TrustICT: An Efficient Trusted Interaction Interface between Isolated Execution Domains on ARM Multi-core Processors

Jie Wang[1,2,3], Yuewu Wang[1,3], Lingguang Lei[1,3*], Kun Sun[2], Jiwu Jing[4], and Quan Zhou[1]
[1]SKLOIS, Institute of Information Engineering, CAS, China
[2]Department of Information Sciences and Technology, CSIS, George Mason University
[3]School of Cyber Security, University of Chinese Academy of Sciences, China
[4]School of Computer Science and Technology, University of Chinese Academy of Sciences, China
{wangjie, wangyuewu, leilingguang, zhouquan}@iie.ac.cn, ksun3@gmu.edu, jwjing@ucas.ac.cn}

ABSTRACT

The Trusted Execution Environment (TEE) has been widely used to protect the security-sensitive sensing systems on Internet-of-Thing (IoT) devices. In the TEE systems, the execution environment is securely divided into a normal domain and a higher privileged secure domain which executing sensing systems through hardware. One common way to achieve the protection is implementing the sensitive functions of the sensing systems as trusted applications (TAs) in the well-isolated secure domain. Users in rich OS have to call TAs through the client applications (CAs), and the invocations must pass through the rich OS kernel. However, an untrusted rich OS may launch man-in-the-middle attacks on the communication between the CAs and TAs, and the misuse of cross-domain communication channel is becoming one severe threat on the TEE systems. In this paper, we develop a defense system named TrustICT to construct a lightweight trusted interaction channel between CAs and TAs without modifying existing TEE architecture. The main idea is to block attacks on the cross-domain interactions via dynamically setting the access permission of domain-shared memory, locking it from kernel mode and unlocking it only to legal CAs in the user mode. Particularly, we propose a multi-core scheduling strategy to defeat potential attacks from all privileged cores. Compared to existing cryptography-based methods, TrustICT dramatically reduces the system overhead since it does not require time-consuming cryptographic computation or sophisticated real-time kernel protection. We implement a prototype of TrustICT on a Freescale i.MX6Quad platform with the OP-TEE software system and evaluate its impacts on rich OS and the cross-domain transactions.

CCS CONCEPTS

• **Security and privacy → Systems security**; • **Computer systems organization → Architectures**.

*Lingguang Lei is the corresponding author.

KEYWORDS

TrustZone, Trusted Interaction Interface, Multi-core Processor

ACM Reference Format:
Jie Wang[1,2,3], Yuewu Wang[1,3], Lingguang Lei[1,3*], Kun Sun[2], Jiwu Jing[4], and Quan Zhou[1]. 2020. TrustICT: An Efficient Trusted Interaction Interface between Isolated Execution Domains on ARM Multi-core Processors. In *The 18th ACM Conference on Embedded Networked Sensor Systems (SenSys '20), November 16–19, 2020, Virtual Event, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3384419.3430718

1 INTRODUCTION

ARM TrustZone is the enclave technology originally developed by ARM company. It was introduced into Cortex-A processors in 2004 [49], which are widely used for mobile devices in the embedded market. ARM also launches Cortex-M series processors (i.e. the Cortex-M23 [8] and the Cortex-M33 [9]) equipped with TrustZone technology to support low-power IoT devices. For example, low-power micro-controllers such as i.MX-RT500 [45], i.MX-RT600 [46], nRF5340 [44], STM32L5 [55] and etc., have deployed these Cortex-M series processors to improve the security of the IoT devices.

The ARM TrustZone technology utilizes hardware-based isolation to construct Trusted Execution Environment (TEE) systems to ensure the security of sensitive assets. It divides the execution environment into a normal privileged non-secure domain (i.e., normal domain) and a higher privileged secure domain. The untrusted OS (also known as rich OS) runs in the normal domain along with normal applications. Meanwhile, the TEE OS runs in the secure domain, along with a limited number of trusted applications (TAs) to provide dedicated security services. The untrusted OS in the normal domain cannot access the resources of the secure domain.

Due to the increasing numbers of vulnerabilities in the untrusted OS and the growing attention to the security of the sensing system, Internet-of-Thing (IoT) device developers begin to protect the security-sensitive sensing systems with the TrustZone-based TEE systems, i.e., deploying the sensitive assets of the sensing systems in the secure domain. For instance, Brasser et al. regulate the usage of sensors and peripherals in restricted spaces [16]. *SeCloak* ensures reliable control of peripherals and sensors even when the platform software is compromised [36]. Liu et al. implement a trusted GPS sensor to support trusted reading operation [38]. *TLR* protects Microsoft's .NET framework which is optimized for sensor networks and wearable devices for mobile devices [52]. *VirtSense* implements an ARM TrustZone based virtual sensing system [39].

In the TEE systems, the sensitive assets in the secure domain are accessible to the normal domain only via invoking the TAs. Each TA usually has a counterpart called client application (CA) running in the normal domain, and the CA is responsible for interacting with the TA. Although the execution of TAs is well protected in the secure domain, the existing TEE systems lack a sound mechanism to secure the access of TAs. Since the transaction between CAs and TAs must go through the untrusted rich OS, the attackers may achieve illegal access to the TAs by compromising the cross-domain communication channel. For instance, a set of vulnerabilities on the cross-domain communication channel named BOOMERANG [40] can be exploited by attackers without kernel privilege to misuse the TAs, e.g., deceiving the TAs to process deliberately crafted data in the fake memory addresses. Moreover, attackers can continuously invoke TAs with crafted arguments to discover and exploit the vulnerabilities in the TEE systems [18, 22, 34, 35, 51]. Therefore, it is critical to prevent the cross-domain interaction interface for TEE systems from being misused by attackers.

Several efforts have been made to protect the cross-domain interactions. Android *Sandbox* [4] and *SEAndroid* [54] mechanisms have been used to ensure only applications with specific privileges can interact with TAs [25]. Machiry et al. [40] resolve the BOOMERANG vulnerabilities by checking the correctness of the addresses with the current process's memory information provided by rich OS. However, all these solutions rely on the security of rich OS, which might also be compromised in real world. *SeCReT* [33] aims to construct a secure data channel between CAs and TAs without trusting rich OS. It encrypts the transaction data and restricts the encryption keys to be only accessible to legal CAs in the user space (but not the privileged rich OS kernel) by interposing each mode switch operation between the user space and the kernel space of rich OS. However, the time-consuming encryption and decryption operations in *SeCReT* noticeably slow down the entire system. Moreover, to protect the inserted trampoline code, *SeCReT* relies on heavy real-time kernel protection solutions (e.g., TZ-RKP [10]) that need to hook and monitor each page table operation, further increasing its computation overhead. In addition, *SeCReT* is a solution designed on single-core platforms, while modern mobile devices are popularly equipped with the multi-core processors. It is difficult to directly apply *SeCReT* on multi-core platforms, since the performance will sharply degrade due to the frequent kernel-user mode switching and the encryption and decryption operations.

In this paper, we develop a lightweight trusted interaction mechanism named TrustICT between CAs and TAs on multi-core platforms. TrustICT achieves the trusted cross-domain communication channel by protecting the access to the ***Domain-shared Memory (DsM)***, i.e., memory designated for cross-domain communication [47], through the ARM TrustZone memory isolation mechanism named *TrustZone Address Space Controller (TZASC)*. Specifically, TrustICT hooks the mode switch operations in rich OS and traps into the secure domain. The manager process executed in the secure domain dynamically controls the access permission of DsM through TZASC and it can ensure (i) the protected DsM memory can only be read and written by legal CAs, but not rich OS or illegal user applications, and (ii) the TAs only process data in the protected DsM memory and return data to the protected DsM memory. In other words, it unlocks the DsM region by setting its access permission as *normal domain accessible* in the kernel-to-user mode switch hooks (hereinafter referred to as `K-U hooks`) to allow legal CAs' access when rich OS switches to the user mode, and it locks the DsM region by setting its access permission as *normal domain inaccessible* in the user-to-kernel mode switch hooks (hereinafter referred to as `U-K hooks`) to prevent normal domain's access once the rich OS switches into the kernel mode. Compared to the cryptographic operations, it is much faster to control the access permission of DsM by setting a couple of registers.

When enabling TrustICT on multi-core platforms, we make four major efforts. First, we introduce a lightweight hook-protecting solution instead of relying on the real-time kernel protection scheme. It leverages TZASC to prevent the hooking codes from being tampered by the untrusted privileged kernel (i.e., setting the associated memory as non-writable to normal domain). Also, it prevents the hooks from being bypassed via intermittent value checking, rather than interposing each page table operation. This is based on two observations: (i) it only needs to prevent the bypassing of U-K hooks [1], and only when certain DsM is unlocked; and (ii) the U-K mode switch is accomplished by hardware components (i.e., CPU and MMU) with the help of a few exception-associated registers and mappings, which are non-writable in the user space. The U-K hooks will definitely be executed if the exception-associated values are correctly configured. As such, we check the correctness of exception-associated values before unlocking certain DsM in the K-U hooks.

Second, TrustICT deters the direct damages on the protected DsM from the privileged rich OS by unlocking the DsM only when none of the cores are running in the kernel mode of the normal domain. However, this scheme might affect the DsM accessing operations in the legal CAs, since they are running in the user mode of normal domain and are unaware of the DsM's locking state change (which is controlled in the secure domain). We introduce a polling mechanism to resolve this problem, i.e., repeating the read and write operations of CAs until the corresponding DsM regions are correctly read and written. TrustICT also includes an optimization solution to prevent the DsM accessing operations from being stuck by the kernel-stuck cores (e.g., the cores retaining in the kernel-mode *idle* state).

Third, TrustICT unlocks the protected DsM only to legal CAs and prevents the indirect disruption launched through manipulating the execution of legal CAs. Particularly, it introduces three protection measures: (i) Once a CA process is started, the code integrity check is conducted. If the check passes, the CA's execution codes are set as non-writable to the rich OS through TZASC. (ii) The CA's executing context that is stored in the kernel space stack is hidden in the U-K hooks and restored in the K-U hooks. Also, the CA's critical user space data memory (e.g., stack, data section, etc.) is locked to rich OS kernel. (iii) A double map check is performed on the protected memory (including a legal CA's DsM region and critical user space data) before they are unlocked.

Forth, we address the confused deputy attacks that deceive the TAs to process data in fake addresses rather than DsM written by the CAs. Due to the semantic gap between secure domain and

[1]Bypassing of the K-U hooks may only cause usability issue such as hindering the proper unlocking of DsM, rather than sensitive data leakage .

normal domain, it is difficult for TAs in secure domain to verify the correctness of the memory addresses [40]. To resolve this problem, TrustICT ensures each DsM region is only allocated for one CA and maintains a one-to-one mapping between the CA and its corresponding DsM region. Furthermore, it performs two security checks on the address passed to a TA, namely, if the address corresponds to a specific DsM region maintained in the one-to-one mapping table and if the TA invocation is initiated by the corresponding CA.

We implement a system prototype of TrustICT on the i.MX6Quad platform, with an OP-TEE OS 2.2.0 [48] [2] ported to the secure domain and an Android OS 6.0.1 (Linux kernel 4.1.15) ported as the rich OS in the normal domain. We also evaluate the performance of TrustICT based on the prototype. The experimental results show that TrustICT is efficient, and incurs about 2% overhead on rich OS.

In summary, we make the following contributions.

- We design a lightweight trusted interaction channel named TrustICT between CAs and TAs when the rich OS cannot be trusted. Instead of relying on heavy encryption/decryption operations and real-time kernel protection, TrustICT achieves a lightweight secure communication channel by enforcing the secure access of DsM via the TZASC that is available on most mainstream ARM-based platforms.
- We present a systematic study on designing and securing the cross-domain communication on multi-core platforms, including potential attacks and their countermeasures.
- We implement a system prototype on i.MX6Quad platform, running the OP-TEE OS system that is compliant with the GlobalPlatform TEE specifications [27] and compatible with many hardware platforms. Experiment results show that TrustICT only incurs a small performance overhead on rich OS.

2 BACKGROUND

This section provides necessary backgrounds of TrustICT, including architecture of the TEE system, the Domain-shared Memory (DsM) mechanism, and the TZASC of ARM TrustZone.

2.1 Architecture of the TEE System

According to the specification published by GlobalPlatform [27], a TEE system [28] generally contains two domains (i.e., secure domain and normal domain) isolated through hardware-based mechanism such as TrustZone [2], as shown in Figure 1. A traditional untrusted OS (often called as rich OS) runs in the normal domain, along with various feature-rich applications running on it. A trusted TEE OS and several Trusted Applications (TAs) run in the secure domain. Each TA usually has a corresponding client application (CA) running in the normal domain, and the CA is responsible for interacting with the TA. The hardware mechanism (e.g., TrustZone) protects the secure domain against the software attacks from the normal domain, even when the attackers obtain the kernel privilege to access all resources in the normal domain(e.g., through exploiting the privilege escalation vulnerabilities [1, 59]).

The applications in the normal domain invoke the TAs to accomplish dedicated security functions (e.g., digital rights management,

[2]OP-TEE OS 2.2.0 is an open source TEE system compatible with many hardware platforms.

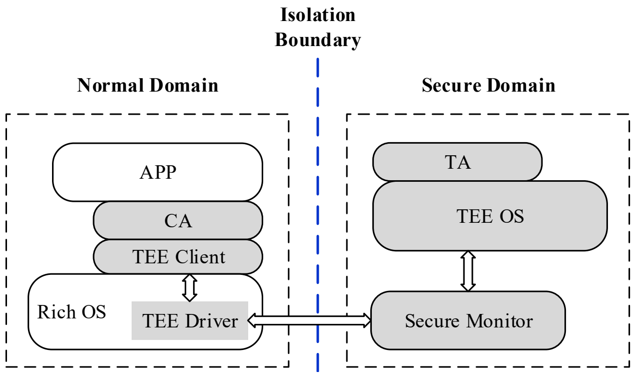

Figure 1: The Architecture of TEE System

authentication, etc.). Since the execution of the TA is isolated in secure domain, two components named TEE client and TEE driver are introduced in rich OS to facilitate the invocation. TEE client is a dynamic library, which hides the transaction details and provides user-friendly APIs for the CAs. TEE driver is the one that directly interacts with the secure domain. When a CA initiates a TA invocation through certain TEE client API, the TEE driver executes a domain switch instruction (e.g., Secure Monitor Call (SMC) instruction in TrustZone) to freeze the rich OS and switch the processor into the secure monitor mode. The secure monitor then accomplishes the context switching from the normal domain to the secure domain. Next, the TEE OS and a corresponding TA are activated to respond to the invocation. When the TA finishes its execution, the secure monitor restores the system to the point where rich OS suspends and returns the results to the CA.

2.2 TZASC of ARM TrustZone

ARM TrustZone is a hardware security extension since ARMv6 architecture, and it has been widely deployed on the smart mobile devices [2]. It provides hardware-based security by partitioning the resources of the ARM System-on-a-Chip (SoC) including processor, memory, and peripherals. TrustZone Address Space Controller (TZASC) [7] is the mechanism used to achieve memory isolation, and it can only be configured in the secure domain. Some regions of memory could be protected against untrusted rich OS by setting TZASC. TZASC set the memory as secure to prevent access from normal world. TZASC contains a set of registers for each memory region, and it supports to protect 7 memory regions simultaneously. Each region represents a segment of physically contiguous memory, whose size should be no less than 32KB. The starting and ending address of the region, and the access permission on the region could both be flexibly configured via TZASC registers in secure domain. If several regions are overlapped, the access permission of the region with the largest identifier number will take effect. For example, if region 1 is set as only accessible to the secure domain and region 2 is set as accessible to both domains, then the overlapped memory will follow the access permission of region 2 (i.e., accessible to both domains). TZASC also supports to set a region as read-only or write-only to normal domain, rather than both readable and writable.

2.3 Domain-shared Memory

Domain-shared Memory (DsM) is a common cross-domain communication mechanism, whose implementation might vary slightly on different TEE systems. We focus on the DsM mechanism of the OP-TEE system, which is a typical TEE system maintained by Linaro [37]. OP-TEE is compliant with the architecture illustrated in Figure 1 and is compatible with many hardware platforms. The cross-domain communication in OP-TEE system is accomplished through DsM mechanism. In general, DsM is a physically contiguous memory pool and primarily managed by the TEE driver. When system boots up, the TEE driver initializes DsM based on the configuration information obtained from TEE OS, such as the starting physical address of DsM, size of DsM and if the memory should be cached, etc. During the run time, the TEE driver allocates or frees DsM according to the demands of the CAs. The DsM is managed in chunks of 4KB, and each chunk is assigned an ID when allocated. Chunk operations are encapsulated into a group of user-friendly TEE client APIs to facilitate the development of CAs. The CAs read and write a DsM chunk by specifying its corresponding ID. The TEE driver then converts the IDs into physical memory addresses, and informs the addresses to TEE OS. As such, the TAs could locate and obtain data inputted by the corresponding CAs.

3 THREAT MODEL AND ASSUMPTIONS

TrustICT aims to construct a lightweight trusted interaction channel between CAs and TAs on the multi-core platforms, defending against the potential man-in-the-middle attacks from untrusted rich OS kernel. We assume the platforms are equipped with TrustZone security extension [2], and the hardware implementations of TrustZone are correct and could be trusted. Components running in the secure domain (including TAs, TEE OS, and the secure monitor) are benign and can be securely booted up via the *secure boot* technology of TrustZone. We assume that a list of legal CAs permitted to access the TAs is predefined and their identification information (e.g., hash values of the execution codes) is safely maintained in the secure domain. We assume legal CAs running in normal domain do not either deliberately misuse the TAs by providing fake transaction data or leak the cross-domain communication data (e.g., the data inputted to TAs or returned by TAs) to the rich OS kernel.

In this paper, we focus on protecting the confidentiality and integrity of the cross-domain communication data, and the availability issue is out of our consideration. Specifically, we enforce the security of cross-domain communication against following four attacks from the normal world. First, the attackers can get the location of the cross-domain interaction channel and directly access data in it. Second, they might illegally access the sensitive communication data through obtaining the execution information of legal CAs (e.g., codes) and modifying the control flows. Third, they may construct a malicious application to disguise as a legal CA. Fourth, they could provide a fake communication address and deceive TAs to process the malicious data in it.

4 DESIGN AND IMPLEMENTATION

We present the design and implementation of TrustICT on OP-TEE system. We first give an overview of the system architecture along with the main communication procedure between CAs and TAs.

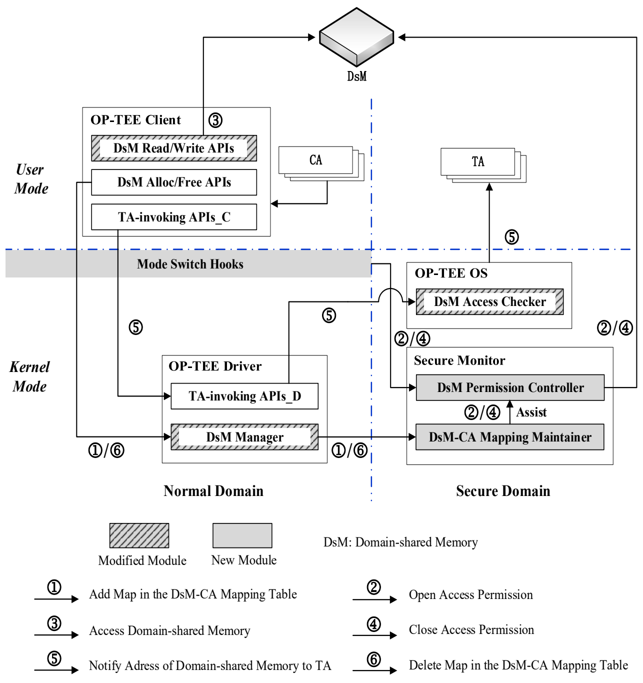

Figure 2: The Architecture of TrustICT

Then, we depict the dynamic configuration of the access permission on DsM regions during system mode switching. It is the core technique to achieve a lightweight trusted interaction channel. Next, we provide detailed solutions to resolve three problems, namely, how to securely instrument mode switch hooks, how to securely inform CAs about the locking states of DsM, and how to prevent the attacks from the privileged rich OS.

4.1 System Architecture

The architecture of TrustICT is illustrated in Figure 2. The gray boxes denote the modules newly introduced by TrustICT, and the gray boxes with slash represent the modules extended from existing functions of the OP-TEE system. Generally, we interpose the cross-world communication, ensuring the associated DsM memory being accessible only to legal CAs and TAs, and only data in the protected DsM memory being processed by the TAs. In the following, we introduce the main components of TrustICT along with a cross-world communication operation, which includes six steps, as shown in Figure 2.

① **Allocating a DsM region for a CA**. The cross-domain communication is initiated by a CA process through invoking the *DsM Alloc APIs* (provided by the *TEE Client*) to apply for one or more DsM regions. The *DsM Alloc APIs* will then invoke the *DsM Manager* (in the *TEE Driver*) to allocate the DsM regions. TrustICT modifies the DsM allocation procedure to add a *one to one* mapping, which called as DSM-CA mapping, between CAs and DsM regions. Specifically, TrustICT introduces a *DsM-CA Mapping Maintainer* module in the secure monitor to manage the DsM-CA mapping relationship. It also modifies the *DsM Manager* to launch a mapping constructing request after a DsM region is allocated, with the physical address of the DsM region as a parameter. After receiving the request, *DsM-CA Mapping Maintainer* inserts a DsM-CA mapping item if that request is initiated by a legal CA process.

② **Opening access permission of DsM region.** TrustICT locks all DsM region (by setting the access permission as inaccessible) when the system boots up. So the DsM region should first be unlocked before being accessed by a CA process. TrustICT introduces two modules to securely control the access permission of DsM regions, including *Mode Switch Hooks* in the kernel of rich OS and *DsM Permission Controller* in the secure monitor. The *Mode Switch Hooks* interpose all user-to-kernel (U-K) and kernel-to-user (K-U) mode switch operations in rich OS, and will inform the *DsM Permission Controller* module to lock or unlock certain DsM region. In step ②, after the *DsM Manager* finishes allocating the DsM region, it will return the control flow to the *DsM Alloc APIs*, which will trigger a K-U mode switch. Then, in the K-U hook, the *DsM Permission Controller* could be informed to unlock certain DsM region if the CA process is legal and the DsM region belongs to that CA process (i.e., matching one item in the DsM-CA mapping list).

③ **Reading and writing DsM regions.** When a DsM region is unlocked, the corresponding CA process can read and write it. However, the DsM accessing operations in the CA process might be disrupted due to the locking state change of DsM region, the DsM accessing operations will access incorrect data if the DsM is locked. Since the CA processes are running in the user mode of normal domain and are unknown of the DsM's locking state change in the secure domain. Therefore, TrustICT introduces a mechanism to securely inform CAs of the DsM states, which is detailed in Section 4.4.

④ **Closing access permission of DsM region.** After the DsM reading/writing operations are finished, the CA process will invoke the TEE driver to inform the address which contains data written by the CA to the corresponding TA. It will cause a U-K mode switch in the rich OS. Then, in the U-K hook, the *DsM Permission Controller* will be informed to lock all DsM regions by configuring them as inaccessible again.

⑤ **Notifying the address of DsM region to the TA being invoked.** When the TA is invoked, the TEE driver will pass to TA the physical address of the DsM region written by the CA. In the original cross-domain communication, the *DsM Access Checker* in the TEE OS will check whether the addresses are among a specific physical memory range. This loose verification cannot defend against the attacks launched through exploiting semantic gap vulnerabilities (e.g., BOOMERANG vulnerabilities [40]). For example, the TEE driver can substitute real address of the DsM region with a fake address that is also inside the legal memory range. TrustICT enhances the *DsM Access Checker* with fine-grained check, i.e., verifying if the address provided by TEE driver is maintained in the *DsM-CA Mapping Maintainer* and the TA invocation is initiated by the corresponding CA. If it passes the checking, the TA processes data in the DsM region and writes the results back to the DsM region. After the TA finishes writing DsM region, the context is switched back to the TEE driver. The TEE driver will perform a K-U mode switch and transfer the control flow to CA. In the K-U hook, the DsM region will be unlocked. Then, the CA can read the data returned by TA.

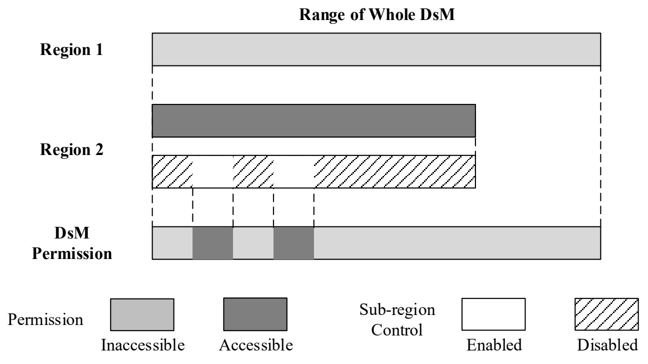

Figure 3: Fine-grained DsM Access Control over Sub-regions

⑥ **Freeing DsM region.** Finally, the *DsM Free APIs* in the TEE Client will be called to free the DsM region, which will trigger a U-K mode switch and cause the DsM region being locked. We modify the *DsM Manager* to launch a mapping removing request after a DsM region is freed, with the physical address of the DsM region as a parameter. Then the mapping relationship will be removed by the *DsM-CA Mapping Maintainer* if the CA process is legal and the DsM region belongs to that CA process matches on item in the DsM-CA mapping list. To prevent data leakage, TrustICT also clears the data in the freed DsM region.

In step ①, the attackers may provide a fake DsM region (i.e., a fake physical address) to the *DsM-CA Mapping Maintainer*, since the *TEE Driver* is untrusted. However, the attackers could not misuse the communication channel by pre-storing malicious data in the fake DsM region, since legacy data in the DsM region will be cleaned once it is allocated to a CA. Moreover, although the subsequent cross-world communication is accomplished through the fake DsM region, the attacker could not manipulate or steal the data stored in it since TrustICT will securely lock the fake region.

4.2 Dynamically Configuring DsM

The *DsM Permission Controller* module is responsible to lock and unlock specific DsM regions dynamically, based on the requests from the mode switch hooks. The locking operations are achieved by setting specific DsM regions as inaccessible to the normal domain through TZASC, and the unlocking operations are achieved by setting the DsM regions as accessible.

However, as mentioned in Sections 2.2 and 2.3, the DsM is allocated at the granularity of 4KB, while the minimum size of a TZASC region is 32KB. If we protect DsM regions directly through TZASC permission setting on the allocation of at least 32KB for a CA, it will waste a lot of memory. To solve this problem, TrustICT leverages the sub-region mechanism of TZASC to achieve a fine-grained control. Specifically, each TZASC region could be divided into 8 equal sub-regions. Though all sub-regions share the same access policy enforced on that region, TZASC sub-region mechanism allow turning off access policy for each sub-region. Thus, we can lock one sub-region by locking the region and disabling the access policy on other 7 sub-regions.

As shown in Figure 3, the *DsM Permission Controller* creates a TZASC region (i.e., region 1) when the system boots up, which covers the entire DsM memory and is set as inaccessible to the normal domain. When receiving DsM memory allocation requests, a new TZASC region (i.e., region 2) whose physical memory is among region 1 will be created, with the access permission set as accessible and access policy disabled on all sub-regions. DsM regions could then be unlocked (or locked) by enabling (or disabling) the access policy on specific sub-regions, since the memory's access permission is determined by the region with largest identifier number (as described in Section 2.2). Figure 3 shows an example that by enabling the access policy of two sub-regions in region 2, the associated physical memory is unlocked, while other DsM memory is still locked.

For each U-K mode switch, the U-K hook will lock the DsM memory. And when K-U mode switch happens, the K-U hook only unlock the DsM memory when such memory is accessed by the legal CA and none of the cores are running in the kernel mode.

4.3 Securely Instrumenting Mode Switch Hooks

Locking and unlocking DsM are triggered by the mode switch hooks instrumented in the rich OS. We must assure that these hooks cannot be bypassed or tampered by rich OS. However existing hook protecting solutions are usually achieved through heavy page-table-operation monitoring, not well suited for TrustICT. So, we introduces a more lightweight solution. We first describe the normal mode switch procedure in rich OS, and then depict how TrustICT instruments the mode switch procedure by inserting hooks. Next, we present the details on protecting the mode switch hooks.

4.3.1 Normal Mode Switch Procedure in Rich OS. On ARM platforms, the mode switch operations are triggered via exceptions, including the undefined instructions, supervisor calls, prefetch abort, data abort, and IRQ/FIQ interrupts. When encountering an exception, system will switch to kernel mode and jump to corresponding handler function through hardware mechanism. Each exception handler function can be addressed through the exception vector table. The execution is switched back to user mode when the handler function returns. Specifically, the mode switch procedure is as follows.

First, it locates virtual address of the exception vector table by reading the *System Control Register (SCTLR)*. If V-bit of SCTLR is set, the virtual address is fixed to 0xffff0000; otherwise, the address will be recorded in *Vector Base Address Register (VBAR)*. In Android OS, the virtual address of the exception vector table is usually fixed to 0xffff0000. Then, it obtains the physical memory of the exception vector table by leveraging the *Memory Management Unit (MMU)* to perform memory translation. Next, it obtains the virtual address of the handler function by retrieving the exception vector table, and finally locates the handler function's physical address via MMU.

In above procedure, the MMU relies on the virtual-to-physical mappings in specific page tables to perform the memory translation. The base address of page tables is stored in the *Translation Table Base Registers (TTBRs)* on most ARM platform. Generally, there are two TTBRs, i.e., TTBR0 and TTBR1. The N-bit of *Translation Table Base Control Register (TTBCR)* determines which register actually takes effect. In Android OS, the N-bit of TTBCR is usually set as 0, so only TTBR0 is used.

Moreover, although the execution codes of the handler functions are distributed among several memory pages, the entry points of all handler functions are normally concentrated in one memory page (hereinafter referred to as `Exception-Entry-Page`). The exception vector table is also allocated in one memory page (hereinafter referred to as `Exception-Vector-Page`). Further more, virtual and physical addresses of the `Exception-Entry-Page` and `Exception-Vector-Page` are set when the system boots up and will remain unchanged during the run time.

4.3.2 Instrumenting the Mode Switch Hooks. TrustICT interposes the mode switching operations by instrumenting the exception handler functions. Both U-K hooks and K-U hooks are implemented as the SMC instruction which switches to the secure domain. Specifically, TrustICT adds U-K hooks at the entry points of each exception handler function. The U-K hooks are naturally allocated in the same page accommodating the handler functions' entry points (i.e., the `Exception-Entry-Page`). The K-U hooks should be inserted at the end of each exception handler function. Unfortunately, the exception handler functions distribute in several different memory pages. To facilitate the control of K-U hooks, we instrument only one K-U hook and add a jump instruction to K-U hook at the end of each handler function. Both U-K hooks and K-U hook need to switch the execution to *DsM Permission Controller* in the secure domain, so that the access permission of the DsM regions could be securely revoked and granted, respectively. We allocate the code of the K-U hook in the same page as U-K hooks.

In a U-K hook, the execution will be switched back to the original exception handler function in the normal domain when the *DsM Permission Controller* finishes its execution in the secure domain. As for the K-U hook, since certain DsM regions might be unlocked, the *DsM Permission Controller* in the secure world will directly transfer the control flow to the user space of the CA, to prevent the unlocked DsM regions from being accessed by the malicious kernel. To achieve the exact transferring, TrustICT will record the address of the CA's user-space instruction interrupted by the U-K hook and resumes that instruction in the K-U hook.

4.3.3 Protecting Mode Switch Hooks from Being Tampered. Attackers with rich OS kernel privilege can directly modify the hooking codes, therefore we should provide real-time integrity protection on them. In rich OS (i.e., Android OS), the entry points of the exception handler functions will not change once loaded. Since the hooking codes are located in the same page as the entry points (i.e., the `Exception-Entry-Page`), we can prevent the codes from being tampered by setting the `Exception-Entry-Page` as readable and executable (but non-writable) to normal domain via TZASC.

4.3.4 Protecting Mode Switch Hooks from Being Bypassed. Since U-K hooks are responsible for invoking the *DsM Permission Controller* to close the access permission of all DsM regions, if they are deliberately bypassed, the access permission on DsM regions will not be closed securely. However, K-U hooks are the interface to unlock a DsM region, bypassing of them will make the to-be-unlocked regions remain locked, which will not damage the confidentiality or integrity of the communication data. Therefore, to protect DsM regions, we have to ensure that the U-K hooks cannot be bypassed when one or more DsM regions' access permission are unlocked.

U-K hooks may be bypassed in three ways through disrupting the procedure of the handler function locating. First, the attackers may directly modify the `Exception-Vector-Page` to manipulate the virtual addresses of exception-handler-function entry points. Second, the attackers may modify the values of exception-associated registers (i.e., SCTLR and VBAR), which are used to determine the virtual address of the `Exception-Vector-Page`. The attackers can provide a fake virtual address by modifying the exception-associated registers. Third, attackers may modify the page table entries utilized by MMU to perform virtual-to-physical memory translation for the `Exception-Entry-Page` and the `Exception-Vector-Page` (hereinafter referred to as `exception-associated-mappings`). Specifically, the attackers may directly tamper with exception-associated-mappings from kernel space, or change the permissions of exception-associated-mappings so that they could be tampered from user space. As such, fake physical addresses of exception vector table and handler-function entry points will be fetched.

For security, we need to protect exception-associated registers (i.e., SCTLR and VBAR), exception-associated-mappings and `Exception-Vector-Page` to prevent U-K hooks from being bypassed. TrustICT provides the run-time integrity protection for these critical values. The `Exception-Vector-Page` is protected by being configured as non-writable to the normal domain once the system boots up. The exception-associated registers and mappings are guarded as follows. Generally, values of the the exception-associated registers and mappings will remain unchanged once the rich OS boots up. TrustICT records the original values of the exception-associated registers and mappings. In K-U hook, before unlocking certain DsM regions, TrustICT will ensure values of the exception-associated registers and mappings are the same as the original ones and set the page tables accommodating the exception-associated-mappings as read-only to user space. After the K-U hook, the control flow will be switched to user space, where the exception-associated-mappings are read-only and the exception-associated registers could neither be modified since the corresponding instructions are only executable in kernel mode. As such, when the execution is switched from user space to kernel space, the U-K hooks will definitely be invoked since the value integrity of the exception-associated registers and mappings are ensured.

With the above protection mechanisms, the DsM regions will be securely locked to the kernel space, although the malicious kernel might manipulate the values of exception-associated registers and mappings. Specifically, all DsM regions are by default configured as inaccessible to normal world when the system boots up. The only way to unlock certain DsM region is through invoking the K-U hook, however we bind the DsM memory unlocking operations together with the value integrity protection of exception-associated registers and mappings in the K-U hook. As such, once certain DsM region is unlocked in the K-U hook, values of the exception-associated registers and mappings will be resumed to the correct ones. Then, the hardware components (i.e., CPU and MMU) will ensure the U-K hooks being invoked to lock all DsM regions when performing the user to kernel mode switch (i.e.,before the control flow is switched to kernel mode).

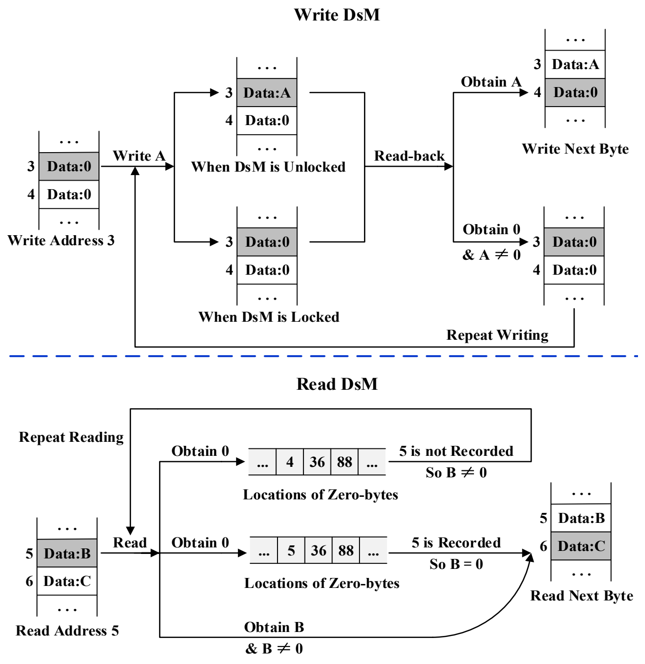

Figure 4: Polling Mechanism for Informing DsM States

4.4 Securely Informing CAs of DsM States

TrustICT needs to unlock the DsM regions only when none of the cores are running in the kernel mode of normal domain. However, it may cause problems to the CAs. Specifically, when CA_1 is accessing DsM regions on core_2, if core_1 is switching from user mode to kernel mode, all DsM regions will be locked. In this time, CA_1 has no idea of this and continues its read/write operations. The data read/written may be incorrect, since reading a locked DsM region only reads zero no matter what real values are, and writing to a locked DsM region causes no changes. To resolve this problem, TrustICT introduces two mechanisms that are both transparent to CA/TA developers.

Polling Mechanism. First, we introduce a polling mechanism by modifying the *DsM Read/Write APIs* in the TEE Clients. The basic idea is that the APIs will repeat the reading and writing operations until the data is correctly read or written. As illustrated in Figure 4, we add a read-back operation for each writing operation, and continue the writing operation until the read-back operation obtains the same value as the written one (it means the DsM memory is correctly written). For reading operations, obtaining a none-zero data means the DsM region is unlocked (i.e., the reading operation is successful). However, obtaining a zero data does not necessarily mean the DsM region is locked (i.e., the reading operation is failed), since the original data output by the TAs may be zero. To distinguish them, we modify the TEE OS to add location information on the data returned by the TAs (rather than transparently forwarding the data as in the original cross-domain communication procedure), which identifies the locations of bytes whose value is zero. Thus, the DsM Read APIs repeat the reading operation until the data read out is not zero, or the identified zero-bytes are read out.

One problem of Polling Mechanism is that TZASC may only be used to protect the DsM memory but not CPU cache. When verifying if the data written in is correct, the writing and reading back operations by default will process the data on the cache rather than memory. Therefore, the read-back operation may get correct data even if the DsM memory is inaccessible. There are two options to resolve this problem, i.e., setting the DsM memory as non-cacheable, or enforcing memory flush for each DsM writing operation. Our preliminary evaluation shows that the overhead caused by flushing is about eight times that of the cache-disabling solution. Therefore, TrustICT chooses to disable the cache for the DsM memory.

Dealing with the Kernel-stuck Cores. The polling mechanism ensures the correctness of the CA's reading and writing operations, and works well in most cases. However, the execution of related CAs may be stuck if certain cores run in the kernel mode for a long time, e.g., when the cores retain in the kernel-mode *idle* state for power saving or execute time-consuming system calls (e.g., `fork`). To resolve this problem, TrustICT transiently pauses the cores long-running in the kernel mode (hereinafter referred to as `kernel-stuck cores`) when one or more CAs are accessing the DsM memory, the details are as follows.

In U-K hooks, TrustICT will record the core state and the time that core enters kernel mode. When any core encounters a K-U hook, TrustICT updates core state and checks the duration of each core that resides in the kernel mode. If the duration is higher than a threshold, the associated core will be paused. The DsM accessing operations might still be stuck if no K-U hooks happen because a core may enter and retain in kernel mode. We solve it by adding a time monitor in the *DsM Read/Write APIs* and a "no-operation" system call in rich OS (i.e., the system call directly switches the context to user mode). If one DsM reading/writing operation is stuck for a certain time length, it will invoke the "no-operation" system call to trigger a K-U hook. The paused cores will be restarted when the DsM operations finish.

4.5 Defeating Attacks on Legal CAs

Attacks from malicious rich OS might also damage the cross-domain communication in three ways, even if the U-K/K-U hooks are enhanced and the DsM memory is securely locked to kernel. First, it could misuse the DsM memory allocated for the legal CA by either disguising as a legal CA or manipulating the execution of a legal CA. Second, it could double map a legal CA's DsM region to other user application. As such, the user application could illegally access the DsM region when it is unlocked. Third, it can manipulate the TEE driver to provide a fake address (i.e., not the physical address of the DsM memory written by CA) to TA. Due to the semantic gap between normal domain and secure domain, it is difficult for a TA to judge if it is a fake address. We develop countermeasures against all above attacks.

4.5.1 Defending against CA Manipulating or Disguising Attacks. TrustICT achieves this goal using three mechanisms. First, it performs code and data integrity check when a CA process (including both the user program and the shared libraries in it) is started, and sets legal CA's execution codes as non-writable to rich OS through TZASC. As such, the attackers could not access DsM through disguising as a legal CA or tampering with its execution codes. Second, it locks the legal CA's critical data memory (including the .data section and the user-space stack) in the U-K hook and unlocks it in the K-U hook, to avoid data breaches and tampering. For the heap memory, we provide two functions for the CA program to malloc (and free) heap memory from (and to) the protected DsM memory. This could prevent malicious kernel from manipulating the legal CA's user-space data. Third, to prevent the malicious kernel from manipulating the CA's execution flow by disturbing the execution context stored in the kernel stack (e.g., the values of *Link Register (LR)*, *Stack Pointer (SP)* and return address etc.), TrustICT records and hides the CA's context in the U-K hooks, and restores the values in the K-U hooks. The performance of CA may be affected due to locking of its critical data memory. Fortunately, in most case, the CA is only invoked when needing to interact with the TA. Therefore, it will not cause much overhead to rich OS.

Specifically, we locate the CA process' code and data in memory through two steps. First, we get the virtual address and memory range of the segments including `code` and `data` sections through parsing the ELF-formatted CA program file. Then, we get the corresponding physical addresses with the help of the page tables located through TTBR0 value. On ARM platforms, the TTBR0 register stores the base address of a running process' page table set. As for the user space stack, we first derive the CA process' user space memory layout through the page tables, and then locate the `stack` section which is continuous and resides at the end of the memory layout. When a CA process is started, TrustICT will check whether the Program Counter Pointer and Stack Pointer are located in the `code` and `stack` sections respectively, before transferring control flow to the CA in the K-U hook. Meanwhile, it sets the memory accommodating the page table mappings of these sections as non-writable to normal world.

The TTBR0 value assigned for each running process is unique and will not be changed until the process runs to completion. Moreover, TrustICT introduces run-time protection for the data associated with a TTBR0 value (i.e., a running process' data, code and page table set). Therefore, we can use TTBR0 value to represent a CA process. To quickly judge whether the running process is a legal CA, TrustICT maintains the legal CA processes' TTBR0 values in the secure world, and identity a legal CA through comparing the TTBR0 value when performing the checks in the mode switch hooks.

4.5.2 Defending against Double Map Attacks. Before unlocking a CA's protected memory (including DsM memory and critical user space data memory like stack, .data section), TrustICT will check if the memory is double mapped by any processes running on other cores than the core this CA is being executed. Specifically, it first locates the processes' page tables by reading the corresponding core's TTBR0. Then, it goes through the page table entries to check if the protected memory is double mapped. It is not necessary to check each page table entry, which might be time-consuming. In rich OS, the page tables are organized in two levels. Level-1 page tables are used to locate the level-2 page tables, while level-2 page tables are used for locating the real memory accessed by a process. And a permission-bit in the level-1 page tables indicates

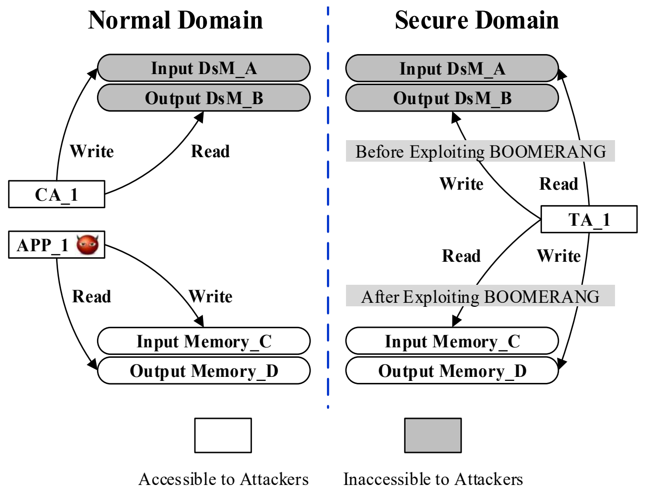

Figure 5: BOOMERANG-based Attack

if the associated memory belongs to kernel space or user space. The processors will prevent the kernel-space memory from being accessed from user space.

Since the protected memory is only unlocked when all cores are running in the user mode, we only need to prevent the double mapping of user-space memory. Instead of traversing all page table entries, TrustICT first checks level-1 page tables, and ensures the permission-bit of all designated kernel-space memory are correctly set. Then, it checks the entries of level-2 user-space page tables to ensure the protected memory is not double mapped to user space. To prevent the rich OS from hiding the double maps in the Translation Lookaside Buffer (TLB) cache (rather than memory), TrustICT will flush the associated TLB entries. According to our evaluation, it needs about 200 µs to check one process. And the double map checking will be triggered in the K-U hooks only when certain protected memory region needs to be unlocked.

4.5.3 Defending against Semantic Gap Vulnerability. As mentioned in Section 4.1, the physical address of DsM memory written by CA will be informed to TA via the TEE driver. Due to the semantic gap between normal domain and secure domain, it is difficult for a TA to judge if it is a fake address. This is also the primary cause of the BOOMERANG attacks [40]. As illustrated in Figure 5, DsM_A and DsM_B are allocated for CA_1 as the input memory and output memory, and CA_1 writes data in to DsM_A and reads return values from DsM_B. However, attackers can control TEE Driver to inform Memory_C and Memory_D to TA_1, then TA_1 will read and write through these memory. TrustICT resolves this problem by introducing an additional check in the TEE OS. Specifically, before the DsM memory is accessed by a TA, the *DsM Access Checker* module will check if the addresses informed by TEE driver match certain ones recorded in the *DsM-CA Mapping Maintainer*, and if the TA invocation is initiated by the corresponding CA. The paper introducing BOOMERANG also provides the defensive method, *Cooperative Semantic Reconstruction* [40]. Such method relies on the trusted kernel while our design defends against untrusted kernel.

5 EVALUATION

We evaluate TrustICT by conducting extensive experiments on the prototype implemented on the FreeScale i.MX6Quad development board. The board is equipped with a quad-core ARM Cortex-A9 processor running at 1.2GHz with 1GB DDR3 SDRAM. The secure world is deployed with the OP-TEE OS 2.2.0, and the normal world is installed with a FreeScale Android 6.0.1 system with a 4.1.15 Linux kernel. In total, TrustICT adds about 210 source line of code (SLOC) in the normal domain, and about 2690 SLOC is added in the secure domain. To minimize the noise involved during our experiment, we run each test with 1,000 iterations and take the average as our measures.

To better illustrate the performance of TrustICT, we provide a comparison between TrustICT and SeCReT [33] in the evaluation. SeCReT [33] also focuses on constructing a trusted cross-world communication channel, and it mainly contains two modules, i.e., the kernel protection module which ensures real-time integrity of rich OS kernel codes through interposing page table operations and the data protection module which protects the communication data through encryption/decryption operations. Since the source codes of SeCReT are unavailable, we implement a prototype of SeCReT (including both the kernel protection module and the data protection module) according to their paper (hereinafter referred to as the SeCReT-like system).

5.1 Impacts on Rich OS

The primary performance impacts introduced by TrustICT on rich OS are due to hooking the mode switch operations, which introduces extra cross-domain switching operations and additional processing tasks in the secure world such as recording the core states (see Section 4.4) and restoring exception-associated-mappings and registers (see Section 4.3.4). Also, the kernel-stuck cores might be transiently paused when one or more CAs are accessing the DsM memory (see Section 4.4). SeCReT's main impacts on rich OS are caused by the kernel protection module which adopts the real-time kernel protection technology TZ-RKP [10] to protect the kernel hooking codes. Besides the overall overhead, overhead on the application loading time and the overhead of the system call invocations evaluated in the SeCReT and TZ-RKP papers, we additionally test the probability for the cores to be transiently paused to learn the influence of multi-core in our design.

5.1.1 Overall Overhead. We first study the overall performance impacts on the rich OS through a comprehensive benchmark suite AnTuTu 2.9.4 [5]. The scores for the original system, TrustICT and the SeCReT-like system are 3358, 3289 and 3247 respectively. The higher score means the better performance. The SeCReT-like system introduces about 3% overhead, which is in consonance with the results in the TZ-RKP paper [10]. TrustICT reduces the performance loss by about one percent. This is mainly because we avoid the frequent page table operation checking and achieve real-time protection of hooking codes through dynamically setting the TZASC registers (Section 4.3). Moreover, TrustICT could be further optimized by disabling the real-time protection when no legal CAs are running, then it will not introduce overhead on the rich OS. But the protection in SeCReT should be always activated during the execution of rich OS.

Table 1: Overhead of System Call Invocations on Rich OS (in *us*)

System Call	Original	TrustICT (With CA Not Access DsM)	Overhead	SeCReT-like (With CA Not Access DsM)	Overhead	TrustICT (With CA Access DsM)	Overhead	SeCReT-like (With CA Access DsM)	Overhead
Open	47.2	109.4	2.3178x	111.3	2.3581x	323.1	6.8453x	111.8	2.3686x
Close	23.3	90.6	3.8884x	88.3	3.7897x	305.2	13.0987x	88.9	3.8155x
Read	29.5	92.4	3.1322x	94.8	3.2136x	303.2	10.2780x	96.3	3.2644x
Write	174.1	244.5	1.4044x	242.2	1.3912x	436.4	2.5066x	244.6	1.4049x
Malloc	30.2	86.3	2.8576x	89.5	2.9636x	302.3	10.0099x	90.1	2.9834x
Free	80.8	179.8	2.2252x	183.7	2.2735x	413.6	5.1188x	185.7	2.2983x
Fork	3890.9	4828.6	1.2410x	4783.6	1.2294x	14709.3	3.7804x	4793.9	1.2321x
Send	377.2	611.8	1.6220x	643.1	1.7049x	828.2	2.1957x	654.7	1.7357x
Recv	46.6	110.2	2.3648x	106.3	2.2811x	311.8	6.6910x	111.2	2.3863x
Connect	1489.4	1916.1	1.2865x	1893.8	1.2715x	6989.5	4.6928x	1912.7	1.2842x

Table 2: Overhead of App Loading Time (in Seconds)

Test Item	Original	TrustICT	Overhead	SeCReT-like	Overhead
Calculator	3.01	3.09	2.66%	3.24	7.64%
Calendar	3.14	3.21	2.23%	3.36	7.01%
Music	1.26	1.31	3.97%	1.38	9.52%
Settings	3.77	3.86	2.39%	3.97	5.31%

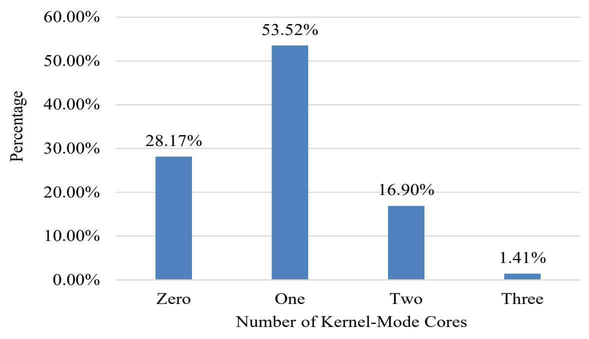

Figure 6: State of the Cores When Running System-Call-Intensive Workloads

5.1.2 Overhead of System Call Invocations. TrustICT's primary impact on rich OS lies in the system call invocations, and the overhead might vary when the system is running in different execution contexts. For example, certain system call invocation might be transiently suspended when a CA is accessing DsM memory. To better illustrate the overhead introduced by TrustICT, we evaluate the performance of system call invocations in three different scenarios, i.e., when ① the original system is running, ② TrustICT or SeCRet-like system is enabled with a CA running, but the CA is not accessing DsM, ③ TrustICT or SeCRet-like system is enabled with a CA continuously accessing DsM. In total, we test ten frequently used system calls including `open`, `close`, `read`, `write`, `malloc`, `free`, `fork`, `send`, `recv` and `connect`. For `read`, `write`, `malloc`, `free`, `send` and `recv` system calls, we call them to process 4KB data or memory, respectively.

The experimental results are illustrated in Table 1. When TrustICT is enabled with a CA running but the CA is not accessing DsM (i.e., scenario ②), most of the system call invocations are slowed down by tens of microseconds, due to two extra cross-domain context switch operations triggered by hooks. The extra operations conducted to protect the CA's control flow in each hook when a CA program is running (as illustrated in Section 4.5.1). Some complex system calls may trigger extra exception handler functions to be executed, thus introducing higher overhead. For example, the `fork` system call will trigger several additional page fault exceptions. The SeCReT-like system has similar results in scenario ②, since it also has two extra cross-domain context switch operations triggered by hooks.

When a CA is accessing DsM (i.e., scenario ③), about 200µ*s* overhead is additionally introduced to most of the system call invocations in TrustICT, which is caused by the double map checking enforced in the K-U hooks before unlocking the DsM (as depicted in Section 4.5.2). Since most kernel operations (e.g., system calls) could be finished in less than 1 millisecond, we set the threshold for identifying kernel-stuck cores as 1 millisecond (referred to Section 4.4). Therefore, the higher overhead in `fork` and `connect` is caused by the transient pausing of the core. The results of the SeCReT-like system have no big difference for the scenarios ② and ③, since SeCReT mainly focuses on defending against the attacks on the single-core platforms and introduces no protection for the multi-core platforms.)

5.1.3 Overhead of Application Loading Time. As illustrated in Table 2, we test the loading time of four Android applications, including Calculator, Calendar, Music and Setting. The overheads introduced by the SeCReT-like system and TrustICT are about 7% and 3% respectively. Also, the overhead of TrustICT could be avoided when no legal CAs are running by disabling the protection.

5.1.4 Probability for Cores to be Transiently Paused. In TrustICT, certain cores might be transiently paused when a CA is running, and it mainly depends on the possibility for the cores to be executed in the kernel mode and the time duration of the kernel operations. The cores will only be paused during a CA is accessing the DsM, and more kernel operations will be involved when the system is executing system-call-intensive jobs. Therefore, we investigate the state of the cores when the system is running a CA program and four additional system-call-intensive tasks (i.e., an audio player, an image viewer, a user program continually sending and receiving packages and an AES encryption program). The CA program continuously initiates the DsM accessing operations through TA invocations. The results are illustrated in Figure 6. It shows that in more than half of the time, only one core is executing in the kernel mode, and the chance for all three cores (except the core running the CA program) to be executed in kernel mode is only 1.14%. Since 90% kernel operations could be finished in 1 millisecond, it is about 0.114% percentage for all the other three cores to be paused. Actually, this will hardly happen, since the cores will only be transiently paused when the CA is accessing DsM memory. Since SeCReT doesn't provide protection for the multi-core scenario, we don't perform this test for it.

5.2 Impacts on Cross-domain Transaction

The cross-domain communication is launched through TA invocations, which are initiated during the execution of the CAs. Therefore, we first evaluate the slow down on the execution of a CA. We test seven CA-TA pairs shipped with the OP-TEE source codes, i.e., AES Encrypt, AES Decrypt, SHA-256, AES-256 ECB Encrypt, AES-256 ECB Decrypt, Secure File Storage and Key Derivation. A normal TA invocation involves many operations, such as allocating and freeing DsM memory, performing cross-domain context switch, loading TA images, and reading and writing DsM memory, etc, which usually takes hundreds of milliseconds. When the TrustICT is enabled, the CA will perform code integrity check before performing data interaction. And when the CA is running, it may be suspended when other cores switch to kernel mode. As shown in Table 3, TrustICT imposes about 60% overhead on the CA execution while the value for SeCReT-like system is about 35%. The primary reason for the extra overhead in TrustICT is because it needs to defeat the attacks launched from other cores on the multi-core platforms, while SeCReT only ensures the security on single-core platforms. To simulate the single-core scenario, we close the protection mechanisms used to defeat attacks from other cores (i.e., the double map checking and polling mechanisms), thereafter the overhead of TrustICT reduces to 18%. The efficiency of TrustICT can be further improved by lowering the threshold used to identify the kernel-stuck cores as described in Section 4.4, but this might in contrary increase the overhead on rich OS. In our scheme, *Poly1305* [53] algorithm is utilized to perform the code integrity check for better performance.

Table 3: Overhead of CA Execution (in *us*)

CA-TA Pair	Original	TrustICT	Overhead	SeCReT-like	Overhead
AES Encrypt	176873.3	298876.5	68.98%	246786.3	39.53%
AES Decrypt	187322.6	316754.3	69.10%	258265.3	37.87%
SHA-256	185786.2	276378.1	48.76%	251724.1	35.49%
AES-256 ECB Encrypt	188339.5	309729.8	64.45%	258275.2	37.13%
AES-256 ECB Decrypt	187904.2	287623.3	53.07%	257169.8	36.86%
Secure File Storage	177432.5	283267.2	59.65%	236871.6	33.50%
Key Derivation	165223.7	269876.9	63.34%	228652.8	38.39%

Furthermore, we study the performance overhead on the direct DsM reading and writing operations in the CAs [3]. Specifically, we calculate the time used to write and read a piece of DsM memory, with the memory size increasing from 128 bytes to 4096 bytes, on three different conditions, i.e., when the memory is not protected, when it is protected by TrustICT and SeCReT-like systems. To simulate the worse cases, the overhead introduced by TrustICT is evaluated when the rich OS is running system-call-intensive tasks (i.e., an audio player, an image viewer, a user program continually sending and receiving packages and an AES encryption program). The results are illustrated in Table 4. When protected by TrustICT, the running time increases in tens to thousands of microseconds as the payload increases. The overhead is mainly caused by disabling the CPU cache and the polling mechanism, as described in Section 4.4. When the rich OS is not running system-call-intensive workloads, the overhead could be further reduced (e.g., when the four system-call-intensive tasks are paused, the overhead reduces about 50% to 75%). As a comparison, SeCReT-like system increases in several to tens of milliseconds due to the expensive encryption operations [4]. As such, the overhead of TrustICT is much smaller than that of cryptographic-based solutions.

Table 4: Overhead of DsM Read-Write Operations on CAs (in *us*)

Payload (Bytes)	Original		TrustICT		SeCReT-like	
	Write	Read	Write	Read	Write	Read
128	14.3	14.2	212.7	79.3	790.6	2486.3
256	16.2	16.4	360.3	157.3	1309.0	4002.9
512	22.5	20.8	725.8	291.4	2365.3	6733.7
1024	31.2	27.3	1454.2	638.5	3894.7	12937.5
2048	48.1	42.4	3008.6	1154.6	7498.5	25067.2
4096	85.3	79.7	4499.1	2393.1	15338.9	45389.9

TrustICT mainly achieves the protection through hooking and performing security checks during the user and kernel mode switching in rich OS, which will hardly introduce extra overhead on the operations in the secure domain. Therefore, we do not evaluate its impacts on the secure domain. In summary, our design surpasses the solutions which rely on the real-time kernel protection mechanism and the encryption/decryption operations (e.g., SecReT) in multiple items, such as overall performance, application loading times, CA execution and DsM read-write operations etc.

6 DISCUSSION AND FUTURE WORK

Before the data is inputted into the DsM and processed by TAs, it may be produced and pre-processed in the CAs (hereinafter referred to as `Pre_DsM_Data`). Malicious OS may disrupt the `Pre_DsM_Data` through manipulating the execution of legal CAs. In TrustICT, we defend against this by locking the legal CAs' critical user-space memory (including code, stack, data section etc.) to the kernel and hiding the CAs' execution context in the U-K hooks. This solution works well when the processing of `Pre_DsM_Data` could be accomplished entirely inside the CAs without the help of kernel. In the situation when the pre-processing needs to invoke kernel functions (e.g., the signal-handling function), we should allow the kernel to write portions of the CAs' user space memory (e.g., the stack). There are two approaches to protect the function invocation

[3]In the SecReT paper, the performance overhead on cross-domain transaction is only evaluated through the direct DsM reading and writing operations in the CAs.

[4]The same experiments show more performance loss in the SeCReT paper (e.g., it takes almost a second to operate 4096 bytes), which might be because the different experiment platforms. Also, some extra protection measures in SecReT are not implemented in our SeCReT-like prototype.

between legal CAs and malicious OS. The first one is monitoring each invocation and allowing only secure ones (e.g., the invocations without leaking CAs' sensitive data) to be performed. Similar ideas are leveraged in the solutions such as Trustshadow [31] and Overshadow [17]. The second one is introducing a well isolated micro-kernel as an aid to the legal CAs, just as the implementation of SANCTUARY [15]. In this paper, we focus more on protecting the cross-domain communication channel. We leave it as our future work to design a more complete CA protecting scheme.

7 RELATED WORK

A line of research work focuses on providing real time integrity protection on the critical codes. *HyperSentry* [11] utilizes SMM (System Management Mode) to securely run the monitoring code to check the integrity of the hypervisor. *NICKLE* [50] achieves the real-time integrity protection of kernel codes by controlling the shadow physical memory through hypervisor. *TZ-RKP* [10], *SPROBES* [26], and *SKEE* [13] are solutions towards the ARM platforms, which provide real-time integrity protection by interposing and monitoring each page table operations. Instead of relying on the expensive page table monitoring operations, we leverage a common hardware component TZASC available on most ARM platforms to achieve a lightweight code integrity protection.

A number of research works devote to protect sensitive data through cryptographic technologies. Guan et al. propose two solutions [29, 30], which provide sensitive data protection against physical memory disclosure attacks by locking the plaintext in the cache and illustrating only ciphertext in the memory. *CaSE* [62] is another cache-based solution that further defends against the software attacks by leveraging the isolation provided by ARM TrustZone extension. *Overshadow* [17] and *InkTag* [32] encrypt address space of a sensitive application through a hypervisor, so that a compromised OS can only view the address space of the application in ciphertext. Cryptographic technologies are also adopted by popular mobile OSes (e.g., Android and iOS) to protect the data on the file system [3, 6]. Cryptographic-based solutions introduce non-negligible overhead, especially on the power-constrained mobile platforms.

Besides using cryptographic technologies, system level technologies are utilized for protecting the sensitive data. *Virtual Ghost* [21] prevents the attackers from accessing the protected memory area of applications by using compiler-based instrumentation. It requires the operation system to be recompiled. Several solutions [41, 42, 60] take advantage of the high-privileged hypervisor to enable secure execution and data secrecy for pieces of application logic. Hypervisor-based solution may not provide the best performance for the resource-constrained mobile platforms. In addition, the hypervisor is already struggling with its own security problems due to increasing TCB size [23, 24]. In this work, TrustICT utilizes the TrustZone technology to shield applications from untrusted OS kernel, eliminating complex, error-prone resource allocation in a hypervisor.

Hardware-assisted protection is also widely explored to shield the applications from untrusted OSes. *Intel Software Guard eXtension (SGX)* [43] has been adopted by many solutions to secure the application execution on the Intel platforms [14, 19, 20, 56]. *SICE* [12] protects sensitive workloads running on x86 platforms through SMM. For the mobile devices running on ARM processors, TrustZone technology is widely utilized for shielding applications (e.g., [31, 52, 57, 58]). *TrustShadow* [31] guarantees secure execution of unmodified applications. *TrustOTP* [57] is a solution realizing trusted display of one-time passwords. *TrustICE* [58] creates isolated computing environments in the normal domain. Trusted Language Runtime (TLR) [52] protects the confidentiality and integrity of .NET mobile applications. These systems were implemented without considering a secure cross-domain communication channel. SANCTUARY [15] implements a secure channel but it relies on ARM Fast Models virtualization tools which is not supported by hardware. Ginseng [61] utilizes registers to protect secrets which depends on page table monitoring operations and the size of the protected data is limited. *SeCReT* [33] is the most close work to our solution on securing the data interaction between normal and secure domain. SeCReT leverages the time-consuming cryptographic operations and protects the rich OS kernel's static region via hooking and monitoring each page table operation, where both approaches may introduce heavy system overhead. Moreover, SeCReT only works well on single-core platforms. TrustICT is a lightweight solution on the multi-core platforms, and it achieves a trusted cross-domain communication channel by protecting the access to the DsM via TZASC, rather than relying on the expensive real-time kernel protection solutions (e.g., TZ-RKP [10]) and heavy encryption/decryption operations.

8 CONCLUSIONS

In this paper, we develop a system named TrustICT to secure the data communication between CAs and TAs on the multi-core platforms. Different from existing approaches that rely on heavy cryptography operations, TrustICT protects the cross-domain communication by dynamically setting the access permission of the domain-shared memory (DsM), ensuring that the DsM memory could only be accessed from user space of the normal domain or the secure domain, but not the untrusted kernel space of normal domain. TrustICT only needs to configure a small number of CPU registers, and it is more effective than the schemes based on cryptography which makes it can well adapt to the frequent mode switching on multi-core platforms. We systematically study the potential attacks on multi-core platforms such as double map attacks, semantic gap based attacks etc., and then design and implement corresponding countermeasures. The evaluation results on a hardware testbed show that TrustICT is effective and incurs a small overhead on rich OS.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd for their insightful comments on improving our work. This work is partially supported by National Science and Technology Major Project of China under Award No. 2018ZX03001-010, the National Natural Science Foundation of China under GA No. 61802398, the National Cryptography Development Fund under Award No. MMJJ20180222 and MMJJ20170215, and the NSF grant CNS-1815650.

REFERENCES

[1] 2013. CVE Details. Google: Android: Security Vulnerabilities. http://cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html.

[2] Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *ARM white paper* 3, 4 (2004).

[3] Android. 2018. File-Based Encryption | Android Open Source Project. https://source.android.com/security/encryption/file-based.

[4] Android. 2018. System and kernel security | Android Open Source Project. https://source.android.com/security/overview/kernel-security.

[5] ANTUTU. 2019. Aututu Benchmark. http://www.antutu.com/en/index.htm.

[6] Apple. 2018. Keychain Services. https://developer.apple.com/documentation/security/keychain-services.

[7] ARM. 2010. TrustZone Address Space Controller (TZC-380) Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0431c/DDI0431C_tzasc_tzc380_r0p1_trm.pdf.

[8] ARM. 2016. ARM Cortex-M23 Processor Technical Reference Manual. https://developer.arm.com/documentation/ddi0550/c.

[9] ARM. 2016. ARM Cortex-M33 Processor Technical Reference Manual. https://developer.arm.com/documentation/100230/latest.

[10] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, and Jia Ma. 2014. 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 (CCS), 2014*.

[11] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. 2010. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *ACM Conference on Computer and Communications Security*. 38–49.

[12] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 2011. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *ACM Conference on Computer and Communications Security*. 375–388.

[13] Ahmed M. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24*.

[14] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[15] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. 2019. SANCTUARY: ARMing TrustZone with User-space Enclaves.. In *NDSS*.

[16] Ferdinand Brasser, Daeyoung Kim, Christopher Liebchen, Vinod Ganapathy, Liviu Iftode, and Ahmad-Reza Sadeghi. 2016. Regulating arm trustzone devices in restricted spaces. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. 413–425.

[17] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. (2008), 2–13.

[18] Yue Chen, Yulong Zhang, Zhi Wang, and Tao Wei. 2018. Downgrade attack on TrustZone. https://arxiv.org/pdf/1707.05082.

[19] Yuxia Cheng, Qing Wu, Bei Wang, and Wenzhi Chen. 2017. Protecting In-memory Data Cache with Secure Enclaves in Untrusted Cloud. In *proceeding of International Symposium on Cyberspace Safety and Security*.

[20] Victor Costan, lia A Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *proceeding of usenix security symposium*.

[21] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 81–96.

[22] CVEdetail.com. 2013. cve-2013-3051. http://www.cvedetails.com/cve/CVE-2013-3051/.

[23] CVEdetails.com. 2018. Vmware: Vulnerability statistics. http://www.cvedetails.com/vendor/252/Vmware.html.

[24] CVEdetails.com. 2018. Xen: Vulnerability statistics. http://www.cvedetails.com/vendor/6276/XEN.html.

[25] Jan-Erik Ekberg. 2015. Trusted Execution Environments (and Android). https://usmile.at/sites/default/files/androidsecuritysymposium/presentations2015/Ekberg_AndroidAndTrustedExecutionEnvironments.pdf.

[26] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. 2014. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In *in Proceedings of the 2014 Mobile Security Technologies (MoST) workshop*.

[27] GlobalPlatform. 2017. GlobalPlatform made simple guide: Trusted Execution Environment (TEE) Guide. https://www.globalplatform.org/mediaguidetee.asp.

[28] GlobalPlatform. 2018. TEE System Architecture v1.1. https://globalplatform.org/specs-library/.

[29] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. 2014. Copker: Computing with Private Keys without RAM. In *network and distributed system security symposium*.

[30] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. 2015. Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory. In *Proceedings of ieee symposium on security and privacy*.

[31] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. 2017. TrustShadow: Secure execution of unmodified applications with ARM trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 488–501.

[32] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: secure applications on an untrusted operating system. In *ASPLOS*. 265–278.

[33] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. 2015. SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment. In *Proceeding of network and distributed system security symposium(NDSS)*.

[34] N. Keltner. 2014. Here Be Dragons: Vulnerabilities in TrustZone. https://atredispartners.blogspot.com/2014/08/here-be-dragons-vulnerabilities-in.html.

[35] K. Lady. 2016. Sixty Percent of Enterprise Android Phones Affected by Critical QSEE Vulnerability. https://duo.com/blog/sixty_percent_of_enterprise_android_phones_affected_by_critical_qsee_vulnerability.

[36] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. 2018. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 1–13.

[37] Linaro. 2018. Leading software collaboration in the Arm Ecosystem. https://www.linaro.org/membership/.

[38] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. 2012. Software abstractions for trusted sensors. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 365–378.

[39] Renju Liu and Mani Srivastava. 2018. VirtSense: Virtualize Sensing through ARM TrustZone on Internet-of-Things. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 2–7.

[40] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environment. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[41] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*. 143–158.

[42] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: an execution infrastructure for tcb minimization. In *EuroSys*. 315–328.

[43] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *HASP@ ISCA* 10 (2013).

[44] NORDIC. [n.d.]. nRF5340. https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF5340.

[45] NXP. [n.d.]. i.MX-RT500. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt500-crossover-mcu-with-arm-cortex-m33-core:i.MX-RT500.

[46] NXP. [n.d.]. i.MX-RT600. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/i-mx-rt-crossover-mcus/i-mx-rt600-crossover-mcu-with-arm-cortex-m33-and-dsp-cores:i.MX-RT600.

[47] OP-TEE. 2018. OP-TEE design. https://github.com/OP-TEE/optee-os/blob/master/documentation/optee-design.md.

[48] OP-TEE. 2018. optee-os. https://github.com/OP-TEE.

[49] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[50] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 1–20.

[51] Dan Rosenberg. 2013. Unlocking the motorola bootloader. http://blog.azimuthsecurity.com/2013/04/unlocking-motorola-bootloader.html.

[52] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. 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*. 67–80.

[53] SiteGround. 2018. Poly1305. https://www.poly1305.com/.

[54] Stephen Smalley and Robert Craig. 2013. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of network and distributed system security symposium(NDSS)*.

[55] STMicroelectronics. [n.d.]. STM32L5. https://www.st.com/en/microcontrollers-microprocessors/stm32l5-series.html.

[56] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *Proceeding of usenix security symposium*.

[57] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. TrustOTP: Transforming Smartphones into Secure One-Time Password. In *Proceeding of ACM computer and communications security (CCS)*.
[58] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. 2015. TrustICE: Hardware-Assisted Isolated Computing Environments on Mobile Devices. In *Proceeding of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[59] Common Vulnerabilities and Exposures. 2020. CVE List. https://cve.mitre.org/.
[60] Jisoo Yang and Kang G Shin. 2008. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceeding of virtual execution environments*.
[61] Min Hong Yun and Lin Zhong. 2019. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System.. In *NDSS*.
[62] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. 2016. CaSE: Cache-Assisted Secure Execution on ARM Processors. In *ieee symposium on security and privacy*. 72–90.