel protection. In addition, the integrity of the return value from TEE is guaranteed up to SOTPM deregistration, since REE EL0 and EL1 are unable to modify any content in the shared memory after SOTPM is activated, while EL3, TEE EL1, and TEE EL0 can write data into shared memory through address remapping.

C. CALLER PROCESS AUTHENTICATION

The caller process authentication in our system verifies the static region of the caller process via a cryptographic hash function. The hash values of each page are pre-generated by 4KB granularity on compile-time, and they are included in the EL3 image. Direct page manipulation can be detected through static region hash verification. However, an attacker with kernel privileges may directly compromise the original static region after copying the valid static region to another location, thus making EL3 verify the copied address by manipulating range values to the copied address during registration. Unfortunately, these spoofing attacks cannot be detected by a simple hash verification. In order to prevent these attacks, we applied an address dependent hash that includes the virtual address of each page to our design. If a caller process is Position Independent Executable (PIE), the virtual addresses utilizing the address dependent hash should be relative values that are identical with the Executable Linkable Format (ELF). The SOTPM manager performs an additional procedure to calculate relative addresses in this case.

Holding a processor in the interrupt handler for extended periods causes degradation of overall system performance. Therefore, authentication may cause system performance degradation since the verification for the caller process static region should be performed in the interrupt handler during activation. In order to minimize processing time for verification in the interrupt handler, we divide authentication over the registration phase and activation phase. During the registration phase, all loaded pages of a static region are verified, and only additional loaded pages are verified during the activation phase. In order to distinguish additional loaded pages, memory storage is required to store a verification flag indicating whether the page has been checked or not. As shown in Figure 3, we located an unused bit among bits reserved for software use. Additional memory references can be minimized if an unused bit is used as a verification flag, and as such, we decide to use the unused bit as a verification flag. The SOTPM manager marks a verification flag as 1 if the page is verified during the registration phase. During the activation phase, the SOTPM manager checks the verification flag and verifies only pages whose verification flag is 0. As a result, we minimize the performance overhead of caller process authentication by utilizing the unused bit.

D. OVERALL PROCEDURES

The overall procedures are shown in Figure 4. The SOTPM is made up of four phases: *registration, activation, invocation,* and *deregistration*. All phases except activation are identical with generic communication flow. Each phase is described in the next paragraph.

1) REGISTRATION

The main jobs of the registration phase are SOTPM initialization, first caller process authentication among divided authentication processes, and registration of essential information to the SOTPM management table. Essential information consists of process names, current virtual memory regions, and

so on. A CA allocates the shared memory and Triggering Memory (TM), which are aligned regardless of whether or not the allocated shared memory from free space originates from its own address space or a kernel address space. However, 4KB of aligned memory should be allocated since the MMU handles memory as a page or block, and the minimum page granularity is 4KB. If this does not occur, unused pages among those allocated are unnecessarily affected during the activation phase. The caller process passes virtual addresses of the allocated shared memory and TM to the kernel (REE EL1), where the corresponded process name and addresses of static regions are added to pass parameters from EL0, which then passes these to the SOTPM manager in EL3 via the SMC instruction.

The SOTPM manager performs a first caller process authentication that verifies the information and static regions of the caller process with a pre-defined list. If verification is successful, the SOTPM manager reads TTBR0_EL1 and stores it as an identifier with the relevant TM address together in the SOTPM management table. It is reasonable to use the TTBR0_EL1 value as an identifier because TTBR0_E1 indicates the base address of a translation table for initial lookup of the running process, and this table is unique for each process. After storing values, the SOTPM manager changes TM access permissions to no-access for EL0 and invalidates the Translation Look-aside Buffer (TLB). Note that the kernel is unable to make a reading permission fault exception since the least privileged kernel is designated as read-only. Completing these jobs, the SOTPM manager returns to the REE EL1.

2) ACTIVATION

In this phase, channel protection for SOTPM is performed. The CA writes data into SOTPM and reads the TM to activate SOTPM before invoking TA. Since the access permission is no-access at EL0, data abort exceptions caused by a level 3 reading permission fault occurs, and the process state is changed to EL1. Entering EL1 via data aborts, the kernel jumps to the exception vector table and executes the data abort handler. We added a hooking function related to SOTPM into the data abort handler on compile time so that the hooking function would be executed earlier than the original data abort handler. Note that it is possible to obtain an exception type from the Exception Syndrome Register (ESR_EL1) [9], which indicates syndrome information for an exception taken to EL1. The hooking function checks that the type of data abort exception is a level 3 reading permission fault by referring to ESR_EL1. If the type is verified as a level 3 reading permission fault, the hooking function executes an SMC instruction to change the process state to EL3. Otherwise, it exits the hooking function and executes the original data abort handler.

The SOTPM manager in EL3 reads information related to the data abort exception from TTBR0_EL1, Fault Address Register (FAR_EL1), and Exception Link Register (ELR_EL1) [9]. The SOTPM manager finds a tuple with an identifier that is same as the value of TTBR0_EL1 from the SOTPM management table and runs a second caller process authentication. If such a tuple is not found or authentication fails, this indicates that that exception is unrelated to SOTPM, or that an arbitrary process tried to activate SOTPM. In these cases, SOTPM does nothing and simply returns to EL1. The FAR_EL1 indicates the faulting virtual address for all synchronous instruction or data aborts, PC alignment faults, and watchpoint exceptions that are taken to EL1. The SOTPM manager checks whether the value of FAR_EL1 is the same as the TM address. It is deemed that a page fault occurs during the allocation of a new page if the values are different, in which case the SOTPM manager does nothing and, again, simply returns to EL1. The ELR_EL1 holds the address to which it will return when taking an exception to EL1. The SOTPM manager reads the next instruction address of the data abort caused instruction from ELR_EL1. If the value of ELR_EL1 is not within the range of the code section, the SOTPM manager considers the request as one sent by an attacker through a hijacking library or via modification of control flow, and thus, it returns an error.

If all conditions are satisfied, the SOTPM manager changes access permissions of the shared memory and TM to read-only and invalidates the TLB. Note that the shared memory can be double mapped to EL0 and EL1, in which case permission changes must be executed for each level. After completing the operations, the SOTPM manager returns to REE EL0 via REE EL1. The CA is able to read the TM value since its access permission is read-only. Hence, the CA executes the next instruction after reading the TM value.

3) INVOCATION

The caller application invokes a TA with an API that has already been used, and REE EL1 also runs the same as before. However, the difference is that the verification procedure is added to EL3. The SOTPM manager in EL3 checks whether or not the current TTBR0_EL1 and shared memory address match the registered values in the SOTPM management table. At the same time, the SOTPM manager verifies that the access permissions of the shared memory and TM are read-only. The SOTPM manager passes the message to TEE EL1 if the verification is successful. Otherwise, it returns an error. Through the verification procedure, SOTPM ensures that only permitted CA using SOTPM communicates with TA.

4) DEREGISTRATION

This is the last phase of SOTPM, in which the access permissions of the shared memory and TM revert to read-writable and corresponding tuples are removed from the SOTPM management table. In detail, the caller process requests the kernel to revert SOTPM to its original read-writable memory, and the kernel passes the request to the SOTPM manager. SOTPM finds the tuple which has a corresponding TTBR0_EL1 from the SOTPM management table and reverts access permissions to read-writable for both the shared memory and TM. Finally, the SOTPM manager removes the corresponding tuple and returns to REE EL1. After deregistration

is complete, the caller process deallocates the shared memory and TM, and closes the TEE device of the kernel.

These procedures are repeated for each TA invocation, and it is possible to send multiple messages for specific TA using this same method.

V. IMPLEMENTATION

We implemented a prototype of SOTPM on a Raspberry 3 Model B board, which is made up of Broadcom BCM2837 (4x Cortex-A53 based on Armv8-A 1.2GHz) and 1GB RAM. REE OS is Linux 4.14.56 and OPTEE 3.6.0 [17] is used as the TEE OS. Trusted Firmware-A 2.0 [18] for EL3 is included in OPTEE, and it is used as a secure monitor. In the REE, about 310 LoC for Linux kernel and 130 LoC for TEE client library were added. In the TEE, about 1070 LoC for SOTPM components and 1390 LoC for hash functions were added to Trusted Firmware-A, and no changes were made to OPTEE in TEE EL1.

A. REE CHANGES

1) MMU

Generally, the Linux kernel intends to map a memory region to a larger block descriptor wherever possible if the physical and virtual addresses are aligned. This mapping policy effectively reduces page table reference and management overhead as aspects of performance, while precise controls per page granularity are impossible. Indeed, shared memory allocated by the kernel is mainly mapped as a block descriptor for 2MB. The remaining memory may be wasted given that only tens of kilobytes are used as allocated memory. In order to minimize memory waste, we modified MMU codes to be forcibly allocated into a page descriptor of 4KB granularity on shared memory allocation.

2) DATA ABORT HANDLER

In our model, unlike previous studies, the type of data abort exception can be checked within the data abort handler during the activation phase without needing to remap the exception vectors. To achieve this, we only added a hooking function in front of the data abort handler instead of remapping the exception vectors. The hook function checks a type of data abort exception and invokes SMC instruction if a data abort exception is a level 3 reading permission fault. As a result, we successfully reduced performance overhead by minimizing SMC calls to EL3.

3) TEE DRIVER AND TEE CLIENT LIBRARY

Two system calls and two APIs are respectively added for registration and deregistration to the kernel TEE driver and TEE client library. It is not required at this point to add system calls for activation or invocation since activation uses a data abort mechanism, and invocation uses a prior system call and API. The activation API is implemented as a static inline function in the header file, which is included in the code section of the caller process. An attack that activates SOTPM by hijacking the TEE client library can also be prevented, as the SOTPM manager checks that activation is performed in the code section of the caller process.

B. TEE CHANGES - TRUSTED FIRMWARE-A

1) SOTPM MANAGER

The SMC Calling Conventions (SMCCC) [19] is the standard mechanism to format SMC instruction. Especially, the function identifier determines which service or function to invoke with options such as a calling convention (e.g., 32 or 64bit) and call type (e.g., fast or yielding). Three function identifiers are added to registration, activation, and deregistration in SOTPM, and these share the same values as the OPTEE function identifiers, except for the function number. As a gatekeeper of OPTEE services in EL3, the SOTPM manager pre-handles all OPTEE requests included in SOTPM function identifiers. The SOTPM manager also manages the contexts bound to each SOTPM request and inspects the shared memory in invocation.

In AArch64 of Armv8-A architecture, each Exception level has its own virtual address space and TTBR0. Therefore, address remapping should be done at each Exception level to access the memory of other Exception levels. This is particularly relevant to EL1 or EL0 address spaces that are entirely different from that of EL3 since EL3 uses linear mapping. Thus, we performed memory address remapping from the required regions with virtual addresses, namely EL0 or EL1, to the virtual memory space of EL3. Virtual address translation in reference to TTBR0_EL1 or TTBR1_EL1 is required to locate page tables in channel protection and caller process authentication in EL3. Hence, we implemented the virtual address translation function in the SOTPM manager in EL3.

Context switching between the REE and the TEE is an expensive procedure that causes performance overhead. In order to minimize context switching for SOTPM, the SOTPM manager is executed without context switching in the REE context. This approach calls for EL0 or EL1 to perform address remapping and access the EL3 address space since the virtual address space of EL3 is independent. However, as address remapping is impossible because the kernel integrity monitor traps the translation table update, the memory used by SOTPM in EL3 is secure, even though it runs in the REE context without switching to the TEE. In addition, the SOTPM manager is able to access protected memory via TZASC by setting the NS bit to 0 at SCR_EL3. As a result, we can remove the cost of context switching and obtain a sufficient level of security in EL3 when the SOTPM manager is executed.

2) CRYPTOGRAPHIC HASH FUNCTIONS

The hash function is one major factor that affects performance. Even though the latest algorithm may be more secure, the system designer must be able to choose the hash algorithm in view of a practical system environment. In order

to select the algorithm, we ported the MD5, SHA-1, and SHA256 from OpenSSL 1.1.1d to the SOTPM manager. Even though MD5 is not recommended to use, we ported it to compare performance with other algorithms. These are subsequently utilized for caller process authentication during the registration and activation phases.

VI. EVALUATION

A. SECURITY ANALYSIS

There are two major attack surfaces against SOTPM. One of these surfaces regards channel protection, and the other concerns caller process authentication. The details are described as below.

1) ATTACK SURFACE AGAINST CHANNEL PROTECTION

An attacker can obtain the address of the shared memory by hijacking the TEE client library or TEE driver upon the invocation of the TA. However, direct manipulation of the shared memory at the TEE client library or TEE driver is impossible after activation is complete since access permissions to the shared memory are switched to read-only during SOTPM activation. The attacker may then attempt to allocate the shared memory at another location and invoke TA via an arbitrary creation process. These invocations, though, are also blocked by the SOTPM manager, as the property of the allocated shared memory used by the attacker are not read-only. Additionally, the attacker is unable to manipulate the shared memory property because the page tables for the shared memory are protected by the kernel integrity monitor.

The attacker is unable to activate SOTPM via a process created by the attacker since the process is not registered on the allow list. Moreover, SOTPM activation is impossible in the TEE driver even though the attacker has kernel privileges. Note that a reading permission fault cannot be generated in EL1 because the configurable, least privileged kernel is read-only. An attack that manipulates payloads in shared memory after double mapping to other processes is also impossible because the kernel integrity monitor policy does not allow double mapping. At this point, an attacker may try to replace the shared memory address from registration or invocation in the TEE driver or TEE client library. However, an attempt to replace the address during registration is meaningless because any replaced memory becomes set as read-only during activation. Therefore, since the SOTPM manager can detect any replacement of shared memory addresses during activation by comparing a passing address with one from the SOTPM management table, the replacement of an address during activation is effectively blocked. In summary, SOTPM can prevent various attempts to manipulate the shared memory.

The mprotect API has been originally designed to designate a specific region as a protected region, but it is frequently misused as an attack method to disable memory protection. The mprotect is ultimately a changing request of access permissions for calling process memory pages, so that the request reaches the kernel integrity monitor, where the translation table is managed. The kernel integrity monitor checks that the request is proper and determines whether or not to allow the request. As a result, an improper changing request via an mprotect system call is also prevented by the kernel integrity monitor.

A CA activates SOTPM immediately through a data abort after writing data into the shared memory. The attacker is unable to hijack control flow in the middle even if they had kernel privileges. This is because the data abort handler is protected by the kernel integrity monitor, and the data abort handler merely checks the exception type and invokes SMC instructions immediately. Meanwhile, the MMU generates an exception for all REE EL0 or EL1 attempts that write data into the shared memory after activation. Therefore, the attacker is not able to manipulate the shared memory this way. In order to disable channel protection, the attacker may try to turn off memory protection by disabling the MMU. However, system privileged instructions to disable the MMU are emulated by the kernel integrity monitor, which blocks such requests precisely to render an attacker unable to disable the MMU.

2) ATTACK SURFACE AGAINST CALLER PROCESS AUTHENTICATION

The REE kernel sends the static region virtual addresses of a caller process, called range values, as essential information during registration. The SOTPM manager has a pre-defined list that has addresses and hash values per page for each static region. Therefore, any direct manipulation of the static region can be easily detected by comparing hash values calculated on run-time against the list value.

However, range values can be manipulated by an attacker with kernel privileges. The attacker may copy a valid static region to another location and subsequently compromise the original static region directly. Then, the attacker could pass this copied location address on as essential information to the SOTPM manager in EL3 during registration. In this scenario, it is possible for caller authentication to be bypassed since the SOTPM manager verifies the static region at the copied location. This type of spoofing attack can be prevented by utilizing an address dependent hash that includes a dedicated position address. The address dependent hash uses a virtual address since virtual addresses of all sections are determined at compile time.

The Time-of-check-to-time-of-use (TOCTOU) attack is a method that exploits a timing gap between verification and the use of the verification result. In order to succeed with this attack, the attacker should first place compromised codes somewhere in the memory and manipulate a page table to reference compromised codes after authentication is completed. However, this can all be prevented because unauthorized change requests for page tables are rejected by the kernel integrity monitor.

B. PERFORMANCE EVALUATION

We evaluated our proposal in two different test environments: "Pure Linux," in which no solutions are applied, and

TABLE 3. The results of the LMBench latency benchmark.

Operation	Pure Linux (ms)	SOTPM-Enabled (ms)	Overhead	SeCReT Overhead	SeCReT_Opt Overhead
Null call	0.671	0.672	1.0015x	3.9259x	4.2x
Null I/O	1.091	1.096	1.0046x	3.7327x	4.3x
Stat	17.11	17.25	1.0082x	-	-
Open / Close	29.23	29.2	0.999x	1.6264x	1.5x
Signal handler install	1.563	1.56	0.9981x	-	-
Signal handling	12.81	12.77	0.9969x	-	-
Fork	1013.9	1016.9	1.003x	1.1819x	1.2x
Fork / Exec	5837.2	5893.8	1.0097x	1.1791x	1.2x
Fork / Shell	12K	12K	1x	-	-

TABLE 4. Performance by payload size for test application using SHA256 in a static region about 8KB in size.

Payload Size	Pure Linux (ms)	SOTPM-Enabled (ms)	Overhead	SeCReT Overhead	SeCReT_Opt Overhead
1KB	112.149	112.597	1.004x	137.74x	2.8637x
2KB	112.155	112.834	1.0061x	126.95x	2.2097x
4KB	112.166	112.694	1.0047x	100.32x	1.0077x
8KB	112.3	112.779	1.0043x	100.07x	1.0227x
12KB	112.278	112.668	1.0035x	-	-
16KB	112.36	112.899	1.0047x	-	-

"SOTPM-enabled Linux," in which our scheme is applied. Since the test board we used is more powerful and different from the board used in SeCReT or SeCReT optimization, it is not suitable to directly compare elapsed time between the experiments. However, comparing overhead ratios may be helpful to determine the overhead incurred by each solution. Therefore, we run a comparison with SeCReT or SeCReT optimization only in terms of overhead ratio. Note that SHA-1 is used for caller authentication in SeCReT and SeCReT optimization.

1) LMBench

The LMbench is a series of micro-benchmarks intended to measure basic operating system metrics, such as system call invocation, fork, open/close, and signal handling. Using LMBench, we are able to evaluate the OS performance impact of SOTPM. The benchmark was run 10 times in each test environment.

Table 3 represents the results of LMBench performed, with each value showing the average value after running 10 times. Null call indicates system call invocation, where null I/O means reading and writing 1 byte to /dev/null and the stat, open and close represents the result of each system call on a temporary file. The results show that the overhead for various system calls remains under 1%. In the case of open and close in pure Linux, one of the 10 results took additionally about 400ms. That causes that performance looks enhanced in SOTPM-enabled Linux, but we have to consider that it is within an error range on the running system environment. Consequently, even considering the error, SOTPM hardly affects the performance of system calls.

Next, the operations that the signal handler installs along with signal handling mean that a signal handler has been inserted and is running. The results related to signal handling even suggest an improvement in performance after enabling SOTPM, but it is caused by the same reason with the case of open and close - one of the results took a little time in pure Linux. Hence, we can conclude that SOTPM also does not significantly affect signal handling. Even though system calls and signal handling are types of software exceptions, the hook function added to the data abort handler skips handling them since these are executed only for the reading permission fault exception. Therefore, the results showing low-performance overhead are reasonable and confirm that our hook function works well as per our design.

Finally, fork related operations such as fork and exec represent the creation of a new process. The results show that a small amount of processing time is increased compared with other operations like system calls. This is to be expected because the permission faults for new pages may occur during page table initialization or the page table walk portion of process initialization. However, the overheads for fork related operations still remain under 1% due to minimized involvement of the exception handler.

Our proposal incurs significantly lower system overhead for all system APIs than SeCReT or SeCReT optimization. In addition, the results of LMbench benchmark indicate that the performance overhead of SOTPM is quite low, so that it can be easily applied to commercial products.

2) OVERALL SOTPM OVERHEAD

In order to measure overall overhead resulting during communication due to the addition of SOTPM, we implemented a test application in which a CA copies data into the shared memory as a defined size and the corresponding TA returns a pre-defined string. We performed measurements 10 times on a pure system and a SOTPM-enabled system, holding the hash algorithm as SHA256 and static region size of the caller process at 4KB. As shown in Table 4, the test results illustrate that SOTPM overhead is under 0.5% regardless of message

size, unlike SeCReT and SeCReT optimization, which are largely affected by message size. Moreover, the results show that our proposal has a lower overhead ratio than either SeCReT or SeCReT optimization, even though SHA256 is used instead of SHA-1, which was used in SeCReT. Consequentially, the overhead caused by increasing payload size in SOTPM hardly has an impact on system performance.

Next, we ran the test 10 times with the payload size fixed at 8KB to measure overhead by the static region size of CA and determine the impact of various algorithms. Table 5 represents performance results with the hash algorithm and the static region size. The results show mild increments of growing overhead correlating to incrementally larger static region sizes and more secure algorithms. In detail, it shows that about 150ms is increased by a page when SHA256 is used, and SHA256 is a little more burden than SHA1 or MD5. However, the overhead increments are an acceptable level given that SHA256 security strength. We know that a caller authentication hash calculation majorly affects performance, but our results indicate that the impact is quite limited. However, the system designer should consider that these results may vary with each hardware environment of a targeted board, given that hash calculation time largely depends on CPU performance and whether or not there is support for hardware cryptographic engine.

TABLE 5. Hash algorithm performance for test application using 8KB payload.

Algorithm	Static Region Size	SOTPM-Enabled (ms)	Overhead
SHA256	4KB	112.799	1.0043x
	8KB	112.823	1.0047x
	12KB	112.977	1.006x
	16KB	113.236	1.0083x
	20KB	113.392	1.0097x
	24KB	113.529	1.0109x
SHA1	24KB	113.108	1.0072x
MD5	24KB	113.026	1.0065x

In summary, our performance test results show that our proposal produces an insignificant increase in performance overhead, making it suitable to integrate into commercial products.

VII. DISCUSSION

In this section, we discuss the limitations of our proposal. We assume that our model is robust against the following attacks due to its high difficulty to run, but nevertheless, the attempts at these attacks may still occur. We describe detailed scenarios and suggest countermeasures for each attack.

A. OVERFLOW ATTACK AGAINST SHARED MEMORY

An attacker can perform an overflow attack to manipulate payloads in an activated SOTPM environment if all conditions below are satisfied: 1) The physical addresses are continuously mapped to a virtual address. 2) An attacker can access any virtual address lower than the SOTPM address with write permission, 3) The distance between the lower address and SOTPM address is close enough to perform the attack.

The first condition depends on the kernel address mapping policy for the shared memory. Generally, the kernel makes multiple memory blocks and maps physical addresses to virtual addresses continuously within a memory block. Especially, a memory pool, which is a pre-allocated memory set, is used to minimize allocation overhead. If the shared memory uses a memory pool made up of one memory block, two allocated addresses may be physically contiguous. The second condition may be satisfied if an attacker obtains a low address as a dangling pointer by repeating allocation and deallocation. This method exploits the policy of allocating kernel memory that tends to reuse prior allocated memory regions. In addition, the attacker should have kernel privileges to locate the address of SOTPM, and other SOTPMs should not exist between the obtained lower address and the objective SOTPM address. The third condition is essential to successfully stage an overflow attack. The attacker should know the structures between the obtained lower address and SOTPM to overwrite data at a specific position in SOTPM. If the distance is too far, the analysis of memory structure becomes difficult, so that overflow attack becomes difficult to perform.

Therefore, we recommend not allocating any already used addresses for the shared memory so that the third condition cannot be satisfied. Although a random allocation policy may be a better choice than ''first-fit'' or ''best-fit'' policies in terms of security, this may increase allocation overhead. Moreover, applying to defend techniques, such as stack guard against stacks or heap overflow attacks, it may be a reasonable solution despite the additional burden of the memory allocator implementation for shared memory. Note that system performance may decrease when additional defend techniques are applied.

B. CONTROL-FLOW HIJACKING ATTACK AGAINST APPLICATIONS

Generally, the allocation of memory, writing data, activation, and invocation are sequentially executed as continuous execution flow. An attacker is unable to change control flow through direct code manipulation before execution since the framework authenticates the signature of the application on load. However, control flow hijacking at some point between writing data and activation is possible, even though this is very difficult to accomplish. The attacker may exploit vulnerabilities of the application and manipulate payloads in front of the activation API through control flow hijacking with an overflow attack, code injection attack, or return-oriented programming (ROP) attack. Moreover, the attacker may try to interfere in the control-flow of writing data into shared memory by running the thread on a different CPU core in parallel to the main CA thread before the activation.

A control-flow interference by a thread on a different CPU core can be prevented by applying a locking mechanism like a spinlock to shared memory at the application-level.

Mitigation for control flow hijacking attacks generally involves applying Control Flow Integrity (CFI). There are many techniques to CFI, which are categorized according to correctness (fine-grained, coarse-grained), verification objects (forward-edge, backward-edge), and information (static, dynamic). Therefore, it is required to apply appropriate CFI techniques in view of application security requirements. However, this subject is the topic of other major research domains and out of scope for this paper. As such, we assume that a CA is robust against control-flow hijacking attacks.

VIII. RELATED WORK

A. TrustZone-BASED TRUSTED EXECUTION ENVIRONMENT

The TEE supported by ARM TrustZone is utilized to manage sensitive data on end-to-end communication with an external service provider like banking, payment, and authentication. In addition, a TrustZone-based TEE may be used to obtain reliable data from internal peripherals through TrustZone Protection Controller (TZPC), which controls access to peripherals. In order to source reliable and secure data from sensors, some researchers have proposed the use of a trusted sensor [20]. TrustedUI [21] and TrustedOTP leverage TrustZone to protect sensitive input in a user interface and support software-based one-time passwords. Moreover, a TrustZone-based TEE is used to provide various security-enhancing services for a device. TZ-RKP [5], [6] and Sprobes [7], for example, are kernel integrity monitors that reside in the TEE and monitor the static region of the kernel. TrustDump [22], TrustShadow [23], and Ninja [24], on the other hand, leverage TrustZone to acquire information about and protect the system from malware. C-FLAT [25] and OAT [26] utilizes the TEE to remotely attest the control flow integrity of an application. Through TEE virtualization, the openness of a TrustZone-based TEE can be improved. To this end, vTZ [27] and PrivateZone [28] leverage hypervisor to virtualize a TrustZone-based TEE. Sanctuary [29] also supports multiple isolated, virtualized compartments based on an REE that are able to execute a TA by assigning a dedicated CPU. In order to expand the protection scope of TrustZone, SecTEE [30] proposes a TEE-based architecture that is robust against certain side-channel attacks. Meanwhile, TrustZone-based communication channels also have a critical security weakness that is used to leverage attacks against a TEE. SeCReT [2] and SeCReT optimization [3] prevent attacks exploiting these weakness through channel encryption. SOTPM leverages precise permission controls of MMU to prevent attacks on communication channels.

B. KERNEL INTEGRITY MONITOR

Various approaches have also been proposed to monitor kernel integrity. One representative approach is to keep the monitor in a secure area, such as a hypervisor, external hardware, or secure monitor, to protect the monitor from attacks. Secvior [31] implements a tiny hypervisor-based hardware memory protection and CPU virtualization in order to shield the kernel from attackers. Lares [32] places hooks in a guest virtual machine (VM) and performs security verification on guest VM in security VM on the hypervisor in order to monitor the guest VM. Next, although SIM [33] adopts the same architecture as Lares, it utilizes hardware memory protection and hardware virtualization while minimizing the involvement of the hypervisor. HookSafe [34] relocates and protects thousands of kernel hooks with hardware-based, page-level protection from hijacking attacks. As a hardware-based approach, HyperCheck [35] and HyperSentry [36] leverage the system management mode supported on x86 to monitor the hypervisor and kernel integrity. KI-Mon [37] and Vigilare [38] are implemented via independent hardware based on system bus monitoring. Meanwhile, a secure monitor-based approach utilizes ARM TrustZone. TZ-RKP [5], [6] and Sprobes [7] reside in TEE and monitor the static regions of the kernel. A TrustZone-based kernel integrity monitor performs kernel de-privileging, system control instruction emulation, and page table update trapping. SOTPM should protect the exception vector and page tables in EL1, which can be achieved with a TrustZone-based kernel integrity monitor.

C. MEMORY INTEGRITY VERIFICATION

Caller process authentication is based on memory integrity verification in an application, and the scope of memory may be codes, data, or both. Since software-only integrity verification methods are proven insufficient in terms of reliability, previous studies focused on hardware-assistant verification methods. XOM [39], Palladium [40], and corrected XOM [41] are secure processor-based memory verification methods that use encryption, cryptographic hash, and message authentication code (MAC). Several different approaches have been studied for memory integrity verification in secure processors. A MAC-based scheme [39] and log hash [42], [43] write the MAC or hash into memory and verify the value upon reading from memory. A Merkle Tree-based scheme proposes the structural use of the MAC or hash [41] for efficient verification, while one Bonsai Merkle Tree [44] focuses on reducing calculation overhead of the original Merkle Tree. Overshadow [45] is implemented on a hypervisor and utilizes memory encryption and decryption within an application. We restricted our verification scope in SOTPM to the static region, which is comprised of codes and constants, to reduce the performance overhead. However, the scope can be expanded to dynamic data, such as variables in view of system security requirements. Since TrustZone can also act as a secure processor, caller process authentication can take advantage of previously proposed approaches to ensure data integrity. In addition to memory integrity verification, it is possible to apply additional control-flow integrity verification when required.

IX. CONCLUSION

A shared memory scheme is the most widely used communication technique between isolated execution environments in ARM TrustZone architecture. However, this approach has weaknesses in that it is impossible to check the integrity of a message or authenticate a sender. Our proposed SOTPM is a practical and secure communication scheme that aims to resolve these weaknesses through channel protection and caller process authentication. It can be easily integrated into commercial products since it is uncomplicated and does not significantly increase performance overhead. Moreover, SOTPM can be extended to a cheaper A-series ARM architecture, as it does not require additional H/W extensions except TrustZone and VMSA.

REFERENCES

[1] *ARM Security Technology : Building a Secure System using TrustZone Technology (PRD29-GENC-009492C)*, ARM, Cambridge, U.K., 2009.

[2] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "SeCReT: Secure channel between rich execution environment and trusted execution environment," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2015, pp. 1–15.

[3] J. Jang and B. B. Kang, "Securing a communication channel for the trusted execution environment," *Comput. Secur.*, vol. 83, pp. 79–92, Jun. 2019.

[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proc. IEEE Symp. Secur. Privacy*, May 1997, pp. 65–71.

[5] A. Azab and P. Ning, "Methods, systems, and computer readable medium for active monitoring, memory protection and integrity verification of target devices," U.S. Patent 9 483 635, Nov. 1, 2016.

[6] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 90–102.

[7] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the TrustZone architecture," 2014, *arXiv:1410.7747*. [Online]. Available: http://arxiv.org/abs/1410.7747

[8] *ARM Cortex—A Series Progmammer's Guide for ARMv8-A (DEN0024A)*, ARM, Cambridge, U.K., 2015.

[9] *Arm Architecture Registers Armv8, for Armv8-A architecture profile (DDI 0595)*, ARM, Cambridge, U.K., 2018.

[10] *Architecture Reference Manual: Armv8, for Armv8-A Architecture Profile (DDI 0487E.a)*, ARM, Cambridge, U.K., 2019.

[11] *ARMv8-A Address Translation Version 1.0 (ARM 100940_0100_en)*, ARM, Cambridge, U.K., 2017.

[12] S. R. America, *Whitepaper: Samsung Knox Security Solution*. Suwon-si, South Korea: Samsung Electronics Cooperation, 2017.

[13] Cve-2018-9488. *MITRE Corporation*. Accessed: Aug. 1, 2020. [Online]. Available: https://www.cvedetails.com/cve/CVE-2018-9488

[14] D. Shen, *Defeating Samsung Knox With Zero Privilege*. Seattle, WA, USA: BlackHat, 2017, pp. 13–14.

[15] bits please.blogspot.com. (2016). *Trustzone Kernel Privilege Escalation*. [Online]. Available: http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-esca%lation.html

[16] Q. Blog. (2018). *Attacking the Arm's Trustzone*. [Online]. Available: https://blog.quarkslab.com/attacking-the-arms-trustzone.html

[17] *Op-Tee Documentation. Linaro*. Accessed: Aug. 1, 2020. [Online]. Available: https://optee.readthedocs.io/en/latest/

[18] *Trusted Firmware-A Documentation. ARM Limited and Contributors*. Accessed: Aug. 1, 2020. [Online]. Available: https://trustedfirmware-a.readthedocs.io/en/latest/

[19] *SMC Calling Convention: System Software on ARM Platforms (DEN 0028B)*, ARM, Cambridge, U.K., 2016.

[20] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services - MobiSys*, 2012, pp. 365–378.

[21] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proc. 5th Asia–Pacific Workshop Syst. - APSys*, 2014, pp. 1–7.

[22] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, "Trustdump: Reliable memory acquisition on smartphones," in *Proc. Eur. Symp. Res. Comput. Secur.* Springer, 2014, pp. 202–218.

[23] 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 *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2017, pp. 488–501.

[24] Z. Ning and F. Zhang, "Ninja: Towards transparent tracing and debugging on ARM," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 33–49.

[25] 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 *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 743–754.

[26] Z. Sun, B. Feng, L. Lu, and S. Jha, "OAT: Attesting operation integrity of embedded devices," 2018, *arXiv:1802.03462*. [Online]. Available: http://arxiv.org/abs/1802.03462

[27] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 541–556.

[28] 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 Trans. Depend. Sec. Comput.*, vol. 15, no. 5, pp. 797–810, Sep. 2018.

[29] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with user-space enclaves," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

[30] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1723–1740.

[31] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proc. 21st ACM SIGOPS Symp. Operating Syst. Princ.*, 2007, pp. 335–350.

[32] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2008, pp. 233–247.

[33] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proc. 16th ACM Conf. Comput. Commun. Secur. CCS*, 2009, pp. 477–487.

[34] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proc. 16th ACM Conf. Comput. Commun. Secur. CCS*, 2009, pp. 545–554.

[35] J. Wang, A. Stavrou, and A. Ghosh, "Hypercheck: A hardware-assisted integrity monitor," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Berlin, Germany: Springer, 2010, pp. 158–177.

[36] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: Enabling stealthy in-context measurement of hypervisor integrity," in *Proc. 17th ACM Conf. Comput. Commun. Secur. CCS*, 2010, pp. 38–49.

[37] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang, "Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object," in *Proc. 22nd USENIX Security Symp. (USENIX Secur.)*, 2013, pp. 511–526.

[38] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang, "Vigilare: Toward snoop-based kernel integrity monitor," in *Proc. ACM Conf. Comput. Commun. Secur. CCS*, 2012, pp. 28–37.

[39] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 168–177, Dec. 2000.

[40] A. Carroll, M. Juarez, J. Polk, and T. Leininger, "Microsoft palladium: A business overview," *Microsoft Content Secur. Bus. Unit*, pp. 1–9, Aug. 2002. [Online]. Available: http://download.microsoft.com/documents/australia/corporateaffairs/palladium_white_paper_public.pdf

[41] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *Proc. 9th Int. Symp. High-Perform. Comput. Archit. HPCA*, Feb. 2003, pp. 295–306.

[42] D. Clarke, S. Devadas, M. Van Dijk, B. Gassend, and G. E. Suh, "Incremental multiset hash functions and their application to memory integrity checking," in *Proc. Int. Conf. theory Appl. Cryptol. Inf. Secur.* Berlin, Germany: Springer, 2003, pp. 188–207.

[43] G. E. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proc. 22nd Digit. Avionics Syst. Conf.*, Dec. 2003, pp. 339–350.

[44] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS- and performance-friendly," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2007, pp. 183–196.
[45] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 2, pp. 2–13, Mar. 2008.

DONGWOOK SHIM received the B.S. degree from the Engineering Division of Electronics and Electrical and Computer, Hanyang University, Seoul, South Korea, in 2006. He is currently pursuing the M.S. degree in information security with the Graduate School of Information Security, Korea University, Seoul. Since 2005, he has been working with Mobile Communication Business, Samsung Electronics Corporation, Suwon, South Korea. His research interests include embedded system security, trusted execution environment, applied cryptography, and software security.

DONG HOON LEE (Member, IEEE) received the B.S. degree from the Department of Economics, Korea University, Seoul, in 1985, and the M.S. and Ph.D. degrees in computer science from The University of Oklahoma, Norman, in 1988 and 1992, respectively. Since 1993, he has been with the Faculty of Computer Science and Information Security, Korea University. He is currently a Professor and the Director of the Graduate School of Information Security, Korea University. His research interests include the design and analysis of cryptographic protocols in key agreement, encryption, signatures, embedded device security, and privacy-enhancing technology.