COMPUTERS & SECURITY 83 (2019) 79–92

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Securing a communication channel for the trusted execution environment

*Jinsoo Jang, Brent Byunghoon Kang**

Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology (KAIST), 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea

ARTICLE INFO

Article history:
Received 7 January 2018
Revised 28 January 2019
Accepted 28 January 2019
Available online 31 January 2019

Keywords:
System security
Mobile device security
Trusted execution environment (TEE)
ARM TrustZone
Communication channel protection

ABSTRACT

As a security extension to processor, ARM TrustZone has been widely adopted for various mobile and IoT devices. The protection is conducted by separating the system into two domains: the rich execution environment (REE) and the trusted execution environment (TEE). Although the TEE effectively isolates the critical resources based on hardware access control technologies, the communication channel between the REE and the TEE has been regarded as vulnerable and exploited by attackers to deliver malicious messages to the TEE, which undermines the entire TEE security. SeCReT (NDSS 15) introduced the first solution to protect the communication channel. Unfortunately, this method has several challenges associated with it, making it difficult to deploy the solution in production devices. This study illustrates such challenges in terms of performance and security. In addition, a design optimization of the initial version of SeCReT is proposed to mitigate these challenges and evaluated to highlight its effectiveness.

1. Introduction

Critical operations in computing systems, such as online banking, have increased the need for a secure execution environment. As a response to this, the security community has explored various techniques to provide a trusted execution environment (TEE) to devices. The TEE can be classified as a software- (Payne et al., 2008; Seshadri et al., 2007; Sharif et al., 2009) or hardware-based approach (ARM, 2017; Johnson et al., 2012; Kim et al., 2016; Moon et al., 2012; TPM, 2011) depending on how it is created. One of the hardware-based TEE technologies, ARM TrustZone, has been deployed to a number of mobile and embedded devices to provide the TEE. ARM TrustZone is a security extension to the ARM processor and isolates critical system components, such as memory and peripherals, in the TEE. As a result, an attacker in a rich execution environment (REE) cannot directly access any resources protected in the TEE.

On a device equipped with a TrustZone-based TEE, the application in the REE generally sends a message to the TEE to invoke one of the TEE services. The message contains the required information to invoke the service, such as the task ID, operation number, and parameters. It is delivered to the TEE by invoking the kernel privileged instruction, the secure monitor call (SMC), which is introduced as part of the TrustZone technology.

Unfortunately, this communication channel is vulnerable because TrustZone is not designed with consideration for message integrity protection and authentication. Consequently, an attacker in the REE can abuse the communication channel to indirectly access the TEE resource and attack

* Corresponding author.
E-mail addresses: jisjang@kaist.ac.kr (J. Jang), brentkang@kaist.ac.kr (B.B. Kang).
https://doi.org/10.1016/j.cose.2019.01.012

the TEE. More specifically, the attacker can arbitrarily send a crafted message to the TEE to analyze the internal workings, extract the secrets, and obtain full control over the TEE (CVE-2015-4421, 2017; CVE-2015-6639, 2017; CVE-2015-6647, 2017; CVE-2016-0825, 2017; CVE-2016-2431, 2017; Sensepost, 2017).

To address this problem, SeCReT (Jang et al., 2015) proposed message encryption for communication between the REE and the TEE. To this end, SeCReT provisioned a session key to the legitimate REE application when communication occurs. To protect the key, it interposes the mode switches between the user and the kernel and removes the key from the memory when the kernel mode enters. The key is rewritten to the memory only at the instant when the desired process uses it. Hashes for the static region of the application (code and data) are always checked before making the key accessible to prevent the attacker from modifying the code. A shadow stack is also maintained to mitigate the control-flow manipulation attack.

Despite the solid design for key protection, several challenges still need to be addressed. (C1) a copy of the key can be created. Thus, part of the key can be leaked to attackers if the developer fails to ensure that the key is carefully used. (C2) the code hash check conducted before the application accesses the key can incur a large performance overhead depending on the application size. (C3) multiple threads running in different CPU modes cannot be supported with key protection because of the key protection mechanism that hides the key based on the mode switch.

An optimization of the design is herein proposed to address the challenges and to make SeCReT more practical. The key leakage problem (C1) is addressed by a register-only crypto operation and the removal of footage regarding used registers, which is executed by SeCReT. A crypto library based on a tiny encryption algorithm (TEA) is created. SeCReT is coordinated with a kernel integrity monitor, such as TZ-RKP (Azab et al., 2017) hosted in the TEE, to solve C2. Specifically, the fact that the page table update is only allowed by the kernel integrity monitor is exploited to lock the memory pages after the first hash verification. This exempts the need for repetitive verifications. For C3, the memory domain and the domain access control register (DACR) that controls access permission to the domain on a per-core basis are adopted. This aspect is leveraged to perform thread-based access control to the session key. Security analysis and performance evaluation are also provided to highlight the effectiveness of the proposed optimization. Specifically, in the performance of the session key protection, the optimization outperforms the original SeCReT method by approximately a hundred times for the verification with a 7 KB static region of application.

The main contribution of this study can be summarized as follows. First, a method to secure the communication channel to the TEE is proposed, which optimizes the-state-of-the-art for security and performance improvement. Second, the design details that leverage the ARM system component and the practical example implementation for the secure crypto operation are provided. Third, the coordination of our solution with existing security frameworks, such as Samsung's TrustZone-based integrity measurement architecture (TIMA) (Azab et al., 2017), is explored and evaluated and can be a reference to enhance TEE security on various mobile devices.

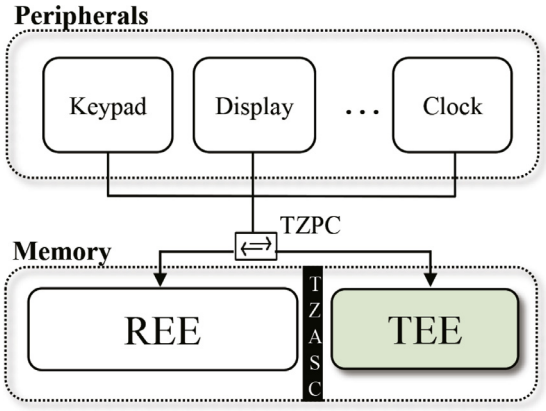

Fig. 1 – Isolation provided by ARM TrustZone.

2. Background

This work aims to secure a communication channel between the REE and the TEE that is built based on the TrustZone technology. Thus, the TrustZone technology is briefly introduced. Moreover, the memory domain and the DACR on the ARM architecture are explained as the core system features utilized to support channel protection. The kernel integrity monitor founded on the TrustZone technology is also described, which cooperates with our system to optimize the performance.

2.1. *ARM TrustZone*

As described in Fig. 1, ARM TrustZone is an ARM processor-based security extension designed to provide the TEE to devices. By orchestrating the TrustZone hardware components, it enables system on chip (SoC) designers to divide the system into two environments: the REE and the TEE. For example, the TrustZone address space controller (TZASC) separates the dynamic random access memory (DRAM) into the trusted and the rich environments. The TrustZone protection controller (TZPC) enables security critical peripherals, such as keypad and display, to be dynamically assigned to one of the two environments. Once the SoC design is completed, secure OS is generally deployed in the TEE to manage the TEE service requests from the REE.

2.1.1. *Communication channel between the REE and the TEE*

The client application (CA) in the REE first opens the TrustZone kernel driver to use the trusted application (TA) in the TEE. The CA then places the arguments in the domain-shared memory and asks the kernel driver to invoke the secure monitor call (SMC) instruction that is only executable with kernel privileges. The processor mode is switched to monitor once the SMC is invoked, which is introduced as part of the TrustZone technology. The code running in the monitor mode generally plays the role of a gatekeeper that saves and restores the context of each domain whenever the switch occurs between the REE and the TEE. The code also invokes the TEE OS, which in turn, dispatches the invoked TA by referring to the arguments delivered from the CA. The dispatched TA performs the operation requested by the CA and returns any result through the

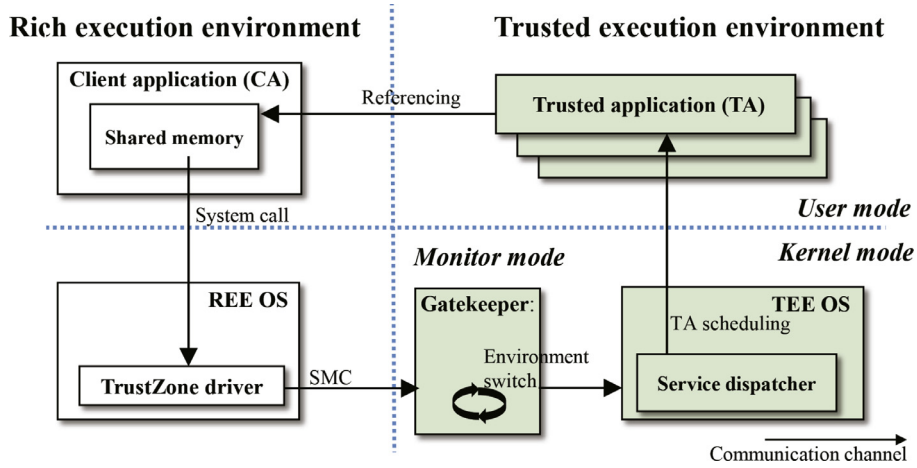

Fig. 2 – Communication channel between the REE and the TEE.

Table 1 – Example of the security-sensitive instructions and the SMC.

Instruction	Size (byte)	Description
MCR p15, 0, < Rt > , c1, c0, 0	4	Write to a control register
MCR p15, 0, < Rt > , c2, c0, 0	4	Write to a page-table base register
SMC # < imm4 >	4	SMC with a 4-bit immediate value

domain-shared memory. This series of processes is defined as a communication channel (Fig. 2).

2.2. *TrustZone-based kernel integrity monitor*

The TrustZone-based TEE not only isolates critical services (i.e., TAs), but also hosts the kernel integrity monitors (Azab et al., 2017; Ge et al., 2017) that aim to protect the static region of the REE OS. The protection is achieved by de-privileging the kernel. First, the kernel monitor configures the static regions, including code and data, as read-only. The page tables are also set to read-only, and update to the tables are verified and emulated by the kernel monitor. Thus, an attacker cannot remap the protected region by manipulating the page tables. Second, the execution of the security critical operations that can be abused to bypass the memory protection is prevented. To this end, the monitor replaces all the privileged instructions with SMC instructions to enforce the critical operations to be emulated in the TEE. As a result, malicious behaviors, such as disabling the memory management unit or manipulating the translation table base register (TTBR), can be effectively hampered. Note that the size of the instructions on the ARM architecture is fixed to 4 or 2 bytes. Hence, the lookup and replacement of such critical instructions are feasible (Table 1). Lastly, the return-to-user-like attacks are considered and prevented. The integrity monitor sets the privileged execute never (PXN) flag in every page table entry that maps the user-level memory. Hence, even if the kernel vulnerability is exploited by a malicious user application, the attacker cannot execute his/her malicious payload, which contains critical privileged instructions, with escalated kernel privilege. All enforcements from the kernel integrity monitor guarantee that the static region of the REE remains immutable.

2.3. *Memory domain and DACR*

The memory on a 32-bit ARM architecture can be defined as one of the 16 memory domains by configuring the 4-bit domain flag in the page directory entry (Fig. 3). The 32-bit domain access control register (DACR) is introduced to setup the access permission of each domain. The DACR has sixteen 2-bit access control flags, and each corresponds to one of the sixteen domains. Each flag can setup the access permission of the domain into (1) no access (0b00), (2) permission check against page table setup (0b01), (3) N/A (0b10), and (4) no permission check (0b11). The most important characteristic of the DACR is that it is banked for each processor in the multicore environment. Thus, it can be individually configured regardless of the configuration of other cores. Moreover, the change in the DACR value is instantly effective without the need to manage the translation lookaside buffer (TLB) caches.

3. Security problem of the TrustZone-based TEE

Although TrustZone provides preeminent security protection, which effectively isolates the critical resources from the REE based on the hardware-based access control technology, several technical limitations still exist and can be exploited by attackers to undermine TEE security. For example, the current design of TrustZone does not provide any mechanisms to authenticate the message sender and protect the integrity of the message from the REE, which makes the communication channel vulnerable (Fig. 4). Specifically, as discussed in Section 2.1, the communication channel on the REE side can be defined as invoking SMC instruction by placing the arguments in the domain-shared memory. This channel is completely exposed to an attacker in the REE. Therefore, it can be easily compromised by the attacker. The SMC instruction can be arbitrarily invoked and the contents in the domain-shared memory can be crafted by the attacker.

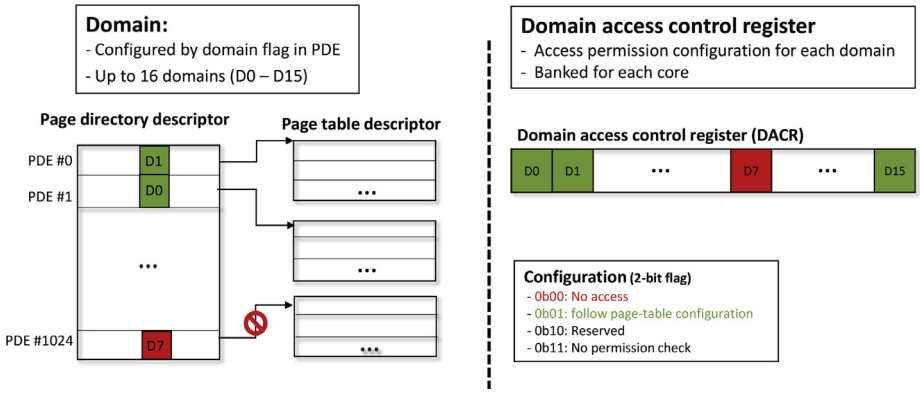

Fig. 3 – Memory domain and DACR.

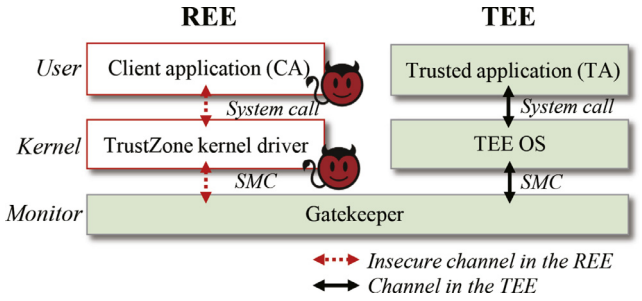

Fig. 4 – Insecure communication channel to the TEE.

4. Attack model and assumptions

4.1. *Trusted computing base*

Commercially available security facilities, such as secure boot (Arbaugh et al., 1997) and kernel integrity monitor (Azab et al., 2017; Ge et al., 2017; Samsung Electronics Co., 2017), are assumed to act as the baseline defense for a device. Hence, the following conditions are satisfied. (1) The images loaded during the device booting procedure are properly verified and loaded intact. In other words, the REE and TEE components are loaded without being compromised by the attacker. (2) During the runtime of the device, the static region of the REE OS is immutable under the protection provided by the kernel integrity monitor. Specifically, even if the attacker obtains kernel privileges, he cannot manipulate the region. The protected region includes critical kernel components, such as the code, data, page tables, and exception vector. In addition to the security conditions, the input-output memory management unit (IOMMU) (CSMMU, 2015) is assumed to be available and appropriately configured in the device. Thus, the attacker cannot execute a direct memory access (DMA) attack.

4.2. *Attack model*

The attacker is assumed to have kernel privilege in the REE. His/her aim is to compromise the TEE by maximizing the use of the kernel privilege. The TEE is separated and protected by hardware access control technology, such as TZASC. Hence, the attacker cannot directly access the TEE even with the kernel privileges. However, he/she can exploit this to attack the TEE because the communication channel (as the route to the TEE) is vulnerable, as discussed in Section 3. For instance, he/she is free to invoke SMC instructions, which is only executable with the kernel privilege. The attacker executes the instruction with maliciously crafted input payloads to lead the misbehavior of the TEE. The internal behavior of the TEE can be inferred by analyzing the return message from the TEE. In addition, the vulnerabilities of the TEE OS and the TA can be found and exploited by the malicious message sent through the vulnerable communication channel.

The vulnerable channel has been exploited to attack the TEE in the literature. The communication between the digital right management (DRM) service in the TEE and the client application in the REE was snooped to analyze the behavior of secure OS (Sensepost, 2017). Qualcomm Snapdragon device was compromised in a way that the bootloader is unlocked by sending a crafted message to TrustZone (QUA, 2017). Boomerang attack (Machiry et al., 2017) shows how to abuse the communication channel and the TEE to perform the confused deputy attack that leads the TEE to compromise the OS kernel.

No method is currently available to authenticate the message sender or protect the message delivered into the TEE. Hence, compromising the communication channel is an uncomplicated task to an attacker with kernel privileges. However, the static region of the kernel is still immutable with the presence of the kernel integrity monitor, which is the foundation for building a defense mechanism for the communication channel between the REE and the TEE.

5. The state-of-the-art study for securing communication channel

5.1. *Overview*

As the first work developed to secure the communication channel between the REE and the TEE, SeCReT proposes using a session key when the CA and TA communicate. To this end, SeCReT running in the monitor mode with the highest privilege in the system creates the session key, assigns it to the

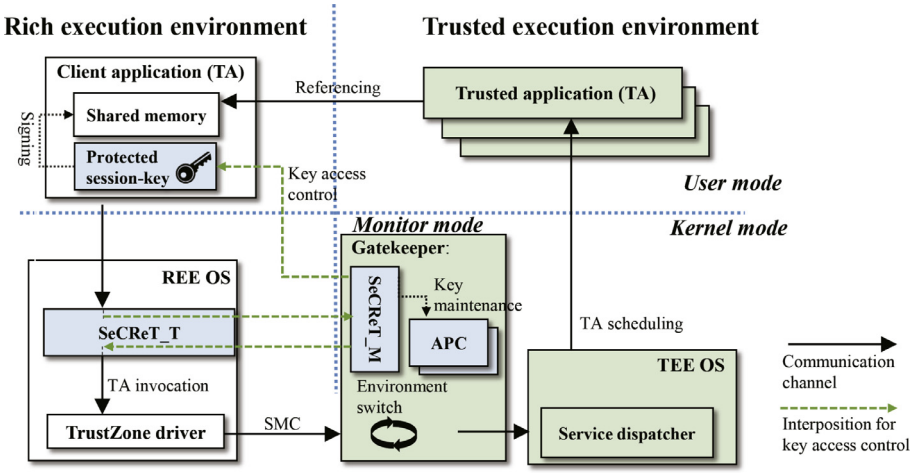

Fig. 5 – Design of SeCReT.

pre-authorized CA, and performs access control to the key to prevent it from being exposed to the attacker in the REE. The key is guaranteed to be visible only when the pre-authorized CA is running, and will be removed from the memory when the kernel mode enters.

Fig. 5 describes the overall design of SeCReT, which consists of two components for the REE and the TEE. The main operation related to the session key management is performed in the TEE. Specifically, the SeCReT code in the monitor mode (i.e., SeCReT_M) creates, assigns, and revokes the session key. It also maintains the list of pre-authorized CAs allowed to invoke the TAs to use the key. SeCReT introduces a data structure, called active process context (APC), to maintain the important information relevant to the session key management. The APC saves the translation table base register (TTBR) as an identifier to lookup the key-assigned CA, the hashes of the CA static region, and the value of the session key.

On the other hand, in the REE, SeCReT trampolines (SeCReT_T) are inserted in the starting points of each user mode exception handler to invoke SeCReT between the user and the kernel mode switches. The trampoline performs a simple invocation of the SMC instruction with arguments that present information necessary to maintain the session key, such as the process descriptor address and kernel stack address. In addition to the exception handlers, the trampolines are also inserted into some kernel codes that handle the creation and termination of processes, which is required to notify the pre-authorized CA events to SeCReT. Note that the trampolines are inserted in the kernel static region protected by the kernel integrity monitor. Thus, the integrity of the trampolines is also protected.

5.2. *Session key protection mechanism*

The session key protection requires the involvement of SeCReT early in the device boot sequence. During the secure boot sequence, SeCReT calculates the hashes of the authorized CA static region and stores them in the TEE. Note that the list of the authorized CAs allowed to use the TEE services is predefined and available during the boot sequence. During the device runtime, the event of the pre-authorized CA execution is notified to SeCReT, which leads SeCReT to create an APC for this CA. When the CA asks for a session key assignment, SeCReT creates a key and stores it in the APC. However, the key is not immediately provisioned to the memory of the CA. Instead, the permission of the memory page for provisioning the key is configured as no-access, which is neither writable nor readable. Because of this page permission, any CA attempts to use the key incurs a page fault, which is notified to SeCReT by the trampoline (SeCReT_T) at the starting point of the data-abort exception handler. When SeCReT recognizes this request for the session key, it calculates the hashes of the present pages of the CA and verifies them against the pre-calculated hashes obtained during the boot sequence. By doing so, SeCReT validates the legitimacy of the current CA. Provided that the current CA is determined as legitimate to use the TEE services, SeCReT makes the key accessible when the CA is rescheduled to run. In addition, SeCReT hides the key whenever the mode enters to the untrusted kernel.

6. Challenges with SeCReT

In this section, the challenges associated with the current design of SeCReT that might make device manufacturers reluctant to adopt SeCReT in production devices are discussed.

6.1. *Performance overhead for the session key protection*

SeCReT provides the session key to client applications in the REE to secure the communication channel. Hence, proper protection of the key is critical to the success of the approach. However, from the point of view of the attacker, one of the easiest ways to obtain the key is to manipulate the application code such that it leaks the key by itself. To prevent this, SeCReT first checks the integrity of the code hash whenever the key is provided to the process. This approach can lead to severe runtime performance degradation if the code to be checked is too large. The hash check is conducted in the TEE, which runs in the secure mode of the processor. Thus, the tasks in the REE

(i.e., non-secure mode) might experience starvation if the user of the session key resides in the memory with a large amount of the present code to be checked.

6.2. Incompatibility with a multi-threaded application

The current design of SeCReT does not support a multi-threaded application because the key protection mechanism relies on the manipulation of the memory page shared between different threads and, thus, cannot support different statuses for each thread. More specifically, the key value in the memory page is flushed and restored depending on the status of the processor mode (flushed in the kernel mode and restored in the user mode). Hence, if different threads are running in different processor modes, this situation cannot be properly handled by SeCReT because the memory can exclusively present one of two cases: key provisioning or key hiding. For example, scheduling a new thread that accesses the key might expose the key to the other threads running in the kernel mode on a different core.

6.3. Security issue with session key usage

Even if the session key is protected by SeCReT, part of the key or the entire key can be exposed to the attackers while the key is used. In other words, the protection of the key can only be guaranteed when the key resides in the memory page initially allocated for the key provisioning. Unfortunately, without the careful usage of the developer, the key can be copied to the memory, such as the stack and heap, while the crypto operation is being conducted. The copy can leak the key to the attacker and undermine the security of SeCReT because the memory, other than that of the initial key assignment, is not protected.

7. Mitigation and design optimization

In this section, the design optimization of SeCReT to address the challenges discussed in Section 6 is proposed.

7.1. Coordination with the kernel integrity monitor

In the original design of SeCReT, a hash verification is conducted right before the key is written to the memory page. All the present code pages are checked in this phase. This verification repeats when the key is used, which is the major reason for the performance degradation in the session key access control.

The code pages are configured as read-only for the kernel and the user after the first hash verification to minimize the number of verifications and reduce the overhead. Once the page table is configured for the read-only permission for the page, it should not be manipulated by an attacker. This can be guaranteed by coordinating SeCReT with the kernel integrity monitor. As discussed in Section 2.2, the integrity monitor restricts the page table update in the REE, verifies, and emulates any update to the page table in the TEE. Although this is essentially designated to protect the static region of the kernel,

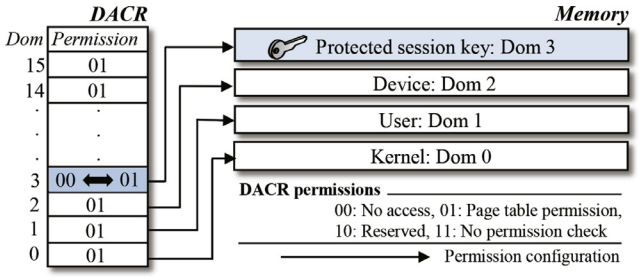

Fig. 6 – Domain and DACR configuration for the session key access control.

the mechanism is extended to protect the client application that uses a session key.

Although the verified code and data are protected by setting the page table, new memory pages can be loaded during the crypto operation process. To address this, the behavior of the kernel integrity monitor needs to be slightly updated such that it can cooperate with SeCReT. The integrity monitor is always invoked to handle this event because the new page loading requires a page table update. Instead of directly returning to the REE when the page table update is completed by the integrity monitor, SeCReT can be invoked to additionally verify the newly loaded page against the pre-calculated hash. If the new page is verified to be intact, its page table entry is then updated once again to enforce the page to be read-only against the kernel.

7.2. Adoption of the DACR for the session key protection

SeCReT interposes between the mode switches to protect the session key and removes the key from the memory when the mode enters the kernel. The key is restored in the memory at the instance it is used in the user mode. This memory and page table manipulation-based access control to the key is effective only when the process is running in a single thread. That is, as discussed in Section 6, the key used by a multi-threaded process cannot be properly protected because the mechanism that manipulates the memory cannot handle a case where different threads run in different processor modes at the same time. As a result, the original design of SeCReT cannot support a multi-threaded process.

Accordingly, the DACR is leveraged in association with a page table configuration to resolve this problem. On the 32-bit ARM architecture, 16 domains can be defined by configuring the domain flag in the first-level page table entry. By default, Linux assigns 0, 1, and 2 for the kernel, user, and device domains, respectively. The permission of each domain is configured by the DACR, which has 16 two-bit permission flags for each domain, as discussed in Section 2.3. In our design optimization for SeCReT, the domain for the session key is newly assigned as “3” (Fig. 6). This domain assignment is fulfilled when the memory page is initially allocated in response to the session key request. Note that unlike the original version of SeCReT that writes the key value to the memory on demand based on the occurrence of the access to the key, the key value always resides in the memory page in our design

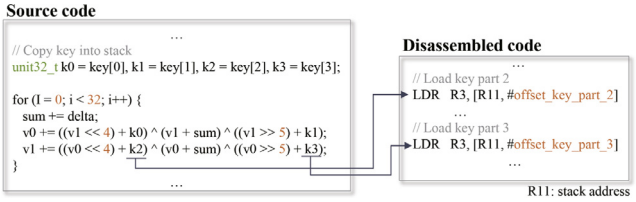

Fig. 7 – Original crypto library.

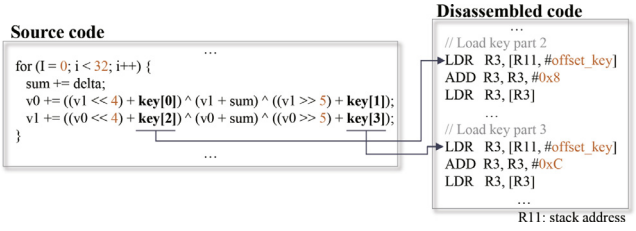

Fig. 8 – PoC of the SeCReT crypto library.

enhancement. The DACR that configures the access permission of each domain is utilized to perform access control to the key.

The access permission to the domain for the session key assignment is initially set to no-access (NA). Processes accessing the key, which resides in the memory domain with NA permission, cause a data-abort exception, which is one of the exception types available in the 32-bit ARM architecture. The occurrence of the data-abort branches the program counter (PC) to the exception vector that defines the address of each exception handler to be invoked. In this case, the handler for the data-abort is invoked. SeCReT_M in the monitor mode is invoked to handle access to the key because the starting point of each exception handler is inserted with a SeCReT trampoline (SeCReT_T).

The operation performed by SeCReT_M is similar to that of the original version. SeCReT_M first checks if the invocation originated from a legitimate kernel code (i.e., starting points of the exception handlers) by referring to the link register (LR), which is automatically set to the return address when the SMC instruction (i.e., SeCRe_T) is executed. The type of exception, which is the data-abort in this case, is also distinguished from other exception types by checking the parameter passed to SeCReT_M. This is required for SeCReT to perform the access control to the key. The APC for the current client application, which defines the session key address, is retrieved to check if the data-abort address falls within the key location. Given that the abort happens because of the key access, SeCReT sets the specific flag (key_request_flag) that indicates this event. The flag is referred to later on when the mode switches back to the user mode to enable the client application to use the key.

The remaining data-abort handling routine in the kernel does not need to be executed because the data-abort caused by the session key access is already handled by SeCReT. Thus, the return_to_user is directly executed to resume the client application. SeCReT_M is invoked again as SeCReT_T is inserted in the return to user. At this time, SeCReT checks if the last entry to the kernel was caused by the key access by checking the value of the key_request_flag. Once confirmed, SeCReT enables the key to be accessible to the client application before the mode switches to the user. This is conducted by manipulating the DACR. SeCReT_M sets the access permission for the domain that contains the key from 0b00 (NA) to 0b01, which enforces the permission to follow the setting in the page table. This way, the process can read the key by retrying the instruction that failed because of the domain access permission initially set to NA.

The advantages of using the DACR over direct memory manipulation for the key protection can be summarized in two ways. (1) A multi-threaded application can be supported for the use of the SeCReT-provided session key because the DACR is banked for each processor, thereby enabling the application of different configurations to each thread running on different processors. (2) The TLB and cache maintenance is not required for the key protection. Any change on the DACR instantly takes effect without performing cache operations, which is in contrary to the case of the REE memory being updated from the TEE for the key access control (e.g., provisioning the session key by SeCReT).

7.3. *Secure crypto library*

The session key should be used in a secure manner because simple operations can create copies of the key, as discussed in Section 6. SeCReT crypto libraries that help developers safely use the session key are developed based on a tiny encryption algorithm (TEA) (Wheeler and Needham, 1995).

To protect the key, only general registers are enforced to be used when the crypto computation is processed. Four registers of 4 bytes each are used because the key is 128-bit long. Fig. 8 shows that a simple change (instrumentation) is required in the original library to prevent part of the key from being copied out of the protected memory. This change involves some performance degradation because the stack is not used to cache a part of the key in the instrumented library in contrary to the original library (Fig. 7). Fig. 8 shows that the instrumented library requires two more instructions to load the key from the protected memory to the register. Therefore, the computation using the SeCReT library requires at least 256 additional instructions (2*4 key parts*32 loops) compared to that in the original library in this example. A comparison of the performance between the two libraries is provided in Section 10.2.2.

The footprint of the key must also be clearly removed when the mode enters the kernel mode. All the general registers are masked when mode switching occurs because of asynchronous exceptions, such as an interrupt. In contrast, some registers need to be reserved for handling system calls, such as system call number (R7) and parameters (R0-R6). Hence, the rest of the registers are masked in this case. Note that the cache for the session key does not need to be invalidated because the key is protected by leveraging the DACR, thereby eliminating the requirement for cache and TLB maintenance.

Our PoC library is built on TEA, which is quite simple. Hence, it can be manually verified that part of the key is not written to unprotected memory areas. That is, it can be readily confirmed that part of the key contained in the register during the computation is never written to the memory. However, a

manual verification could not be conducted when more complex crypto algorithms are used. Therefore, instrumentation that removes such unwanted memory writes or a runtime verification that checks such cases must be further studied. This task is set aside for our future work.

8. Implementation

The original version of SeCReT was designed based on ARMv7, which specifies the 32-bit ARM architecture. It was developed on an Arndale board with a Cortex-A15 dual-core processor. Linux version 3.9.1 and Sierra TEE (Sierraware, 2017) were deployed as the REE and TEE software stacks, respectively. The same development environment was used in our optimization to properly evaluate the changes.

8.1. *Domain and DACR adoption*

The memory domain and the DACR are leveraged to perform access control to the key. The 32-bit ARM architecture has two translation table formats: short and long. In the short-translation table format, the memory can be mapped as sections (16 MB or 1 MB) or pages (64 KB or 4 KB), while in the long-translation table format, block (2 MB or 1 GB) or page (4 KB) is supported to map the memory. The domain can be defined by configuring the domain flag on the first-level short-translation table descriptor also known as page directory entry on Linux. The long format does not support the domain. The domain flag is only available for the descriptor that points to the 1 MB section or page table, indicating that the minimum granularity of the domain is 1 MB. Thus, the 1 MB-aligned memory page is allocated for the protected session key. In addition, "3" is assigned for the session key domain because domains 0, 1, and 2 are already used by Linux.

The access permission of each domain is controlled by the DACR, which is 32-bit long. Each 2 bit defines the permission of the corresponding domain (e.g., the least significant 2 bit defines the permission of domain "0"). The DACR is configured whenever the mode switches to kernel and the data-abort caused by the legitimate access to the key occurs to perform the access control to the session key domain. On entry to the kernel, the value of the DACR is set as "0x55555515," indicating that domain 3 is not accessible. The DACR value is restored to the default value "0x55555555," which makes the access permission of all domains follow the page table entry configuration and ensures that the key is accessible to the legitimate application.

The data fault address register (DFAR) containing the faulting address of the synchronous data-abort exception is read to verify the legitimacy of the access to the key. It is checked if the value of the DFAR falls within the session key address stored in the ACL. Note that the data fault status register (DFSR) contains the domain field, but is deprecated from ARMv7.

8.2. *Coordination with the kernel integrity monitor*

The kernel integrity monitor running in the TEE is utilized to optimize the hash verification performance. The fact that any page table update is performed by the monitor in the TEE is particularly exploited. Moreover, the integrity of the client application is verified when the session key is accessed for the first time. Then, the descriptor of a small page (4 KB) is configured such that the access permission of the application static pages is set as read-only for both the kernel and user modes. All bits (i.e., 0b111) of the permission flag in the descriptor are set. To handle the runtime loading of the new page, the integrity monitor invokes SeCReT to check and lock the new page. According to the proposed design, the invocation should occur in the TEE. However, the invocation was managed to happen in the REE because the integrity monitor was not fully implemented in our work. This is realized by inserting an SMC instruction that invokes SeCReT right before the updated page table entry (PTE) is flushed from the cache (in set_pte_ext). Once SeCReT is invoked, it first retrieves the translation table base register (TTBR) of the current process and looks up the ACL with the TTBR as a key to check if the current process is actually assigned with the session key. This check happens for every process and degrades the performance of the entire system. The performance overhead caused by this procedure is evaluated in Section 10.1.1.

9. Security analysis

In terms of the session key protection, the optimization proposed in this study should not degrade the security level of the original version of SeCReT. The key is provisioned in the REE, which is regarded as an unsafe area. Hence, various attack vectors must be considered.

The attacker can try to snapshot the memory to find any footprint of the key and extract it. The untrusted kernel can try to dump the protected memory allocated for the key when the process that uses the key enters the kernel mode. This attack is prevented by managing the access permission of the memory domain specifically assigned for the session key provisioning. SeCReT interposes every mode switch to the kernel and configures the DACR to enforce the permission of the key domain as "no access" from any privileged mode. Thus, any attempt to access the key is blocked.

The attacker can also try to manipulate the DACR to restore accessibility to the key domain because he/she has kernel privileges. However, this attack, which executes the privileged instruction to manipulate the DACR, is also prevented by the kernel integrity monitor (Azab et al., 2017; Ge et al., 2017). Specifically, every privileged instruction in the REE OS is replaced with SMC instructions that invoke the kernel integrity monitor, and security critical operations are emulated by the integrity monitor in the TEE. In addition to this, executing the privileged instructions by loading a malicious kernel module or launching a return-to-user attack is also addressed by the kernel integrity monitor.

In a multi-core system, the attacker would try to exploit another core with the DACR set to the default value (i.e., every domain is set to accessible) because the DACR is banked for each core. To succeed in this attack, the attacker needs to manipulate the page table to create a new map to the session key. Unfortunately, this trial is also hindered because page table manipulation is only allowed in the TEE with the presence of the kernel integrity monitor.

Table 2 – LMBench latency microbenchmark results (in microseconds).

	Linux	SeCReT	SeCReT_Opt
Null	0.27	1.06 (3.9 ×)	1.16 (4.2 ×)
Open/Close	5.43	8.83 (1.6 ×)	8.64 (1.5 ×)
Read	0.33	1.23 (3.7 ×)	1.47 (4.4 ×)
Write	0.42	1.57 (3.7 ×)	1.78 (4.2 ×)
Fork	147.78	174.66 (1.1 ×)	181.39 (1.2 ×)
Fork/Exec	160.32	189.03 (1.1 ×)	196.11 (1.2 ×)

The attacker would try to modify the application code such that it exposes the key outside of the protected domain. SeCReT hampers this attack by setting the static region of the application to read-only after verifying the hashes of the region. The time-of-check-to-time-of-use attack that exploits the timing gap between the hash verification and the locking of the page table must also be considered. The malicious code running in the non-TEE mode can try to modify the verified code before it is locked by configuring the page table. However, this attack also essentially requires page table manipulation, which is restricted by the kernel integrity monitor. The libraries that use the key (e.g., crypto library) need to be verified before it is loaded. In our implementation, the instrumented crypto library is statically compiled such that it can be verified together with the application code.

Finally, the prevention of a control flow-based attack depends on the coarse-grained control flow integrity provided by the original version of SeCReT. As a limitation of this work, any vulnerability that enables the attacker to gain control flow of the application might expose the key. This issue will be addressed in a future work.

10. Performance evaluation

In this section, the performance variation enabled by the SeCReT design optimization is measured and evaluated. The micro and application benchmarks are executed on the native, SeCReT-enabled, and optimized SeCReT-enabled Linux OSs to evaluate the OS performance. The application performance that benefits from the protection provided by SeCReT is also measured by running a test program that performs crypto operations.

10.1. *REE OS*

SeCReT interposes every mode switch between the user and the kernel when it is enabled, regardless of the current process using the session key. Hence, it imposes some performance overhead to the entire system. This overhead is measured by running LMBench and Phoronix test suites.

10.1.1. *LMBench*

As a microbenchmark suite, LMBench provides a collection of test programs that can evaluate the performance of OS system operations. As can be seen in Table 2, six test cases, including pure context switch (null), read/write, and fork were run on three different test environments: Linux, SeCReT, and optimized SeCReT (SeCRe_Opt). SeCReT essentially imposes some overhead to the system, which is a maximum of 3.9 for the context switch, because of the interposition between the mode switches. This overhead decreased to approximately 10% for the fork and exec as the latency increased. However, SeCReT_Opt slightly induces more overhead than SeCReT because of utilizing the kernel integrity monitor in handling page fault. That is, whenever a new page is loaded, SeCReT is invoked to check if the current process was assigned with a session key and performs the integrity check for the new page (given that it falls within the static region of the application). The system calls with a low latency (e.g., null, read, and write) imposed a high overhead (approximately 4 ×), but the result revealed the tendency of the overhead being reduced to 1.2 × for the fork and fork/exec similar to the test with SeCReT. Note that SeCReT_Opt was temporally invoked from the REE OS, which imposed an additional world switch overhead, because the kernel integrity monitor was not fully implemented in our prototype. Hence, in a real situation where SeCReT_Opt and the integrity monitor cooperate in the TEE, the overall overhead is expected to be reduced because of the exemption of the world switch.

10.1.2. *Phoronix test suite*

A Phoronix test suite was run as an application benchmark. It provides comprehensive benchmark testing applications to evaluate the performance of various system features such as graphics, processor and disk. For example, nginx is an apache benchmark that measures the CPU throughput of concurrent and huge web requests. Among the applications, 10 processor-bound test cases were chosen because SeCReT does not interact with peripherals. Fig. 9 describes the results of the test runs with 10 applications, indicating the overhead of SeCReT and SeCReT_Opt normalized to Linux. In most cases, the overhead was less than 10% for both SeCReT and SeCReT_Opt. The comparison of SeCReT and SeCReT_Opt did not show any clear superiority in the performance of SeCReT over SeCReT_Opt despite the fact that the additional logic for the coordination with the kernel integrity monitor was deployed in SeCReT_Opt. The reason for this can be explained as follows: this benchmark aims to measure the system overhead introduced by optimizing SeCReT, not the overhead of a protected application. Hence, the additional overhead imposed by the coordination between the SeCReT and the integrity monitor was limited. As illustrated in Section 7.1, the latency was caused by the world switch between the REE and the TEE whenever a new page is mapped. It was quite small compared to the overall runtime of each benchmark application. Hence, most overhead was obscured. This result highlights that the SeCReT enhancement imposes a negligible overhead to the system.

10.2. *Client application*

In this section, the performance of an application that benefits from the SeCReT-provided session key protection is evaluated. The performance of SeCReT_Opt is specifically compared with the original SeCReT and analyzed to show how much performance gain or loss is achieved by the design change.

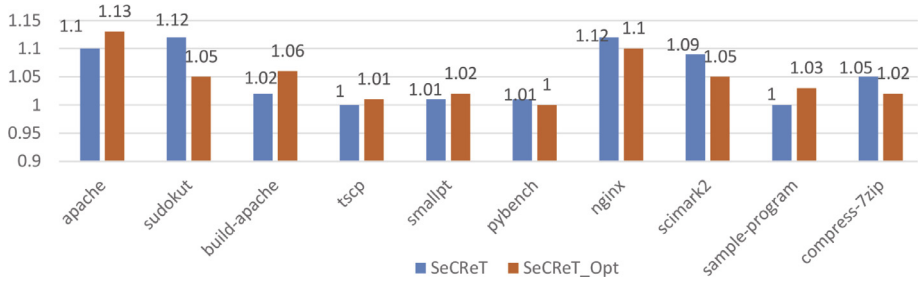

Fig. 9 – Performance comparison between SeCReT and SeCReT_Opt with 10 test cases (normalized to Linux).

Table 3 – Comparison of the crypto library performance with the logics shown in Figs. 7 (original) and 8 (instrumented).

Loop count	Original (μs)	Instrumented (μs)	Overhead
1	3.7	4.1	1.108 ×
2	7.3	7.6	1.041 ×
4	11.1	11.5	1.036 ×
8	15.1	15.7	1.039 ×
16	19.5	20.1	1.030 ×
32	24.7	25.5	1.032 ×

10.2.1. *DACR configuration vs. memory manipulation*

A multi-threaded application support requires changing the key access control mechanism. In the original version, the key is protected by directly manipulating the memory on every mode switch, which requires removing the key value in the memory, cache flushing, page table entry update, and TLB flushing. However, the mechanism was optimized to use the DACR and the memory domain to support the multi-threaded application. Hence, access control to the key was performed by manipulating the DACR between the mode switches. The performance was compared by measuring the cycle count of each case. A small cycle count difference (two or three cycle counts) was observed between SeCReT and SeCReT_Opt, which was negligible.

10.2.2. *Secure crypto operation*

Table 3 describes the performance of the original (Fig. 7) and instrumented (Fig. 8) crypto libraries. The performance of the looping part consisting of 32 rounds of loops was measured. Accordingly, 10% overhead was imposed for the first round of the loop because the instrumented library had twice as many more memory operations than the original version. However, this overhead significantly decreased from the second round because of the impact of cache that removed the need for MMUs memory read. Consequently, as an overall overhead, it was reduced to 3% to complete the loop.

10.2.3. *Key access control overhead*

Finally, the impact of SeCReT (and SeCReT_Opt) on the client applications running in three different environments was evaluated: Linux, SeCReT without key protection, and SeCReT with key protection. SeCReT without key protection denotes the case, where SeCReT is enabled, but the current application does not use the protected key.

Input: An ascii *payload* of size: 128 to 8192 bytes
Output: Encrypted *payload*
$*key$ = allocMemory()
if Key_protection **then**
  assignKeyBySeCReT(*key*)
else
  $*key$=randomValue()
end if
payload = encrypt_TEA(*payload*, $*key$)
printString(*payload*)

Fig. 10 – Pseudocode for measuring the key access control overhead.

The client application parses an input payload, encrypts it with TEA, and prints the ciphertext (Fig. 10). The payload size varies from 128 bytes to 8 KB. The 4 byte key is used for the encryption. Linux and SeCReT without key protection do not use the protected key. Hence, a random value was assigned as a key. For the key protection, the key is provisioned by SeCReT in the TEE.

Table 4 enumerates the result indicating the latency of running the CA in each environment and presents the overheads normalized to the baseline (Linux). The column without key protection indicates the performance of application that uses a random value as a key. Because the key is not protected by SeCReT, the performance of application is only affected by the system overhead imposed because of enabling SeCReT. In this case, the performance deteriorated with SeCReT_Opt because of SeCReTs cooperation with the kernel integrity monitor, which invokes SeCReT whenever demand paging happens. Consequently, 24% of maximum overhead was observed with SeCReT_Opt whereas SeCReT imposed 18% of overhead when the minimum payload size was used. As discussed in Section 8, the invocation requires the world switch between the REE and the TEE in the current implementation, and the latency of which is approximately 6 µs. However, the implementation following our design

Table 4 – Performance of client applications, the session keys of which are protected by the original and optimized versions of SeCReT (in microseconds).

Payload size (bytes)	Linux (baseline)	Without key protection		With key protection	
		SeCReT	SeCReT_Opt	SeCReT	SeCReT_Opt
128	1324.9	1572 (1.18 ×)	1648.9 (1.24 ×)	238418.1 (179.95 ×)	6539.2 (4.15 ×)
256	1543.1	1776.6 (1.15 ×)	1860 (1.20 ×)	258818.8 (167.72 ×)	6786.4 (3.81 ×)
512	1962.1	2189.2 (1.11 ×)	2274.9 (1.15 ×)	299918.7 (152.85 ×)	7135.4 (3.25 ×)
1024	2800.9	3049 (1.08 ×)	3246.6 (1.15 ×)	385818.4 (137.74 ×)	8021.1 (2.63 ×)
2048	4537.1	4738.6 (1.04 ×)	4907.5 (1.08 ×)	576018.4 (126.95 ×)	10025.8 (2.11 ×)
4096	676005.5	676804.7 (1.00 ×)	677293.9 (1.00 ×)	67823220.2 (100.32 ×)	681223.9 (1.00 ×)
8192	1685473.9	1686312.1 (1.00 ×)	1686863.3 (1.00 ×)	168672421.9 (100.07 ×)	1723824.4 (1.02 ×)

Table 5 – Performance of client applications with various size of static region (in microseconds).

	CA size (KB)			
	7	11	15	19
SeCReT	238418.1	369522.7	489621.2	637571.1
SeCReT_Opt	6539.2	8654.2	10791.3	13192.3

proposal is expected not to add the world switch overhead because the page table update will be conducted by the kernel integrity monitor that also resides in the TEE.

The test with key protection demonstrates the performance of application that uses a SeCReT-protected key. In this case, SeCReT_Opt significantly outperformed SeCReT because of the optimization on the hash check. In SeCReT, the hash check for the CA happens whenever the key is accessed. In our test, the size of the static region that needed to be checked was 7 KB, and approximately 2200 µs was required to measure the 4 KB memory page. Hence, the latency proportionally increased to the number of access to the key. The maximum overhead (179 times) was observed with the 128 byte payload, and the overhead reduced down to 100 × with the largest payload having an 8 KB length. However, compared to SeCReT, the overhead significantly reduced with our optimization. The overhead was 4 × with the 128 byte payload and reduced down to 2% with the 8 KB payload. Note that the hash check latency dominated the most overhead, and the impact of the DACR adoption and crypto library instrumentation was negligible in the CA performance evaluation.

The impact of size of CA on the performance of access control to the key is also evaluated. The experiment runs applications that encrypt 128 bytes of payload with a SeCReT-protected key. Several dummy pages, the size of which varies from 4 to 12 KB, are added to the CA to emulate the CA size increase. The effectiveness of our optimization is noticeable as can be seen in Table 5. The latency of SeCReT surges up when the CA size increases because the hashes of all the static regions of CA are repeatedly checked before the key is provisioned to the CA. By contrast, only a limited increase in the latency was observed with SeCReT_Opt. Hash validation overhead was added only once for the initial page loading, thanks to the optimization. The checked region is locked by the integrity monitor to prevent any modification. As a result, approximately 2200 µs of latency was added to the overall runtime of CA when a new dummy page is loaded.

11. Discussion

11.1. *Instrumentation for secure computation*

As an example of achieving a secure key usage, a crypto library built based on the TEA was instrumented. Because of the simplicity of the algorithm, the library was manually instrumented such that it only used the general registers for the crypto operation. The library code that saves part of the session key in the stack (Fig. 8) was manually removed. It was confirmed that the compiled binary does not contain any memory operation that leaks part of the key from the protected memory. However, this manual instrumentation would not work for more complicated crypto algorithms because modifying the source code and verifying the binary might require a significant engineering effort. As a result, a compiler extension that ensures the crypto library performs a register-only operation with the protected key will be developed in our future work. In addition, an LLVM pass (Writing an llvm pass, 2017) that conducts an alias analysis for the annotated key, tracks the user of the key, and removes any instruction that stores part of the key in the unprotected memory will be created. This pass will ensure that part of the key is always directly loaded from the protected memory into the general register whenever it is used.

11.2. *Key protection granularity*

The original version of SeCReT protected the key based on a granularity of 4 KB, which is a small page size on the 32-bit ARM architecture. In our optimization, the adoption of the memory domain and the DACR for the multi-threaded application support increased the protection granularity. The domain was defined by configuring the first-level translation table descriptor (i.e., page directory entry) that maps the memory based on 1 MB granularity. Hence, the size of the protected memory also became 1 MB. This increase in the protection size led to the waste of a large portion of the memory because the key storage occupied only a few bytes (e.g., a 4 byte key was used for our secure crypto library).

The waste can be reduced by utilizing the protected memory as a secure buffer that contains the message to the TEE. The part of the application code responsible for message creation can be located in the protected memory as well. This approach might require a further design change because not only the code, but also the stack and heap, need to be located in the protected memory. In contrast, the adoption of a

fine-grained memory protection technique can be considered. Vigilare (Moon et al., 2012) and KI-MON (Lee et al., 2017) show that the byte granularity memory protection is possible with the external monitoring technology, which can be leveraged to protect the session key. Coordinating SeCReT with external monitoring techniques will be addressed in our future work.

11.3. *Compatibility with ARMv8*

The SeCReT prototype was built on ARMv7, which specifies the 32-bit ARM architecture. However, ARMv8 that supports both 32-bit and 64-bit ARM instruction sets is different from ARMv7 in some aspects, including exception handling. ARMv8 simplifies the exception handling mechanism in a way that it consolidates various exception modes in ARMv7 (e.g., IRQ and data abort) and introduces the exception level (EL) as a unified CPU mode (e.g., EL0 for user and EL1 for kernel modes). Any exception that occurs in EL0 is caught by the exception vector in EL1, which dispatches exception handlers for the corresponding exceptions by referring to the exception syndrome register. Therefore, applying SeCReT to the 64-bit system requires some engineering effort in terms of adding the trampoline code to the proper location in a 64-bit exception vector and handlers. In addition to the change on the exception handling, some control registers are deprecated in the 64-bit architecture. For instance, the DACR, which has been leveraged herein to support the multi-threaded application, is no longer supported in the 64-bit mode. Hence, supporting the multi-thread in the 64-bit system requires the exploration of an alternative feature for the DACR. This topic is left for our future work.

12. Related work

12.1. *Secure I/O channel*

A line of studies on x86 introduced methods to build a trusted path between the application and devices and protect the interaction between the user and the I/O devices. McCune et al. (Perrig and Reiter, 2009) built a secure channel between the encrypting input device and the application that runs in the secure execution environment created by Flicker (McCune et al., 2008). A software security token (Brasser et al., 2012) and a secure transaction confirmation architecture (Filyanov et al., 2011), both of which enable the user to securely communicate with I/O devices, were built based on Flicker and the trusted platform module (TPM) (TPM, 2011). As a hypervisor-based approach, Zhou et al. (2012) showed how to create a general and human-verifiable trusted path between the arbitrary application and devices. The wimpy kernel (Zhou et al., 2014) built based on XMHF (Vasudevan et al., 2013) was introduced to provide on-demand isolated I/O channels. The hypervisor was leveraged to secure SGX I/O as well. As the first work, SGXIO (Weiser and Werner, 2017) coordinates a small hypervisor with an SGX-provided security property (e.g., attestation) to create a trusted path between the SGX enclave and the I/O devices.

Meanwhile, the TrustZone technology on the ARM essentially provides a secure I/O to the TEE services. TZPC (ARM, 2017) enables the peripherals (e.g., keypad and display) to be dynamically assigned as the TEE resources. Hence, additional security facilities are not required in the TEE to build a secure channel between the TEE service and devices. However, securing the channel between the REE and the TEE is still important in terms of the TEE service protection necessary for the trustworthiness of the secure I/O. As a result, SeCReT (Jang et al., 2015) can also be regarded as part of works that build a trusted path between a user and devices.

12.2. *Kernel integrity monitor*

As a baseline system security, many works studied efficient and safe ways to monitor the kernel integrity. Many approaches host the monitor outside of the OS to protect the monitoring framework from attackers. As a hypervisor-based approach, Secvisor (Seshadri et al., 2007) implemented a tiny hypervisor that leverages hardware-assisted virtualization technology to protect the kernel from privileged attackers (e.g., rootkit). Lares (Payne et al., 2008) introduces active monitoring that places hooks and trampolines in the monitored VM to conduct the security verification in the security VM. SIM (Sharif et al., 2009) places not only the hooks, but also the security agent inside the monitored VM to reduce the overhead incurred by switching VMs. HookSafe (Wang et al., 2009) relocates and protects the hooks based on page granularity to efficiently protect the implanted hooks.

A hardware-based approach was also explored to secure the kernel. HyperSentry (Azab et al., 2010) and HyperCheck adopt the system management mode to monitor the kernel in a secure and isolated manner. Copilot (Petroni Jr et al., 2004) snapshots the system memory from a PCI card, which provides the isolated monitoring environment. Vigilare (Moon et al., 2012) and KI-Mon (Lee et al., 2017) propose snoop-based system monitoring that can prevent a transient attack by designing external security hardware. TrustZone also enables kernel integrity monitoring to be securely conducted. TZ-RKP (Azab et al., 2017) and Sprobes (Ge et al., 2017) introduce a TrustZone-based kernel integrity monitor that de-privileges the kernel to enforce critical operations, such as page table update, to be always verified and emulated by the monitor. SeCReT benefits from the kernel integrity monitoring, in that part of SeCReT components, such as the trampoline, is protected as a part of the monitored objects. By cooperating with the integrity monitor, the performance improvement of the session key protection has been shown in our work.

12.3. *TrustZone-based TEE*

The TEE built based on the TrustZone technology has been adopted to protect critical services in mobile devices. Liu et al. (2012) implemented a trusted global positioning system (GPS) in the TrustZone-based TEE to protect the sensor service and data. Adattester (Li et al., 2015) and TrustUI (Li et al., 2014) leverage TrustZone to secure ad-related operations and critical user interface (UI), respectively. TrustOTP (Sun et al., 2015) also isolates the software-based OTP token in the mobile device TEE to achieve flexibility and security. TrustZone is used not only to provide an isolated execution environment, but also to build security-related systems. TZ-RKP (Azab et al., 2017) and Sprobes (Ge et al., 2017) host the kernel integrity monitor by taking advantage of the highest privilege accorded

to TrustZone. TrustDump (Sun et al., 2014) uses TrustZone to securely acquire system information needed for malware analysis. Ninja (Ning and Zhang, 2017) achieves stealthy application tracing and debugging by leveraging TrustZone-based TEE. TrustShadow (Guan et al., 2017) enables an unmodified application to be protected from the malicious OS by leveraging TrustZone. Brasser et al. (2016) proposed a TrustZone-based mechanism that can remotely regulate the behavior of smart devices in the restricted space. Moreover, C-FLAT (Abera et al., 2016) leverages TrustZone to realize the runtime control-flow verification. In addition to the utilization of TrustZone, ways to improve its usability and openness were also explored. TLR (Santos et al., 2014) ports a.NET framework in the TEE to improve the development productivity of trusted services. PrivateZone (Jang et al., 2018) and vTZ (Hua et al., 2017) virtualize the TrustZone for its security and accessibility improvement. Finally, the vulnerability of TrustZone and its exploitation are studied. A BOOMERANG attack (Machiry et al., 2017) abuses the semantic gap and communication channel between the REE and the TEE to execute a confused deputy attack, which leads the TEE to attack the REE OS. As a possible defense mechanism for such an attack, SeCReT (Jang et al., 2015) proposes message encryption to prevent the attackers from abusing the vulnerable communication channel.

13. Conclusion

In spite of the execution environment separation and isolation technology, TrustZone-based TEE suffers from a vulnerable communication channel that is abused by attackers to deliver a maliciously crafted message to the TEE, which can undermine the entire TEE security. To address this problem, SeCReT introduces a communication channel protection mechanism that enables a legitimate REE application to use a session key when it communicates with the TEE. To make the technique more practical, a design optimization of the original version of SeCReT is proposed. It was shown that the optimization can reduce the performance overhead, minimize the key leakage, and support a multi-threaded application.

Conflict of Interest

None.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Education, Science, and Technology (MEST) (No. NRF-2017R1A2B3006360) and the Office of Naval Research (ONR) TPCP program (No. GRANT12593416).

REFERENCES

ARM Security Technology: Building a Secure System Using TrustZone Technology. 2017, Tech. rep. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

Abera T., Asokan N., Davi L., Ekberg J.E., Nyman T., Paverd A., Sadeghi A.R., Tsudik G. C-FLAT: Control-FLow ATtestation for embedded systems software 2016, doi:10.1145/2976749.2978358.

Arbaugh WA, Farber DJ, Smith JM. A secure and reliable bootstrap architecture. In: Proceedings of the 1997 IEEE symposium on security and privacy; 1997. p. 65–71.

Azab AM, Ning P, Shah J, Chen Q, Bhutkar R, Ganesh G, Ma J, Shen W. Hypervision across worlds: Real-time kernel protection from the arm TrustZone secure world. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. ACM; 2017. p. 90–102.

Azab AM, Ning P, Wang Z, Jiang X, Zhang X, Skalsky NC. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM conference on computer and communications security. ACM; 2010. p. 38–49.

Brasser F, Bugiel S, Filyanov A, Sadeghi AR, Schulz S. Softer smartcards. Financial Cryptogr Data Secur 2012:329–43.

Brasser F, Kim D, Liebchen C, Ganapathy V, Iftode L, Sadeghi AR. Regulating arm trustzone devices in restricted spaces. In: Proceedings of the 14th annual international conference on mobile systems, applications, and services. New York, NY, USA: ACM; 2016. p. 413–25. doi:10.1145/2906388.2906390.

Cve-2015-4421. http://firmwaresecurity.com/tag/cve-2015-4421/ 2017.

Cve-2015-6639, 2017, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6639.

Cve-2015-6647. 2017 http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-6647.

Cve-2016-0825, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-0825. 2017

Cve-2016-2431, http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2016-2431. 2017.

Corelink system memory management unit, http://www.arm.com/products/system-ip/controllers/system-mmu.php, 2015.

Filyanov A, McCuney JM, Sadeghiz AR, Winandy M. Uni-directional trusted path: transaction confirmation on just one device. In: Proceedings of the 2011 IEEE/IFIP 41st international conference on dependable systems & networks (DSN). IEEE; 2011. p. 1–12.

Ge X, Vijayakumar H, Jaeger T. SPROBES: Enforcing kernel code integrity on the trustzone architecture. Proceedings of the 2014 Mobile Security Technologies (MoST) workshop, 2014.

Guan L, Liu P, Xing X, Ge X, Zhang S, Yu M, Jaeger T. Trustshadow: secure execution of unmodified applications with arm TrustZone. Proceedings of the 15th annual international conference on mobile systems, applications, and services. ACM, 2017.

Hua Z, Gu J, Xia Y, Chen H, Zang B, Guan H. vtz: Virtualizing ARM Trustzone. In: Proceedings of the 26th usenix security symposium (USENIX Security 17). Vancouver, BC: USENIX Association; 2017. p. 541–56. http://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua.

Jang J, Choi C, Lee J, Kwak N, Lee S, Choi Y, Kang B. Privatezone: providing a private execution environment using arm TrustZone. IEEE Transactions on Dependable and Secure Computing, 15(5); 2018. p. 797–810.

Jang J, Kong S, Kim M, Kim D, Kang BB. Secret: secure channel between rich execution environment and trusted execution environment. Proceedings of 2015 annual network and distributed system security symposium (NDSS'15), 2015.

Johnson S., Savagaonkar U., Scarlata V., McKeen F., Rozas C. Technique for supporting multiple secure enclaves 2012, US Patent App. 12/972,406.

Kim Y, Kwon O, Jang J, Jin S, Baek H, Kang BB, Yoon H. On-demand bootstrapping mechanism for isolated cryptographic

operations on commodity accelerators. Comput Secur 2016;62:33–48. doi:10.1016/j.cose.2016.06.006.
Lee H, Moon H, Heo I, Jang D, Jang J, Kim K, Paek Y, Kang B. Ki-mon arm: a hardware-assisted event-triggered monitoring platform for mutable kernel object. IEEE Trans Depend Secure Comput 2017.
Li W, Li H, Chen H, Xia Y. Adattester: Secure online mobile advertisement attestation using TrustZone. In: Proceedings of the 13th annual international conference on mobile systems, applications, and services. ACM; 2015. p. 75–88.
Li W, Ma M, Han J, Xia Y, Zang B, Chu CK, Li T. Building trusted path on untrusted device drivers for mobile devices. In: Proceedings of 5th Asia-pacific workshop on systems. ACM; 2014. p. 8.
Liu H, Saroiu S, Wolman A, Raj H. Software abstractions for trusted sensors. In: Proceedings of the 10th international conference on Mobile systems, applications, and services. ACM; 2012. p. 365–78.
Machiry A, Gustafson E, Spensky C, Salls C, Stephens N, Wang R, Bianchi A, Choe YR, Kruegel C, Vigna G. Boomerang: Exploiting the semantic gap in trusted execution environments. Proceedings of the network and distributed system security symposium, 2017.
McCune JM, Parno BJ, Perrig A, Reiter MK, Isozaki H. Flicker: An execution infrastructure for tcb minimization, 42. ACM; 2008. p. 315–28.
Moon H, Lee H, Lee J, Kim K, Paek Y, Kang BB. Vigilare: Toward snoop-based kernel integrity monitor. In: Proceedings of the 2012 ACM conference on computer and communications security. New York, NY, USA: ACM; 2012. p. 28–37. doi:10.1145/2382196.2382202.
Ning Z, Zhang F. Ninja: Towards transparent tracing and debugging on ARM. In: Proceedings of the 26th USENIX security symposium (usenix security 17). Vancouver, BC: USENIX Association; 2017. p. 33–49.
Payne BD, Carbone M, Sharif M, Lee W. Lares: An architecture for secure active monitoring using virtualization. In: SP '08: Proceedings of the 2008 IEEE symposium on security and privacy (sp 2008). Washington, DC, USA: IEEE Computer Society; 2008. p. 233–47.
Perrig JMMA, Reiter MK. Safe passage for passwords and other sensitive data. Proceeding of the 16th annual network and distributed system security Symposium, 2009.
Petroni Jr NL, Fraser T, Molina J, Arbaugh WA. Copilot-a coprocessor-based kernel runtime integrity monitor.. In: Proceedings of the USENIX Security Symposium. San Diego, USA; 2004. p. 179–94.
Sensepost, 2017, http://www.sensepost.com/blog/9114.html.
Sierraware, 2017. http://www.openvirtualization.org/.
Samsung Electronics Co.. In: Technical Report. White paper: an overview of Samsung KNOX; 2017.
Santos N, Raj H, Saroiu S, Wolman A. Using arm TrustZone to build a trusted language runtime for mobile applications. In: Proceedings of the 19th international conference on architectural support for programming languages and operating systems. ACM; 2014. p. 67–80.
Seshadri A, Luk M, Qu N, Perrig A. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. SIGOPS Oper Syst Rev 2007;41(6):335–50. doi:10.1145/1323293.1294294.
Sharif MI, Lee W, Cui W, Lanzi A. Secure in-vm monitoring using hardware virtualization. In: Proceedings of the 16th ACM conference on computer and communications security. New York, NY, USA: ACM; 2009. p. 477–87. doi:10.1145/1653662.1653720.
Sun H, Sun K, Wang Y, Jing J. Trustotp: Transforming smartphones into secure one-time password tokens. In: Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM; 2015. p. 976–88.
Sun H, Sun K, Wang Y, Jing J, Jajodia S. Trustdump: Reliable memory acquisition on smartphones. In: Proceedings of the computer security-ESORICS 2014. Springer; 2014. p. 202–18.
Tpm main specification, http://trustedcomputinggroup.org/tpm-main-specification/ 2011.
Unlocking the motorola bootloader, 2017, http://bits-please.blogspot.com/2016/02/unlocking-motorola-bootloader.html.
Vasudevan A, Chaki S, Jia L, McCune J, Newsome J, Datta A. Design, implementation and verification of an extensible and modular hypervisor framework. In: Proceedings of the 2013 IEEE symposium on security and privacy (SP). IEEE; 2013. p. 430–44.
Writing an llvm pass, http://llvm.org/docs/WritingAnLLVMPass.html. 2017.
Wang Z, Jiang X, Cui W, Ning P. Countering kernel rootkits with lightweight hook protection. In: Proceedings of the 16th ACM conference on computer and communications security. ACM; 2009. p. 545–54.
Weiser S, Werner M. Sgxio: Generic trusted i/o path for intel sgx. In: Proceedings of the Seventh ACM on conference on data and application security and privacy. ACM; 2017. p. 261–8.
Wheeler D, Needham R. Tea, a tiny encryption algorithm. In: Fast software encryption. Springer; 1995. p. 363–6.
Zhou Z, Gligor VD, Newsome J, McCune JM. Building verifiable trusted path on commodity x86 computers. In: Proceedings of the 2012 IEEE symposium on security and privacy (SP). IEEE; 2012. p. 616–30.
Zhou Z, Yu M, Gligor VD. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In: Proceedings of the 2014 IEEE symposium on security and privacy. IEEE; 2014. p. 308–23.

Jinsoo Jang is a postdoctoral fellow at Korea Advanced Institute of Science and Technology (KAIST). He received his Ph.D. and M.S. in Information Security from KAIST in 2018 and 2014, and B.S. in Information and Computer Engineering from Ajou University in 2007. His research interest includes system security, particularly in the trusted execution environment (TEE).

Brent Byunghoon Kang is currently an associate professor at the GSIS (Graduate School of Information Security) at KAIST (Korea Advanced Institute of Science & Technology). Before KAIST, he has been with George Mason University as an associate professor in the Volgenau School of Engineering. Dr. Kang received his Ph.D. in Computer Science from the University of California at Berkeley, and M.S. from the University of Maryland at College Park, and B.S. from Seoul National University. He has been working on systems Security area including OS kernel integrity monitor, trusted execution environment, hardware-assisted security, botnet malware defense, and DNS analytics. He is currently a member of the IEEE, the USENIX and the ACM.