SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment

Jinsoo Jang, Sunjune Kong*, Minsu Kim, Daegyeong Kim, Brent Byunghoon Kang
Graduate School of Information Security
Korea Advanced Institute of Science and Technology
{jisjang, sunjune.kong, pshskms, daegyeong.kim, brentkang}@kaist.ac.kr

***Abstract*—ARM TrustZone, which provides a Trusted Execution Environment (TEE), normally plays a role in keeping security-sensitive resources safe. However, to properly control access to the resources, it is not enough to just isolate them from the Rich Execution Environment (REE). In addition to the isolation, secure communication should be guaranteed between security-critical resources in the TEE and legitimate REE processes that are permitted to use them. Even though there is a TEE security solution — namely, a kernel-integrity monitor — it aims to protect the REE kernel's static regions, not to secure communication between the REE and TEE.**

We propose SeCReT to ameliorate this problem. SeCReT is a framework that builds a secure channel between the REE and TEE by enabling REE processes to use session keys in the REE that is regarded as unsafe region. SeCReT provides the session key to a requestor process only when the requestor's code and control flow integrity are verified. To prevent the key from being exposed to an attacker who already compromised the REE kernel, SeCReT flushes the key from the memory every time the processor switches into kernel mode.

In this paper, we present the design and implementation of SeCReT to show how it protects the key in the REE. Our prototype is implemented on Arndale board, which offers a Cortex-A15 dual-core processor with TrustZone as its security extension. We performed a security analysis by using a kernel rootkit and also ran LMBench microbenchmark to evaluate the performance overhead imposed by SeCReT.

I. Introduction

A Trusted Execution Environment (TEE) that is isolated from a Rich Execution Environment (REE) aims to protect assets such as crypto keys and user credentials. As an example of a TEE for embedded devices, ARM TrustZone has been used to execute security-critical services [7], [9]. TrustZone's resources are physically isolated from the REE, such that attackers in the REE cannot access them directly. Any direct access from the REE to TrustZone's memory region is restricted by a hardware access-control mechanism (e.g., TZASC) [5], which guarantees the confidentiality of the resources it protects.

However, the current design of TrustZone's architecture does not authenticate the access to the resources in TrustZone. To access the resources in TrustZone, a legitimate process in the REE uses a communication channel between the REE and TrustZone. The channel is created with an REE process that synchronously invokes the specific instruction with the arguments written on domain-shared memory that is allocated in the REE. The legitimate REE process can send a request to (and get a response from) TrustZone through the channel. Unfortunately, the channel is vulnerable to an attacker with the REE's kernel privilege who attempts to access the resources in TrustZone. Such an attacker can create a malicious process that continuously sends requests with crafted arguments to discover the vulnerabilities of the resources in TrustZone.

To the best of our knowledge, no message-protection mechanism exists in TrustZone. Thus, the attacker could easily perform a man-in-the-middle attack to manipulate the messages transferred through the channel. Some security analysts performed a simple fuzzing test, exploiting the insecure communication channel, against a PlayReady — a TrustZone-based DRM service — on a Samsung Galaxy S3 phone to demonstrate such an attack "in the wild" [10]. The Motorola phone's bootloader was unlocked by sending a crafted message through the insecure channel to exploit the vulnerability of TrustZone's kernel [2], [3]. The assets in TrustZone, such as DRM and mobile-payment services, are closely related to the monetary interest. Thus, failing to protect them can lead to the potentially severe economic consequences, which not only affect the owner of the device, but also the manufacturer of devices and the service provider in TrustZone; the importance of enhancing the security of TrustZone cannot be overemphasized.

To minimize the attack surface and to protect the resources in TrustZone, we propose the following security requirements. First, access to the resources in TrustZone should be restricted to those on the access control list (ACL) maintained by TrustZone. The ACL contains a list of REE processes that are permitted to access TrustZone's resources. Second, the messages transferred through the channel should be signed for secure communication. To this end, a session key for signing the messages should be created with a trusted anchor to protect against attackers in the REE.

To satisfy these requirements, we propose SeCReT — a framework for building a secure channel between the REE

*Co-first author. Also, an employee of SAMSUNG ELECTRONICS (sunjune.kong@samsung.com).

NDSS '15, 8-11 February 2015, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2015.23189

and TrustZone. SeCReT creates a session key to sign the messages transferred during inter-domain communication. With the signed messages, the process in the REE can communicate with TrustZone securely. Because the session key is symmetrically assigned for both the REE and TrustZone, SeCReT protects the key in the REE that is regarded as an unsafe region.

SeCReT verifies every access to the memory page that contains the key in the REE. SeCReT's verification mechanism ensures that only predefined and legitimate user processes can read the key. To prevent an attacker from directly reading the memory page containing the key value, SeCReT flushes the page and shadows the register values every time the processor mode switches to kernel. However, an attacker might also manipulate the user-process' control flow to read the key value and copy it to an unmonitored memory region. SeCReT limits this kind of attack by verifying the registers that contain critical values such as the return-address. SeCReT performs this register verification whenever the process switches back and forth between user and kernel mode. To minimize performance degradation, these key-protection mechanisms are activated only when the process that is assigned with a session key exists in the REE.

We developed a prototype of SeCReT on Arndale board [6] that provides an ARM processor integrating TrustZone. We ran Linux 3.9.1 as the REE OS and assumed that active monitoring was running on TrustZone [14], [19] to protect the REE kernel's static region, such as the code and exception vector. To implement SeCReT, approximately 140 LoC for the REE's kernel code and 900 LoC for TrustZone's monitor code were added. To evaluate the prototype, we performed a security analysis by using a kernel rootkit that snapshots the REE's specific memory page to exfiltrate the session key. We measured the performance degradation for the entire system when SeCReT's key protection mechanism is activated. The specific overhead induced by the access control to the page that contains the session key was also evaluated.

SeCReT is similar to [16], [17], [20], [29] in that it protects security-critical resources. However, SeCReT capitalizes on existing hardware components rather than software stacks. Previous works utilized hypervisors and compiler techniques. TrustZone does not maintain additional page tables, such as the extended page tables (EPT) of the hypervisor [16], [20]. Moreover, it does not provide the instruction-level introspection that the compiler-based approach does [17]. The lack of these features leads to difficulties in deploying the security functions in TrustZone. However, it is worth noting that encryption is not necessarily required to prevent attackers from reading the security-critical part of the user process, provided that the critical data and code are properly classified and located in TrustZone. Furthermore, *TIMA*, an active monitoring system that protects the kernel's static region, is already available in TrustZone [14], [27], [30]. To protect trampoline code on the REE kernel — part of SeCReT's implementation — we assume that TrustZone-based active monitoring (e.g., *TIMA*) is available.

In summary, SeCReT makes the following contributions:

- This is the first work that generates a secure channel to reinforce the access control of the resources in TrustZone. We propose comprehensive steps designed to protect the session key that is used to sign the messages transferred between the REE and TrustZone.
- We provide a technique that enables TrustZone to protect the specific memory area from the compromised kernel in the REE. To this end, SeCReT makes the best use of an existing hardware component without adopting additional software stacks such as hypervisor. We believe SeCReT can easily coordinate with already deployed TrustZone-based security solutions such as *TIMA*.
- We introduce an interface for user processes to communicate with TrustZone, which prevents the intervention of an attacker even with kernel privileges. With the proposed interface, we can extend the usage of TrustZone more flexibly, not limited to simply providing a TEE.

The next section reviews ARM TrustZone and TrustZone-based active monitoring, constituting the trusted computing base (TCB) for SeCReT. Section III describes the attack models and assumptions. Section IV presents the design for SeCReT, which is comprised of trampoline code in the REE and access-control modules in TrustZone. Section V explains the details regarding the implementation of SeCReT on the ARM architecture. Section VI details the security analysis and performance evaluation of SeCReT. Section VII discusses the remaining issues pertaining to SeCReT, and Section VIII examines related work. We summarize and conclude in Section IX.

II. Background

SeCReT is a framework to build a secure communication channel between the REE and TrustZone. We implemented SeCReT based on the assumption that the REE kernel's static region and the inserted code in the REE's kernel-code area are protected. We believe this is a reasonable assumption because TrustZone-based active monitoring systems, such as *TIMA*, are already available today. In this section, we provide the background for ARM TrustZone and TrustZone-based active monitoring.

A. ARM Trustzone

ARM TrustZone is a hardware-based security extension to processors, which separates the system logically into two domains: the Rich Execution Environment (REE) and the Trusted Execution Environment (TEE). Each domain has banked registers and memory to run the domain-dedicated OS and software. Security-critical services, such as E-Ticket, Bring Your Own Device (BYOD), and Digital Rights Management (DRM), can be executed in the TEE. Processes in the REE place the arguments in domain-shared memory and invoke a *secure-monitor call (*SMC*)* to trigger one of the services in the TEE. The domain-shared memory is allocated physically in the REE's memory region. However, any process that runs in the TEE can access it by simply mapping the REE's memory address to a page table maintained in the TEE. This is possible because the ARM processor's architectural design accords the highest privilege to the TEE.

The security extension to the ARM processor provides a monitor mode that manages the switches between the two

TABLE I. EXAMPLE OF SECURITY-SENSITIVE INSTRUCTIONS AND THE SMC

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

domains. In most designs, the code that runs in monitor mode ensures that the state for each domain is saved safely and restored correctly after every domain switch. The *Non-Secure (NS)* bit from the *Secure Configuration Register (SCR)* indicates the current context of the domain. That is, if the NS bit is set, the context is in the REE. Otherwise, it is in the TEE. To switch domains, the monitor code changes the value of the NS bit after saving the context for the "from domain", and then restores the context for the "to domain." There are two ways to enter monitor mode from the REE: an SMC instruction can be explicitly invoked, guaranteeing synchronous entry into monitor mode; or, the control register can be set to enter monitor mode asynchronously with each occurrence of an interrupt or an external abort. In our prototype for SeCReT, we used only an SMC instruction to synchronously invoke the SeCReT code deployed in monitor mode.

The *Translation Table Base Register (TTBR)* contains the address for the page-table base address that is required to translate the address from virtual to physical. The TTBR is banked for each domain, so the monitor code uses the TTBR in the TEE. However, the monitor code can access any address in the REE by constructing a map for the address in the REE for the page table in the TEE. Because the TTBR is unique for each process, it was adopted in the implementation of SeCReT. The details regarding the TTBR's usage are described in Section IV.

B. TrustZone-based Active Monitoring

TrustZone-based active monitoring [14], [19], such as *TIMA*, protects the kernel's static region in the REE. To prevent the direct modification of kernel code and static data, active monitoring sets *write-protect (WP)* bits from the page descriptors for pages that contain the objects to be protected. Active monitoring not only write-protects the code and static data, such as system call-tables and exception vectors, but page tables as well. Therefore, any update to the page table causes a data-abort exception. This exception is brought to TrustZone through the inserted SMC instructions on the exception handlers. The active-monitoring agent for TrustZone verifies the exception. If the root cause of the exception is a legitimate update to the page table, the agent emulates it. In other words, the page-table update for the REE is available only in TrustZone, provided that active monitoring is running.

Attackers can exploit privileged instructions that disable the MMU or update the page-table base register. Active monitoring replaces all of these privileged instructions with an SMC such that any execution of security-sensitive operations is verified and emulated in TrustZone. As can be seen in Table I, the size of the security-sensitive instructions and the SMC are equivalent to four bytes. Thus, replacing all of the security-sensitive instructions is feasible. In addition to replacing the instructions, active monitoring ensures that newly allocated pages are set with *Privileged Execute Never (PXN)* bits by default. As a result, any attempt to run security-sensitive instructions on the newly allocated page causes an exception that is also brought to TrustZone. All such enforcements from active monitoring guarantee that the REE kernel's static region remains immutable.

III. ATTACK MODEL AND ASSUMPTIONS

A. Trust Base of SeCReT

The prototype of SeCReT is implemented on ARM-based SoC hardware with security extension. We assume that commercially available security features such as Secure Boot [12] and active monitoring are already activated as a base-line defense against attacks. Therefore, only the authorized OS and applications for both the REE and TEE are loaded during the startup process. Because the presence of a malicious manufacturer is not a consideration for our attack model, intentionally implanted vulnerabilities or malwares are not assumed to be loaded. During runtime, the kernel's static region in the REE is protected by active monitoring in the TEE. We also assume that a list of REE processes permitted to access TEE resources is predefined and safely maintained in the TEE. Furthermore, all valuable resources that could be the target of attacks are properly classified and stored in the TEE.

B. The Weakness of Communication Channel for TrustZone

When an REE process employs resources from the TEE, a communication channel is necessary for the transmission of messages between the two domains. The channel is regarded simply as the domain-shared memory that can be accessed from both domains. Although the shared memory is allocated to the REE memory area, it is still accessible by the TEE. The process that volunteers to use the resources in the TEE places arguments, such as the identity number of the TEE services, in the shared memory before opening the TrustZone device driver to invoke the SMC instruction [10]. Once the SMC instruction is executed, the processor mode is switched to monitor mode and the address for the shared memory is delivered as a parameter of the SMC instruction. The code that runs in monitor mode retrieves the address of the shared memory from this parameter value and maps the address to the page table in the TEE. The arguments procured from the shared memory might contain the information that is required to dispatch the request to the proper service in the TEE. As a motivation for this paper, we focused on some weaknesses of the channel established under the compromised kernel in the REE. We state these weaknesses as follows.

First, it is difficult to verify whether the request from the REE — equivalent to invoking an SMC instruction with parameters — originated from authenticated processes. An attacker with kernel privileges is free to call the SMC instruction with maliciously crafted parameters.

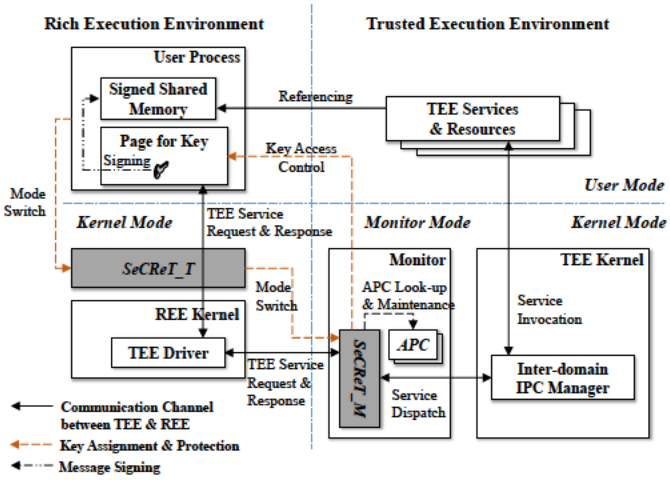

Fig. 1. Design overview: SeCReT_T, the trampoline codes in the REE and SeCReT_M, the monitor code in the TEE constitute a framework of SeCReT which protects a session key in the REE to establish a secure communication channel.

Second, the domain-shared memory can be accessed, modified, and relocated by attackers. To the best of our knowledge, there is no mechanism currently available to verify the integrity of domain-shared memory.

C. Attack Model

We assume that an attacker with kernel privileges in the REE can do everything except manipulate the kernel's static region, owing to the presence of active monitoring. The main purpose of the attack is to exfiltrate the critical resources that are isolated in the TEE. However, the attacker cannot directly access the memory in the TEE because of a hardware access-control mechanism such as *TrustZone Address Space Controller (TZASC)* [5].

Therefore, the attacker must perform a brute-force attack to discover the vulnerabilities to services in the TEE. The weaknesses addressed in Section III-B make such an attack possible. An attacker can create a malicious process that continuously invokes the SMC instructions with crafted parameters. In the TEE, authentication mechanisms can be as simple as checking the *universally unique identifier (UUID)* of the requestor. Unfortunately, an attacker can easily bypass such an authentication mechanism by manipulating the data structure related to the UUID. Attackers can also compromise the channel for a legitimate process by redirecting the function pointer for a TrustZone device driver to a malicious code that hooks every invocation of an SMC instruction. Hooked messages can be analyzed to retrieve useful information for performing a brute-force attack effectively, or they can be adjusted to suit the attacker's needs. Furthermore, attackers might also perform a *Denial-of-Service (DoS)* attack by blocking the requests to TrustZone, this type of attack was exempted from our attack model.

IV. Design

A. Overview

The purpose of SeCReT is to offer access control to critical resources in the TEE by assigning a session key that can be used in the REE. We assume that a list of the legitimate processes that are granted access the resources in the TEE is predefined and maintained in the TEE. The assumption is reasonable because, normally, a service provider does not only deploy a TEE service in TrustZone, but also develops and provides an REE application corresponding to the service. The process list contains the hash of the code area for each legitimate process. Either it is pre-calculated and deployed in the TEE at the time of the device's manufacturing, or it is calculated during the secure boot. SeCReT uses the hash to determine whether the process requesting access to the TEE resources is legitimate. SeCReT also creates an Active Process Context (APC) in the TEE. The APC contains information about legitimate processes such as a page-table base address in the REE, the code hash per page, and the identification of the TEE resource to which the process is granted access. APCs are linked to one another, and together they form an Active Process List (APL) that is also maintained by SeCReT.

The saved APCs are referenced in allocating the session key or to make access-control decisions concerning the critical resources in the TEE. The REE process that requests the use resources in the TEE sends SeCReT a request asking it to create a session key. To handle this request, SeCReT allocates a pinned page in the REE and saves the address of the page to the APC. The session key is also created and saved in the APC, but it is not written to the page immediately. SeCReT writes the key value to the page only at the time when the process accesses the page to use the key. To protect it from attack, the key is only visible when the process runs in user mode, and it is flushed before every mode switch to kernel.

As can be seen in Figure 1, SeCReT consists of two components: the SeCReT trampoline (SeCReT_T) in the REE, and SeCReT's monitor code (SeCReT_M) in the TEE. SeCReT_T is inserted in the REE's kernel code to redirect the control flow to SeCReT_M in the event that there are important system events related to session-key maintenance. SeCReT_T simply invokes the SMC instruction to synchronously enter monitor mode. SeCReT_M, which runs in monitor mode with the REE's context, is responsible for the main functions of SeCReT such as APC updates, session-key creation, and access control against the key. The details of SeCReT's design and the key-maintenance procedure are addressed in the following sections.

B. Active Process Context

The APC is a data structure containing the information necessary for session-key maintenance and protection. SeCReT creates the APC for each process that is permitted to access the critical resources in the TEE. Each APC is linked to another APC forming linked list we call the APL. The link to the data structure that contains the hash for each code-page and the VA information for the start- and end-code of the process is also stored in the APC. This is referenced when the integrity of the process' code is verified during the lifecycle for the session-key management. The TTBR, similar to CR3 in Intel's architecture, is used as a key to traverse the APL because it is uniquely assigned to each process. In ARM architecture, there are two TTBRs available, TTBR0 and TTBR1. Depending on the setting of the N bit for the *Translation Table Base Control Register (TTBCR)*, one or both TTBRs can be used for

translating the address. SeCReT uses TTBR0 exclusively by setting the TTBCR.N to 0. This setting is unchangeable in the REE on account of TrustZone's active monitoring. The session key's value and the VA for the memory page that stores the key in the REE are also elements of the APC. The value of the session key is randomly generated on demand. The APC also provides a link to the information concerning critical resources that are accessible in the TEE. The definition of a critical resource, which is a topic beyond the scope of this paper, depends on the developers' design criteria in implementing TrustZone-based applications.

To protect the session key, SeCReT maintains some flags, whitelist and shadow stacks in the APC as well. The key_request_flag indicates whether the access to the session key is legitimate. The mode_switch_flag and shadow_stack are updated and verified to enforce the coarse-grained control-flow of the process using the session key. The whitelist_sighand registers addresses of the signal handlers that are exempted from shadow-stack verification. We explain the usage of those elements in depth in Section IV-D1 and IV-D2.

C. Session Key Lifecycle

Secure Boot Maintaining the session key entails consecutive interactions between the REE process and SeCReT. During boot time, our SeCReT prototype temporally loads the process that is granted access to the resources in the TEE, and calculates the hash of the code based on the granularity of the small page, which is 4KB in ARMv-7. The hash values are transferred to SeCReT_M, and saved in the TEE. This is adequately safe, because we assume that Secure Boot is enabled. However, calculating the hash at boot time might lead to performance degradation. Therefore, as an alternative design, the pre-calculated hash values from REE processes can be stored in the TEE in advance, along with security-sensitive services that run in the TEE. This might be accomplished during the manufacturing of the device.

New Process Execution During the system's runtime, SeCReT verifies whether newly executed processes in the REE have permission to access the resources in the TEE. To this end, SeCReT simply refers to the process descriptor (e.g., *task_struct* in Linux) to check the name of the currently loaded process. However, an attacker can easily modify the values in the task descriptor. Thus, alternatively, as with Patagonix [24], we can set the *Non-Executable (NX)* bit for all page descriptors of the newly executed process and check the hash from the first code-page when the fault occurs due to the execution of the process. However, in our prototype for SeCReT, immediately prior to assigning the session key, the hashes for all code-pages that are present in the memory are computed and compared with the whitelist in the TEE. This method is simple, but it is comparatively faster than a hash-check to verify the new process. If the new process is found in the whitelist, SeCReT creates an APC for the process and updates it with information such as the TTBR and the link to the pre-calculated hash information corresponding to the process.

Key-Assignment Request The REE process must send a request to SeCReT for the creation of the session key. The key is used to authenticate the communication. Once SeCReT receives a request for the creation of a key, it first traverses the APL to find the APC corresponding to the process asking for the key. Because each REE process is assigned a unique TTBR, the TTBR is used as a key to search for the proper APC in the APL. If the APC is found, SeCReT allocates a page-aligned small page for storing the session key. The virtual address (VA) for the allocated page in the REE is stored in the session_key_address element of the APC. The access permission for the page is set to *No Access (NA)*, which is neither writable nor readable. The session key is also created and stored in the session_key_value element of the APC. However, the key is not written to the allocated page in the REE at this time because SeCReT makes the key visible only at the moment when the process accesses the page to use the key.

Session-Key Protection Access control to the session key is based on the occurrence of a data-abort exception and SeCReT's handling of it. Access to the page that is allocated in the step prior to the key's creation causes the data-abort exception because the access permission for the page is set to NA by default. When the exception occurs, the processor mode is changed to kernel, and the control flow for the current process is redirected to an exception handler for data abort. The SeCReT_T — that is, the trampoline code inserted to the starting point of the exception handlers — causes the control flow to jump to SeCReT_M. SeCReT_M first retrieves the REE process' APC from the APL. Based on the information in the APC, SeCReT determines whether an exception has occurred from a legitimate process' attempt to use the key.

The integrity of the code and the control flow is also checked for the process. If no intervention from an attacker is found, SeCReT writes the session key to the page in the REE and changes the access permission for the page to readable. Additionally, SeCReT changes the control flow, returning to user mode directly without executing the remaining exception handler routines in the REE kernel. Therefore, the process can obtain the key by re-executing the instruction that failed because of the page's initial NA permission. To conceal the session key from an attacker, the key is flushed from the page when the processor mode switches to kernel. The details for the access-control mechanism relating to session-key protection are described in Section IV-D1.

Process Termination Every process-termination event also triggers SeCReT. If the process being terminated has the APC, SeCReT frees the memory page that was allocated for the session key. The APC containing the information for the process is also removed from the APL. An attacker might reuse SeCReT_T, the trampoline code, to remove the victim process' APC from the APL. This would constitute a DoS, but DoS attacks are exempted from our attack model.

D. Session-Key Protection Mechanism

1) Access Control for the Session Key: The session key is readable only when a legitimate process runs in user mode and that process accesses the page allocated to contain the key. This amounts to an exclusive key assignment at moment the process requires it. One effective way to acquire the key synchronously might be for the user process to make a direct request to SeCReT that the session key be made readable on the page. However, this is not possible because the only

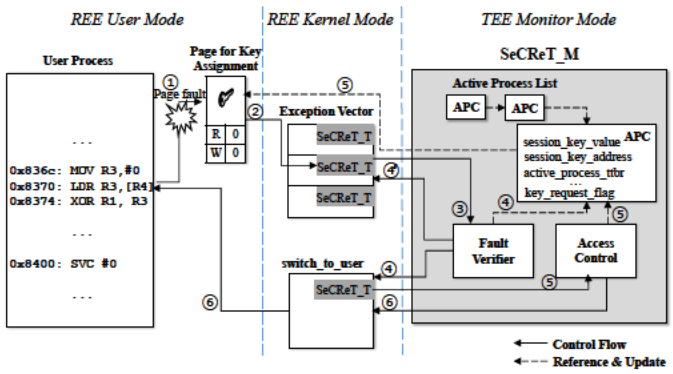

Fig. 2. Key-assignment mechanism: 1. The process in the REE accesses the page that has its access permission set to NA. 2. The control flow moves to SeCReT_T because of the occurrence of a data abort. 3. Monitor mode is entered to invoke SeCReT_M. The data-abort exception is verified to determine whether it is because of the access to the session key. 4'. The normal exception-handler routine is executed in the REE if the exception is unrelated to the session key. 4. The key_request_flag is set, and the control flow changes to the switch_to_user code. 5. SeCReT_M is invoked to make the session key visible if the key_request_flag is set and to clear the flag before leaving monitor mode. 6. The mechanism returns to user mode.

way to synchronously enter monitor mode is by invoking an SMC instruction in kernel mode. Unlike a hypervisor's hypercall, which is available in both user and kernel mode, the behavior resulting from an SMC instruction in user mode is undefined [1]. Hence, as an indirect way to send a request from user mode to TrustZone, we set the access permission to the user-memory page for a session-key assignment to NA. Additionally, we inserted SeCReT_T to the starting point of the exception handler.

Processes accessing pages with NA permission cause a data-abort exception, one type of exception available in ARM's architecture. To handle the exception, the processor mode switches to kernel mode and the control flow for the process moves to the exception vector containing the address of the exception handlers. From this vector, the control flow branches to the corresponding exception handler — in this case, the data-abort handler. SeCReT_T is inserted to the starting point of this data-abort handler and changes the control flow to SeCReT_M in monitor mode. Running SeCReT_T in kernel mode is reasonably safe because active monitoring protects the kernel's static region.

Once the processor mode switches to monitor mode, SeCReT_M is triggered. SeCReT_M checks the value of *Link Register (LR)* that contains the return address when the mode switches back to the REE. The LR helps to verify whether entry to monitor mode originated from the address of the kernel-code region. Furthermore, SeCReT_M parses the input parameter that indicates the exception type that triggered SeCReT_M. Distinguishing the type of the exception is important because SeCReT_T is not only inserted to the data-abort exception-handler, but to every exception handler that causes the mode to switch from user to kernel. We explain the reason for this with describing the key-flush mechanism.

After verifying the LR and the input parameter, SeCReT_M traverses the APL to search for the APC by using TTBR as a key. If SeCReT_M discovers the corresponding APC, it retrieves the value of the session_key_address element and compares that value to a Data Fault Address Register (DFAR), which holds the VA for the faulting address that caused the data-abort exception. If the DFAR falls within the range of addresses that contain the session key, SeCReT_M sets the process' key_request_flag, indicating that the key request is valid. After setting the flag, SeCReT_M changes the LR to the starting address of the switch-to-user code that is part of the kernel code handling the switch to user mode. This ensures that the control flow for the process returns directly to user mode without executing the remaining exception-handler routines that an attacker could compromise.

SeCReT_T is also inserted to the switch-to-user code. Consequently, every return to user mode is coupled with an entry into monitor mode, triggering SeCReT_M. SeCReT_M validates the LR as before and determines whether the APC corresponding to the current TTBR exists in the APL. If it does, SeCReT_M also checks the value of the key_request_flag. The flag is set only if the last switch to kernel mode was from a legitimate key request. To make the key readable, SeCReT_M first validates the integrity of the code pages that are currently loaded for the process. This is done by calculating the hash for the each page and comparing it to the hash stored in the APC. After the code is validated, SeCReT_M retrieves the session key from the APC and writes it to the page that caused the fault. Additionally, SeCReT_M sets the access permission for the page from NA to read-only. This way, the process can read the key by retrying the instruction that failed when access permission to the page was set to NA. The key_request_flag is also cleared before returning to the REE.

Finally, the user context is restored and the mode switches to user by executing the remaining switch-to-user code. Because the interrupt is always disabled from the starting point of switch-to-user code, an attacker cannot intervene during the key-assignment process. By restoring the saved user-mode context from the *Saved Program Status Register (SPSR)*, the interrupt is again enabled when the mode switch to user is complete.

An attacker can try to retrieve the session key by fooling SeCReT_M — which is dependent on the LR values, input parameters, and DFAR to set the key_request_flag. For instance, an attacker can invoke the SMC instruction by manipulating an LR value, an input parameter, or the DFAR. However, such an attack would not be feasible because attackers cannot write the value to the DFAR. A privileged instruction is needed to modify it. As stated in Section II-B, by replacing security-sensitive privileged-instructions with SMC in the kernel code and by setting the PXN bit for newly allocated pages, we can prevent attackers from executing privileged instructions. Even if attackers successfully set the key_request_flag to deceive SeCReT_M for the key assignments, they must still set the LR to the address of the legitimate kernel code — that is, the switch-to-user code. Once the control flow is changed from SeCReT_M to switch-to-user, it returns to user mode immediately with no chance of an attacker's intervention. The overall process for the session-key assignment is summarized in Figure 2.

Key-Flush Mechanism The session key should be visible only in user mode to be protected from an attacker compromising the kernel. To ensure this, SeCReT flushes the key after every event that triggers the switch to kernel mode. The

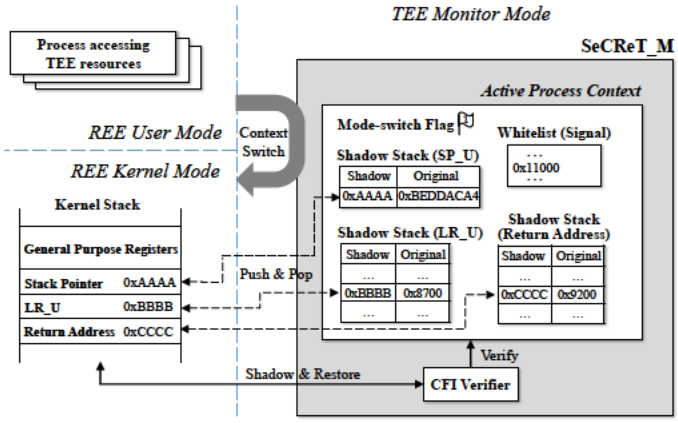

Fig. 3. Coarse-grained control-flow integrity: SeCReT verifies, shadows and restores the return addresses to enforce the control flow integrity for every mode switch between user and kernel. The exceptional cases such as the returning to signal handlers are also handled by SeCReT.

event is the occurrence of exceptions in user mode, including the invocation of the system calls. In ARM architecture, there are several exceptions that cause the switch to kernel mode such as undefined instructions, supervisor calls (caused by a system-call invocation), prefetch abort, data abort, and IRQ and FIQ interrupts. To intervene before every switch to kernel mode, SeCReT_T is inserted at the starting point of the every exception handler. The version of Linux running in the REE's OS in our prototype has separate exception handlers in each of the two modes. Thus, we patched only the handlers for user-mode exceptions. Because SeCReT_T is inserted, SeCReT_M is always executed prior to the REE kernel's exception-handler routines. SeCReT_M retrieves the APC for the current process. If the process was assigned the session key before, SeCReT_M flushes the session key from the REE and sets the access permission of the page to NA based on the information from the APC. In our prototype, to set the permission, we dynamically map the physical address of the page in the REE.

2) Coarse-Grained Control-Flow Integrity: The session key is only visible for an instant when the process runs in user mode, and it is flushed with the switch to kernel mode. Thus, the key is not visible until the next data abort occurs from a legitimate access to the key. Under these circumstances, to exfiltrate the key, the attacker must have the process copy the key and paste it to a memory area that is not protected by SeCReT. To achieve this, the attacker must manipulate the process' code area. However, this is easily detectable because SeCReT always verifies the code's integrity before assigning the session key.

Alternatively, an attacker might try to compromise the control flow of the process when it runs in user mode, similar to a *Return-Oriented Programming (ROP)* attack. For instance, an attacker can gather and assemble the ROP gadgets needed to reveal the session key and exfiltrate it from the protected area. At least one gadget accessing the session key should be found from the process's code area because SeCReT assigns the key only if the page for the key-assignment is accessed by an instruction that resides in the legitimate process' code area. To launch the attack, an attacker might set up the user stack based on the assembled gadgets and change the return address for the switch to user mode in order to trigger the first gadget. Because the user-process' contexts, such as the LR and *Stack Pointer (SP)*, are saved in kernel stack after every switch to kernel mode, an attacker with kernel privileges can manipulate them without exploiting any vulnerability in the process.

To prevent the attack described above, as shown in Figure 3, SeCReT protects important register values that can be exploited by attackers. Before each entry into kernel mode, SeCReT_M copies the original value of the LR that is used in user mode (LR_U) and the return address that is referenced later with the switch back to user mode. Each copy of the values is pushed into the corresponding shadow stacks that are maintained in the APC. After pushing the values, SeCReT also modifies the kernel stack in the REE to shadow the LR_U and return-address by replacing them with arbitrary values. The register that contains a stack pointer of user mode (SP_U) is also shadowed in the same way. The purpose of the shadowing is to hide the current context in user mode. Because of the shadowing, an attacker will not know the latest instruction that was executed in user mode before the exception occurred. We expect that by shadowing, the attacker's ability to set up the ROP stack is limited because it is difficult to find the saved return addresses from the user stack without knowing the current context of the execution.

The stored values in the shadow stacks are popped and written back into the kernel stack in the REE with the switch back to user mode. As a result, the switch-to-user code can restore the appropriate context for switching to user mode. Before writing the values back to the kernel stack, SeCReT_M inspects the shadowed values of the kernel stack to determine whether there were any adjustment to the return address during the execution of kernel's exception-handler routines. Adjusting the return address is normal kernel behavior. For instance, the handler for floating-point exceptions adjusts the return-address to permit the process to retry the failed instruction. The adjustment subtracts at most 0x8 from the original value [1]. Thus, SeCReT_M decides whether the adjustment to the return-address is legitimate by its size. A legitimate adjustment is equal to or smaller than 0x8. If the adjustment is normal, SeCReT_M also adjusts the return address stored in the shadow stack before it is written back into the kernel stack by SeCReT.

In addition to the return addresses, SeCReT also shadows the address of the faulting instruction which is input to the kernel's page-fault handler, and saves the original value of the address in the APC. Normally, shadowing the address can cause malfunction of the page-fault handler because the handler needs to know the address to allocate a new page. In our prototype for SeCReT, we just masked the least significant 12 bits of the address to enforce the shadowed to be page-aligned. This page-aligned masking ensures that the kernel can properly handle the page fault. Although we are not fully shadowing the faulting address, this approach is still reasonable because page faults occur asynchronously, which makes attackers difficult to consistently identify a unique stack layout. Moreover, we assume that active monitoring is running, so page faults will be handled in TrustZone. Thus, active monitoring can cooperate with SeCReT to fully shadow the address; active monitoring will refer to the original value in the APC during the fault handling routine.

Exceptional Cases Simply validating the return address by restricting the change to 0x8 will not work for signal-handling routines. Normally, before the return to user mode from the exception-handling routine, the kernel can change the control flow to invoke the registered handler that is corresponding to the signal that is currently raised (if it exists). To change the control flow, the kernel first saves the original kernel stack in the user-memory area. Moreover, the kernel sets up a signal frame that is a new kernel stack. The signal frame contains the address for the registered signal handler as a new return address. Also, in the signal frame, the address for the original kernel stack is saved as a new user stack, and the address for the code to invoke the *sigreturn* system call is saved as the new LR_U. Consequently, when the execution of the signal handler in user mode is complete, the *sigreturn* system call is invoked and the original kernel stack is restored.

To handle this exceptional case, SeCReT inspects the invocation of some system calls such as *sigaction* and *sigreturn*. The *sigaction* system call registers the signal number and the address of user-process' signal handler into a kernel data structure such as *sighand_struct*. In ARM architecture, an SVC instruction triggers the system call invoking the supervisor-call exception synchronously. Once the exception occurs, SeCReT checks the system-call number delivered by one of general-purpose registers. If the invocation of the *sigaction* system call is noticed, SeCReT_M retrieves the address of the signal handler from the system call's parameters, and registers it in the whitelist_sighand maintained in the APC. Thus, provided that the return address to user mode exists in the whitelist_sighand, SeCReT exempts it in verifying the shadow stack.

Attack Vectors on SeCReT Doubtless, whitelisting the specific address constitutes a vulnerability to SeCReT's security because an attacker can deceive SeCReT by reusing SeCReT_T. That is, attackers can invoke SeCReT_T with parameters that contain the system-call number for *sigaction* and the address where they plan to enter and trigger the first ROP gadget. However, an attack that reuses the kernel code is not feasible because we maintain a mode_switch_flag in the APC. SeCReT_M sets the flag upon entering kernel mode and clears it on the return to user mode. In other words, the flag is always set when in kernel mode in the REE. As a result, if the flag is set when SeCReT_M receives a request from *sigaction*, the request is simply ignored. Furthermore, because SeCReT checks the integrity of the user-process's code area before whitelisting, an attacker cannot manipulate the code to send a decoy request from *sigaction*.

The signal frame can be an attack vector as well because attackers can manipulate it to change the control flow arbitrarily. For example, because the value of the LR_U is in the frame, an attacker can set the address for the first gadget instead of *sigreturn*. To protect the key from this type of attack, SeCReT ignores all key-assignment requests whenever the signal handler runs in user mode. In other words, until the invocation of a legitimate *sigreturn* system call is noticed by SeCReT, neither attackers nor legitimate processes can retrieve the session key. This might be a limitation when developers design applications in SeCReT's framework. Developers should design applications such that they do not use the session key in the signal-handler routine. Despite this limitation, however, we believe that SeCReT will continue to motivate them to design applications carefully because SeCReT reinforces the protection of the critical resources in the TEE.

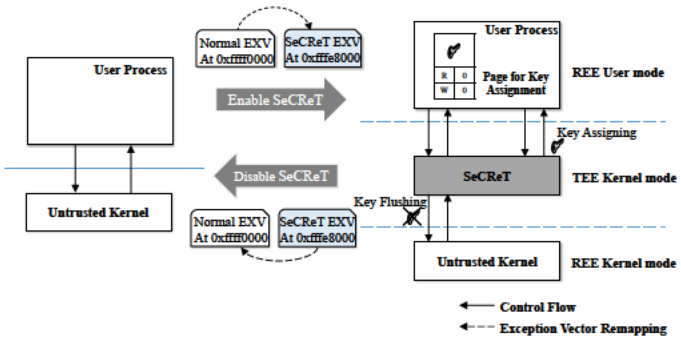

Fig. 4. Exception-vector remapping enables SeCReT's key-protection mechanism to be activated flexibly; the execution of the first process that has a permission to access to the TEE enables the mechanism, and the termination of the last process granted this permission disables it.

SeCReT's register protection is effective only if the legitimate kernel code for switching modes patched with SeCReT_T is used. The switch to kernel mode guarantees this because the exception handler that handles switches to kernel mode is immutable and protected with the active monitoring. On the contrary, however, an attacker can use the malicious code instead of the legitimate kernel code with the switch to user mode. In such a case, SeCReT's register-protection mechanism is bypassed and the attacker can directly manipulate the return address for the mode switch to trigger an ROP attack. From the perspective of the attacker, the direct manipulation of the current return address makes the attack even easier than exploiting the return address saved in the user stack. Fortunately, the mode_switch_flag helps to detect the usage of malicious code during the switch to user mode. It is worth reiterating that the flag is set and cleared with switches to kernel mode and user mode, respectively. If the flag is already set when SeCReT_M attempts to set it upon switching to kernel mode, the last entry to user mode was not done legitimately.

Any attempt to reuse the legitimate SeCReT_T to push the malicious address into the shadow stack before switching back to user mode is prevented by checking the status of the mode_switch_flag before updating the shadow stack. As explained in the section IV-D1 describing the key-assignment mechanism, SeCReT_M always checks the LR to verify the origin of the invocation of the SMC instruction that led to the current entry into monitor mode. This ensures that the attacker cannot fabricate the value of the flag by maliciously invoking the SMC instruction mimicking the legitimate SeCReT_T.

E. Exception-Vector Remapping

To protect the session key, SeCReT interposes between user and kernel modes. The control flow should always be redirected to monitor mode with every switch in modes in order to flush the assigned key and to check the integrity of the control flow. Even if the process that is responsible for the current switch in modes is unrelated to the session key, the control flow is nonetheless redirected to monitor mode. This is because, until the APC in the TEE is checked, SeCReT cannot determine whether the current process was

assigned the session key beforehand. This additional switch in modes causes performance degradation to the entire system. However, SeCReT cannot selectively switch the mode only for the process that assigned the key because the information for the session key is only accessible in the TEE. Thus, as an alternative solution, we enable SeCReT's access control and key-flush mechanism only if the process assigned with the session key exists in the REE. To this end, in addition to the normal exception vector, we create the SeCReT exception vector (SeCReT_EXV).

In Linux, early in the boot time, a 4KB page is allocated for locating an exception vector with other handler codes such as *sigreturn*. The virtual address for the page that is equivalent to the address of the exception vector is fixed at 0xffff0000 in Linux for ARM processors. However, the address can be changed based on the setting in the *System Control Register (SCTLR)*. We utilized this feature to dynamically remap the exception vectors.

The exception-vector remapping and the change of the control flow due to the remapping are described in Figure 4. The remapping is triggered with the occurrence of events such as process executions and terminations. The remapping from normal to SeCReT_EXV is done when the first process is executed that has permission to access the resources in the TEE. Subsequently, the SeCReT_EXV remains enabled until the last process granted this permission is terminated. With every process termination, SeCReT not only deletes the APC corresponding to the terminated process, but also checks whether the APC is the final one in the APL. If it is, SeCReT remaps the exception vector back to the original exception vector. When the SeCReT_EXV is enabled, performance degradation from the additional mode switching for every process remains a problem. However, we believe that the overall degradation is alleviated by this remapping process at least when no process is running with access to the resources in the TEE.

F. Trusted-Computing Base for SeCReT

SeCReT depends on active monitoring as part of its *Trusted Computing Base (TCB)*. Active monitoring protects the kernel code in REE kernel's static region. This ensures the immutability of SeCReT_T, which is inserted to the part of the kernel code in the REE containing exception vectors and handlers. In addition to protecting the kernel code, active monitoring protects the registers that play a significant role such as traversing the APL and distinguishing exception types. The TTBR is used as an identifier to retrieve the APC from the APL because each process has a unique TTBR. SeCReT checks the value of the *Data Fault Status Register (DFSR)* and the *Data Fault Address Register (DFAR)* to demarcate a normal data abort from a data abort specifically occurring because of the demand for the session key. The *Vector Base Address Register (VBAR)* contains the address of the SeCReT_EXV selectively enabled based on the existence of a process with permission to access the TEE. To change the values of those registers, an attacker must run privileged instructions the execution of which is prevented by active monitoring.

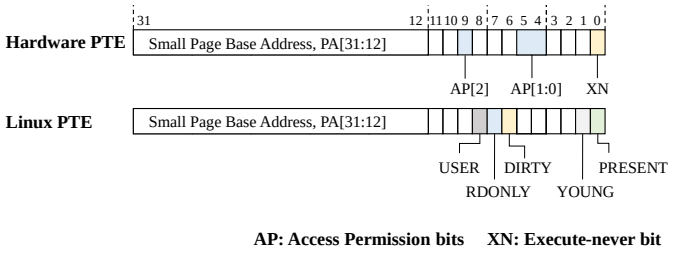

Fig. 5. Hardware page-table descriptor and Linux page-table descriptor

V. Implementation

We implemented a prototype for the SeCReT framework on Arndale board, offering a Cortex-A15 dual-core processor. We used Linux version 3.9.1 as an REE OS and Sierra TEE software [11] as a PoC in the TEE. For the SeCReT components in the REE, 140 LoC were added to the Linux kernel code. In monitor code, approximately 900 LoC were added to implement the SeCReT components in the TEE. In this section, we describe the implementation details for SeCReT based on each domain.

A. SeCReT Components in the REE

In ARM architecture, the exception vector that holds the address for each exception can be located flexibly. The V-bit for SCTLR selects the base address for the exception vector. For example, if the value of the bit is set, the address is fixed at 0xffff0000. Otherwise, it is at 0x00000000. However, if a security extension such as TrustZone is enabled and the V-bit is cleared, the exception vector can be re-mapped to the address held by the *Vector Base Address Register (VBAR)*. In this case, each exception handler is invoked by branching to the address that is computed by adding predefined offsets for each exception to the address in the VBAR. We utilized the VBAR to hold the base address for the SeCReT_EXV. To enable the SeCReT_EXV, the V-bit for SCTLR is cleared when the first process with permission to access the TEE is executed. Moreover, the bit is set to enable the normal exception vector again when the last APC is deleted, signaling that there are no processes remaining with permission to access the TEE. Because configuring the SCTLR requires the privileged instructions that are restricted from execution in the REE by active monitoring, only SeCReT_M in the TEE can dynamically change the SCTLR's configuration.

The SeCReT_EXV is a newly created exception vector that provides addressing for the exception handlers patched with SeCReT_T. The normal exception vector is created during the boot time. To create the SeCReT_EXV, we inserted SeCReT_T to the exception handlers in the *entry-armv.S* and *entry-common.S* files. Furthermore, we patched the *mmu.c* and *traps.c* files to allocate a small page (4KB) where we composed the SeCReT_EXV. Therefore, new vector code that branches to the exception handlers patched with SeCReT_T is copied from the *entry** files and placed into the new page during the boot time. In our prototype for SeCReT, the SeCReT_EXV is mapped to 0xfffe8000 arbitrarily, and we assume that it is protected by active monitoring.

Furthermore, SeCReT_T is statically inserted to the kernel code that handles process execution and termination. To run

TABLE II. SECRET'S ATTACK SURFACE AND DEFENSE MECHANISM

Attack Surfaces	Defense Mechanisms
Process-code modification	Hash check for present code pages
Control-flow manipulation	Shadow-stack verification
Reusing SeCReT_T for shadow-stack manipulation - mimicking *sigaction* system call - Decoy switch to kernel mode	mode_switch_flag verification
Trampoline bypass when switches to user mode	mode_switch_flag verification
Memory snapshot for key exfiltration	Memory page flushing coordinating with active monitoring

TABLE III. LMBENCH LATENCY MICROBENCHMARK RESULTS (IN MICROSECONDS.)

	Linux	SeCReT enabled	Overhead
Null	0.27	1.06	3.9259x
Open/Close	5.43	8.83	1.6264x
Read	0.33	1.23	3.7273x
Write	0.42	1.57	3.7381x
Fork	147.78	174.66	1.1819x
Fork/Exec	160.32	189.03	1.1791x

SeCReT_M with every execution of a new process, we invoked SeCReT_T after *execve* is complete. With this invocation, SeCReT_M can check the *task_struct* of the current process to decide whether a new APC must be created for the process. In our prototype, we simply compare the value of *comm*, which contains the name of the executable, in *task_struct* with the whitelist which defines the legitimate processes that have permission to access the resources in the TEE. With every process termination that invokes a *do_exit* system call, SeCReT_T is also invoked in order for SeCReT_M to delete the APC corresponding to the process that is currently terminated.

B. SeCReT Components in the TEE

When SeCReT_T invokes the SMC instruction, it creates an SMC exception. At the same time, monitor mode is entered, and the exception is handled by the SMC exception handler in monitor mode. Similar to the REE, there are several exceptions, such as FIQ, IRQ, data abort, and SMC, that can occur in monitor mode. The Monitor Vector Base Address Register (MVBAR) holds the base address for the exception vector. Therefore, to handle each exception, the control flow branches to the address corresponding to the MVBAR value added by predefined offsets for each exception.

Because in our prototype every entry to monitor mode happens synchronously by invoking the SMC instruction, every implementation for SeCReT_M is limited by the SMC exception handler. SeCReT_T not only invokes the SMC instruction, it administers parameters through the general-purpose registers — parameters such as the current event type. According to the *Procedure Call Standard for the ARM Architecture (AAPCA)*, four general-purpose registers from r0 to r3 can be used to input arguments [13]. Of the four registers, we utilized two registers, r0 and r1, to pass parameters to SeCReT_M.

The r0 register delivers the current event types such as process execution, key-assignment requests, and process termination. Based on the event type in r0, the subroutine to handle each event is invoked in the SMC exception handler. The r1 register has the virtual address for the kernel-stack pointer in the REE to validate the control flow integrity during switches to the mode. With a switch to kernel mode, SeCReT_M retrieves the LR_U and the return address from the kernel stack and pushes them into the shadow stack maintained in the APC. Moreover, with the execution of the process, the r1 register can offer the address for the process descriptor — that is, *task_struct* in Linux — to refer to the name of currently loaded executable. The value of other registers, such as the TTBR, DFSR, and DFAR can be queried by invoking a privileged instruction in monitor mode. Thus, we did not pass them as parameters of the SMC instruction. It should be noted that, even in the monitor mode, as long as the NS-bit for the SCR is set, we are able to retrieve the value for the registers based on the REE's context.

In ARM Linux, two page tables are maintained: the hardware page table, which supports the ARM-specific page descriptor; and the Linux page table, which is specific to the Linux OS. In ARM's page-table descriptor, some properties, such as the present bit and the dirty bit, are not supported. Thus, Linux maintains the additional page tables in order to emulate the missing properties. Figure 5 shows the difference of the attributes provided by each page table descriptor. SeCReT utilizes both tables. The hardware page table is used to set the NA permission bit that causes the data-abort exception when a process accesses the session key. The Linux page is traversed to discover the currently loaded code pages for the process by checking the present bit. The integrity of the code pages is checked before whitelisting the specific return address of the signal handler and before assigning the session key to the process. Both pages tables are protected by active monitoring and updated only in the TEE.

VI. EVALUATION

In this section, we present a security analysis and a performance evaluation for SeCReT. We evaluated the performance of SeCReT on Arndale board that offers a Cortex-A15 at 1.7 GHz dual-core processor. For the security analysis, not only Arndale board, but ARM Fast Models [4] that emulates ARM Cortex-A15 single-core processor was also utilized.

A. Security Analysis

There are several attack surfaces that an attacker can attempt to exploit for session-key exfiltration. First, the attacker can snapshot the physical memory area that contains the session key. To protect the key from the memory-snapshot attack, the memory area that contains the key should be flushed with the occurrence of event that causes the switch to kernel mode. We utilized a Model Debugger [8] to find a proper point of kernel code that SeCReT_T can be inserted. Because Fast Models are available for processors with various number of cores, we performed our analysis based on a Cortex-A15 single-core processor which makes debugging easier than a multi-core processor. After finishing the analysis, we inserted SeCReT_T to the REE kernels on Fast Models as well as Arndale board.

On each environment, running a single- and dual-core processor, respectively, we ran a rootkit that continuously copies the specific memory region reserved for the session-key assignment. At the same time, we ran a legitimate process

```
Input: An ascii payload of size: 128 to 8192 bytes
Output: Encrypted payload
*key = allocMemory()
if Key_Protection then
   assignKeyBySeCReT(key)
else
   *key=tempValue()
end if
payload = encrypt(payload, *key)
printString(payload)
```

Fig. 6. Measurement for key-access control overhead

that repeatedly accesses the session key. With this experiment on Fast Models with a single-core processor, we could confirm that our prototype for SeCReT flushes the session key clearly with every possible exception that causes a switch to kernel mode. However, on Arndale board with a dual-core processor, the key was exposed to the rootkit when the legitimate process and the rootkit are scheduled to run on different cores at the same time. The mitigation of this problem is quite simple when active monitoring cooperates with SeCReT. We discuss this in more detail in Section VII.

An attacker could also attempt to manipulate the code or the control flow of the process. As explained in Section IV-D1 and IV-D2, verifying the hash check and shadow stack can detect attacks to the code and the control flow, respectively. SeCReT_T, which is trampoline code, can be reused to add the malicious address to the shadow stack or whitelist_sighand. Moreover, SeCReT_T, inserted to the switch_to_user code, can be bypassed to skip the shadow-stack verification and jump to the address sought by the attacker. Such attacks are not feasible because we maintain the mode_switch_flag, which can be updated only once for each legitimate mode switch. SeCReT's defense mechanisms against each attack surface are summarized in Table II.

B. Performance

1) Microbenchmarks: We ran LMBench to evaluate the performance degradation to the overall system from enabling SeCReT. LMBench measures the performance of OS functionalities such as system-call invocations, memory mapping, and page-fault handling. For the evaluation, we created a temporary process that requests a session-key assignment to enable SeCReT. Once SeCReT is enabled, monitor mode is entered with every switch in modes, regardless of what processes are currently scheduled and running. Thus, any process that runs when SeCReT is enabled experiences some performance degradation.

Table III presents the results of the microbenchmark, reporting the average latency after running ten times for each system call and the performance degradation from enabling SeCReT, comparing it to running Linux without SeCReT. It should be noted that we evaluated the performance only on Arndale board because Fast Models does not guarantee the accuracy of cycle counts. The null system call caused the most degradation, indicating the performance overhead for switching between user and kernel mode. The read and write system call also caused high overhead. However, as can be seen with other system calls such as open and fork, the table reveals a tendency for less overhead imposed by SeCReT as the latency increases. When SeCReT is disabled, running the benchmark is meaningless because SeCReT is not triggered during the mode switch. Therefore, the overhead for SeCReT-disabled was not measured in our performance evaluation.

2) Key Access-Control Overhead: The overhead imposed by the access control to the session key was also measured. As can be seen in Figure 6, we wrote a simple test program that parses, encrypts, and prints an input payload. The input payload was a simple string with lengths varying from 128 to 8192 byte. We ran the test in three different environments: Linux, SeCReT-enabled Linux, and SeCReT-enabled Linux with key protection. SeCReT-enabled Linux with key protection refers to the environment where the test process uses the session key that is protected by SeCReT. For the other two environments, we temporarily assigned an integer value as a key to encrypt the payload. The key in both Linux and SeCReT-enabled Linux were unprotected. This was adopted merely to determine the performance overhead from SeCReT's key-protection mechanism.

The results of the experiment can be seen in Table IV, indicating the average latency after running ten times for each input payload. SeCReT Enabled shows that the performance overhead imposed from enabling SeCReT is maximum 16.41% with an input payload of 256 bytes. It is worth reiterating that the control flow is redirected to monitor mode with every switch in modes when SeCReT is enabled, whether the session key is protected or not. This additional mode switch is the main factor that causes the overhead in SeCReT Enabled.

Finally, we measured the overhead of SeCReT with Key Protection. As shown in Table IV, SeCReT with Key Protection indicates the overhead for protecting the session key in addition to enabling SeCReT. Unfortunately, the worst case in our experiment was the test run with an input payload of 128 bytes, outputting at 48.28% overhead, compared to Linux. According to our analysis, the performance was deteriorated since the key protection additionally required the hash-check for code area, and repetitive assignments and flushes of the session key. However, similar to the results of LMBench, the overhead was considerably reduced (almost down to 0.18% for key protection) as the latency of the test increased.

VII. Discussion

A. Extension of SeCReT

The current design of TrustZone's architecture does not provide an interface for user processes to communicate directly with TrustZone. The only way to synchronously enter TrustZone is invoking an SMC instruction, and such an instruction is only available in kernel mode. On the other hand, the hypercall of the hypervisor can be invoked directly from user mode. Thus, previous works [16], [20] can utilize the hypercall to protect the user process from the untrusted kernel. To build a direct communication protocol such as the hypercall in TrustZone's architecture, SeCReT's key-protection mechanism can be applied. For example, the user process can set up parameters to the predefined general-purpose registers and cause an exception intentionally by accessing the pinned page

TABLE IV. BENCHMARK OF SECRET OVERHEAD COMPARED TO LINUX

Payload Size (Bytes)	Linux Time (μs)	SeCReT Enabled Time (μs)	SeCReT Enabled Overhead	SeCReT w/ Key Protection Time (μs)	SeCReT w/ Key Protection Overhead
128	1334.6	1544.5	15.73%	1979.0	48.28%
256	1642.5	1912.1	16.41%	2425.8	47.69%
512	2279.4	2509.8	10.11%	3068.2	34.61%
1024	3650.9	3822.6	4.70%	4516.7	23.71%
2048	340225.7	340244.6	0.01%	341531.4	0.38%
4096	679761.2	679818.7	0.01%	681604.3	0.27%
8192	1693561.2	1693683.6	0.01%	1696639.1	0.18%

that has its access permission set to NA. Subsequently, the trampoline code inserted to the starting point of the exception handlers redirects the control flow to TrustZone. Even though the approach nevertheless requires the trampoline code from the kernel code, it is reasonable because the protection of the kernel code's integrity is available already in TrustZone's architecture [14].

Therefore, as shown in the previous works [16], [20], we can also leverage this approach to protect applications from untrusted kernels and facilitate the generalization of SeCReT.

SeCReT can also be extended to protect the guest VMs from vulnerable hypervisors [21], [38]. By inserting SeCReT_T into the exception vectors in HYP mode (hypervisor mode) and protecting the vectors from the malicious modification, SeCReT can work with virtualization extensions, as well.

B. Attack against SeCReT

In Section VI-A, the memory snapshot attack that was performed on a multi-core processor could exfiltrate the key even with SeCReT. Although the key is only transiently accessible, the rootkit still has a chance to snapshot the key if the running of rootkit is synchronized to the instance of access to the key by the legitimate process in another core.

However, such an attack is not feasible because of the following reasons. First, the attacker cannot directly manipulate the page table to map the physical address that contains the key because the page-table updates are available only on TrustZone running active monitoring [14], [19]. Second, active monitoring can refer to the APC and exclude the address that contains the key during the memory allocation and page-table update. A *Direct Memory Access (DMA)* is already controlled by the active monitoring to enforce the kernel code's integrity. Thus, an attack utilizing the DMA from another core is also prevented.

SeCReT is triggered by exceptions that occur asynchronously as well as synchronously. With synchronous triggers, the attacker may be able to predict when SeCReT will be engaged and create a transient attack that can hide itself before SeCReT is activated. However, the fact that SeCReT can also be triggered asynchronously would prevent the attackers from consistently predicting the exact instant that SeCReT runs. Thus, in SeCReT, any attempt that transiently manipulates the legitimate code in user mode or maps the malicious code in the data region of a pre-authorized application will be difficult, although not impossible, to conduct. SeCReT can further limit the occurrence of this transient attack by increasing the frequency of checking the code hash and/or by validating the return address to user mode to check if it falls within the range of a legitimate code area.

Attackers may attempt to analyze all possible paths of the control flow to pinpoint a specific exception that occurs consistently during the runtime of the process. Based on the analysis, an attacker might succeed at inferring the offsets of the saved return address from the current SP or manipulating the message directly. To mitigate this attack, we can write-protect the data area, such as user stack, and allow SeCReT to emulate writing to it. To this end, we must scrutinize the behavior of all system calls to learn how they update the user-level memory. However, we expect that the emulation of writing will cause additional performance degradation. Thus, without the emulation, SeCReT can selectively protect some critical data, such as the saved return address and message buffers. The list of critical data can be defined by analyzing the binary in advance. It can then be deployed in TrustZone and referred to as part of APC.

The registers that configure debug events can also be exploited to hijack the control flow arbitrarily. For instance, an attacker can set a break point to the instruction that is executed right before signing the message with the session key. This message can be replaced with a maliciously crafted one making the session key useless. As a simple mitigation for this attack, we can disable every break point by configuring the control register, such as the Breakpoint Control Registers (DBGBCR) in ARM, at every mode switch to user for the process that accesses the resources in the TEE.

Applications in the REE that use the session key also require careful development to protect against vulnerabilities — both control and non-control data vulnerability — that an attacker can exploit to exfiltrate the key. Although this would appear to be a task for application developers, we plan to explore an efficient way for TrustZone to aid in obfuscating the application in order to prevent attackers from analyzing the application in advance or during run-time.

C. Usability of SeCReT

In this section, we consider two aspects of SeCReT's usability: (1) protecting the session key, and (2) updating the list of pre-authorized applications.

The session key should be used in a secure manner because simple operations can create copies of the key. We are designing SeCReT libraries that help developers safely use the session key. For instance, a SeCReT library should not copy the key values to unprotected memory during message signing. The library can achieve this by strictly using the general-purpose registers that SeCReT can flush out on every mode switch.

We note that an alternate design could avoid using keys and transfer data directly to the TEE using a secure buffer in the memory area that is protected by SeCReT. Unfortunately, this approach is limited by the the following implementation challenges. First, the secure buffer approach requires more protected memory pages, depending on the size of the messages that need to be transferred to the TEE. Second, because the pages that SeCReT should protect can be dynamic during the runtime of the application (e.g., the location, size, and value of the data can change), more interactions between the REE and the TEE may be required to update the information appropriately.

Finally, to provide the TEE services to newly developed applications, we must securely update the list of pre-authorized applications. To this end, we can leverage the mechanism that TrustZone-based DRM solutions utilize. In other words, we can send an encrypted list outside of the device and safely decrypt it inside TrustZone. Alternatively, we can also update the list as part of a firmware-upgrade procedure that is currently available for updating the TEE resource on commercial devices.

VIII. Related Work

A. Active Monitoring

Active monitoring utilizes either trampoline code or hooks implanted in the monitored system to redirect the control flow to the monitoring code. Thus, active monitoring relies on the protection of the trampolines and the monitoring code to guarantee the event-driven nature and consistency of the monitoring. The hypervisor was widely adopted to satisfy this requirement because, in essence, it provides an isolation between guest VMs. Furthermore, the hypervisor can control and inspect the guest VMs because it is accorded a higher privilege than the guest VMs. *Lares* [28] implants the trampoline code in a monitored VM and uses the hypervisor-layer memory protection to prevent attackers from compromising the trampolines. *SIM* [33] is similar to *Lares* in that it uses trampolines to branch to the monitoring code. However, *SIM* is focused more on efficiency. To this end, it locates the monitoring code in an untrusted guest VM and removes the involvement of the hypervisor during the transition between the trampolines and the monitoring code. *SecVisor* [32] is a tiny hypervisor that protects the kernel code by leveraging hardware-memory protection. To protect the module code, it hooks the kernel code to invoke *SecVisor* with a hypercall. *HookSafe* [37] relocates thousands of kernel hooks into continuous memory and protects them with the page granularity. Even though *HookSafe* aims to solve the problem related to the protection-granularity gap, its effect is somewhat redundant because it redirects or verifies accesses to the hooks that are done similarly with active monitoring.

On the other hand, TrustZone is also utilized to implement active monitoring. *TIMA* [14], [27], [30] and *SPROBES* [19] locate the monitoring code in TrustZone and implants the trampoline in the kernel code in the REE to enforce the integrity of the kernel's static region. SeCReT can also be regarded as a variety of active monitoring in TrustZone because it implements trampolines and isolated access-control modules. However, the purpose of SeCReT is to secure a communication channel, and this is clearly different from other works. Moreover, SeCReT was implemented with the assumption that the kernel's static region is immutable due to the protection provided by active monitoring.

B. Process-Data Protection

Several previous works aim to protect user processes from the untrusted kernel. *XOM* [22] and *XOMOS* [23] are hardware-based approaches — specially designed hardware and a dedicated OS for *XOM*, respectively — enabling the protection of secure processes by using cryptographic technology. All code and data remains encrypted outside *XOM*, but securely decrypted and executed during run-time in *XOM*.

A hypervisor and compiler have also been used as a software-based approach. *Overshadow* [16] and *InkTag* [20], both of which are implemented on hypervisors, encrypt and decrypt the memory of user processes based on the context switch. *Overshadow* introduces a *Shim* that communicates with the hypervisor to interpose every mode switch between a cloaked application and the OS. *InkTag* uses a trampoline code similar to the Shim, and it also provide *para-verification* so that the untrusted OS verifies its own behavior to some degree. *Virtual Ghost* [17] uses compiler-based instrumentation to prevent untrusted kernels from writing to or reading from the protected memory area of the user process.

SeCReT, like previous work, attempts to protect the critical component of the process, whether it is data or code. Securing a communication channel also requires the interposition of SeCReT at every mode switch. However, because it utilizes an existing hardware component — namely, TrustZone — SeCReT does not require additional hardware or software stacks. Furthermore, TrustZone basically ensures the confidentiality of the contents inside it, so we only focused on a way to reinforce the access control to the resources in TrustZone. In the absence of a direct channel, such as hypercall, SeCReT creates the protocol that the user process can use to communicate with TrustZone as directly as possible.

C. Trusted-Execution Environment

Isolating individual guest VMs, ensured by a hypervisor, enables the TEE to be built on the hypervisor. Previous systems [18], [28], [34], [35] that implement an external monitor on separated VMs are examples of a TEE built on the hypervisor. Intel's SMM is an operating mode with special software, such as firmware or a debugger, running with all normal execution suspended. This is also regarded as a TEE, because any process that runs in SMM is isolated from an attacker outside SMM. *HyperCheck* [36] and *HyperSentry* [15] use SMM as a TEE to securely run the monitoring code to check the integrity of the hypervisor. Intel's *Software Guard Extensions (SGX)* is similar to TrustZone in that it provides separated memory regions from the REE [26]. The isolated area, called an *enclave*, is protected against all external software access, guaranteeing the security of critical resources inside the enclave.

In addition to *TIMA*, *Trusted Sensor* [25] and *Trusted Language Runtime (TLR)* [31] also leverage TrustZone as a TEE. *Trusted Sensor* attempts to ensure that mobile applications read sensors securely in TrustZone. *TLR* enables separating the critical part of a .NET mobile application and runs it in TrustZone. These systems were implemented without

considering a secure communication channel. We believe that SeCReT can help to reinforce the security of those systems.

IX. Conclusion

SeCReT is a framework that strengthens the security of the communication channel between two domains, the REE and the TEE built in TrustZone. To establish a secure channel, SeCReT enables a legitimate process to use a session key in the REE. The key is only readable at the moment the legitimate process accesses the memory that is reserved for the key assignment. To protect the key, SeCReT interposes with every switch between user mode and kernel mode, verifying the code's integrity and the coarse-grained control flow of the process. To minimize the performance overhead, SeCReT's key-protection mechanism is activated only during the runtime of the process that has permission to access TrustZone. As the first work to secure a communication channel in TrustZone, we believe that SeCReT will not only regulate malicious access to the critical resources, but also cooperate with existing TrustZone-based security solutions such as *TIMA*.

Acknowledgment

We would like to thank our shepherd William Enck and the anonymous reviewers for insightful comments and suggestions. This research was supported by MOTIE (The Minister of Trade, Industry and Energy), Korea, under the BrainScouting-Program (HB609-12-3002) by the NIPA (National IT Promotion Agency). This work was also sponsored by Agency for Defense Development (ADD) under Grant No. UD140002ED and the research project from SAMSUNG ELECTRONICS.

References

[1] "Architecture reference manual (armv7-a and armv7-r edition)," *ARM DDI C*, vol. 406, 2008.

[2] "cve-2013-3051," April 2013. [Online]. Available: http://www.cvedetails.com/cve/CVE-2013-3051/

[3] "Unlocking the motorola bootloader," April 2013. [Online]. Available: http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html

[4] "Arm fastmodels," June 2014. [Online]. Available: http://www.arm.com/products/tools/models/fast-models/

[5] "Arm security technology: Building a secure system using trustzone technology," Tech. Rep., June 2014. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[6] "Arndale board," June 2014. [Online]. Available: http://www.arndaleboard.org/wiki/index.php/Main_Page

[7] "Discretix," June 2014. [Online]. Available: http://www.discretix.com/products-solutions

[8] "Model debugger for fast models," Tech. Rep., June 2014. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0314i/DUI0314I_model_debugger_ug.pdf

[9] "Proxama," June 2014. [Online]. Available: http://www.proxama.com/products-and-services/trustzone

[10] "Sensepost," June 2014. [Online]. Available: http://www.sensepost.com/blog/9114.html

[11] "Sierraware," June 2014. [Online]. Available: http://www.openvirtualization.org/

[12] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997, pp. 65–71.

[13] ARM, "Procedure call standard for the arm architecture," Tech. Rep., November 2012. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf

[14] A. AZAB and P. Ning, "Methods, systems, and computer readable medium for active monitoring, memory protection and integrity verification of target devices," Patent WO 2014/021 919, June 2, 2014. [Online]. Available: http://patentscope.wipo.int/search/en/WO2014021919

[15] 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 *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 38–49.

[16] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2. ACM, 2008, pp. 2–13.

[17] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: protecting applications from hostile operating systems," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 81–96.

[18] Y. Fu and Z. Lin, "Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 586–600.

[19] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," 2014.

[20] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: secure applications on an untrusted operating system," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 265–278, 2013.

[21] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 272–283.

[22] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[23] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 178–192.

[24] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *USENIX Security Symposium*, 2008, pp. 243–258.

[25] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 365–378.

[26] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," *HASP*, vol. 13, p. 10, 2013.

[27] P. Ning, "Introducing the samsung knox plaform," Tech. Rep., June 2014. [Online]. Available: http://samsungdevus.com/sites/Default/files/IntroducingtheSamsungKNOXPlatform-PengNing.pdf

[28] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 233–247.

[29] D. R. Ports and T. Garfinkel, "Towards application security on untrusted operating systems," in *HotSec*, 2008.

[30] L. Samsung Electronics Co., "White paper: An overview of samsung knox," Tech. Rep., June 2014. [Online]. Available: http://www.samsung.com/my/business-images/resource/white-paper/2013/11/Samsung_KNOX_whitepaper_An_Overview_of_Samsung_KNOX-0.pdf

[31] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. ACM, 2014, pp. 67–80.

[32] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: A tiny hypervisor

to provide lifetime kernel code integrity for commodity oses," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

[33] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 477–487.

[34] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 363–374.

[35] A. Srivastava and J. T. Giffin, "Efficient monitoring of untrusted kernel-mode execution." in *NDSS*, 2011.

[36] J. Wang, A. Stavrou, and A. Ghosh, "Hypercheck: A hardware-assisted integrity monitor," in *Recent Advances in Intrusion Detection*. Springer, 2010, pp. 158–177.

[37] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 545–554.

[38] F. Zhang, J. Chen, H. Chen, and B. Zang, "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 203–216.