Cache-in-the-Middle (CITM) Attacks : Manipulating Sensitive Data in Isolated Execution Environments

Jie Wang[1,2,3], Kun Sun[2], Lingguang Lei[1,3], Shengye Wan[4], Yuewu Wang[1,3], and Jiwu Jing[5]
[1]SKLOIS, Institute of Information Engineering, CAS, China
[2]Department of Information Sciences and Technology, George Mason University
[3]School of Cyber Security, University of Chinese Academy of Sciences, China
[4]Department of Computer Science, College of William and Mary
[5]School of Computer Science and Technology, University of Chinese Academy of Sciences
{wangjie, leilingguang, wangyuewu}iie.ac.cn, ksun3@gmu.edu, swan@email.wm.edu, jwjing@ucas.ac.cn

ABSTRACT

The traditional usage of ARM TrustZone has difficulty on solving the conflicts between the manufacturers that want to minimize the trusted computing base by constraining the installation of third-party applications in the secure world and the third-party application developers who prefer to have the freedom of installing their applications into the secure world. To address this issue, researchers propose to create Isolated Execution Environments (called IEEs) in the normal world to protect the security-sensitive applications. In this paper, we perform a systematic study on the IEE data protection models and the ARM cache attributes, and discover three cache-based attacks called CITM that can be leveraged to manipulate the sensitive data protected in IEEs. Specifically, due to the inefficient and incoherent security measures on the cache that maps to the IEE memory (i.e., memory designated for IEEs), attackers in the normal world may compromise the security of IEE data by manipulating the IEE memory during concurrent execution, bypassing the security measures enforced when a security-sensitive application is suspended or finished, or misusing the incomplete security measures during IEE's context switching processes. We conduct case studies of CITM attacks on three well-known IEE systems including SANCTUARY, Ginseng, and TrustICE to illustrate the feasibility to exploit them on real hardware testbeds. Finally, we analyze the root causes of the CITM attacks and propose a countermeasure to defeat them. The experimental results show that our defense scheme has a small overhead.

CCS CONCEPTS

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

KEYWORDS

TrustZone; Isolated Execution Environment; Cache Manipulation

CCS '20, November 9–13, 2020, CCS, virtual

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Jie Wang[1,2,3], Kun Sun[2], Lingguang Lei[1,3], Shengye Wan[4], Yuewu Wang[1,3], and Jiwu Jing[5]. 2020. Cache-in-the-Middle (CITM) Attacks : Manipulating Sensitive Data in Isolated Execution Environments. In *CCS '20: ACM Symposium on Computer and Communications Security, November 9–13, 2020, CCS, virtual.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

ARM TrustZone has become one popular security technology to protect security-sensitive applications in an isolated trusted execution environment [1]. Many manufacturers enable the TrustZone-based protection on their commercial mobile devices [2, 25, 28, 34]. As a hardware-assisted technology, TrustZone divides platform into two execution environments, namely, the *normal world* (or *non-secure world*) and the *secure world*, where the normal world is responsible for running normal applications over an rich OS and the secure world is preserved to protect security-sensitive code and data.

The traditional usage of TrustZone is to run the security-sensitive applications and store their sensitive data in the secure world, as shown in Figure 1(a). Those solutions are usually called *Trusted Execution Environment (TEE)* systems, where the security-sensitive applications are implemented as *Trusted Applications (TAs)* running in the secure world (e.g., *SAMSUNG KNOX* [25], *OP-TEE* [47], *Qualcomm QSEE* [28], and *Huawei Secure OS* [34]). The TEE solutions rely on built-in hardware supports to enforce a secure isolation and defend against attacks from untrusted rich OS.

Although the traditional TEE systems can ensure a strict isolation and secure protection of the security-sensitive applications in the secure world, they have difficulty on addressing two conflicting requirements from device manufacturers and third-party security-sensitive application developers. On one side, since the *trusted computing base (TCB)* of the TEE systems keeps increasing along with the number of applications installed in secure world, the manufacturers are reluctant to open the secure world for freely installing third-party applications. Instead, they prefer to only install their own security-sensitive applications that may have gone thorough a more strict security assessment. On the other side, more third-party security-sensitive applications expect to be imported into the secure world for an enhanced security protection, which is critical to foster an ecosystem for more third-party application developers to develop their security-sensitive applications for the TEE systems.

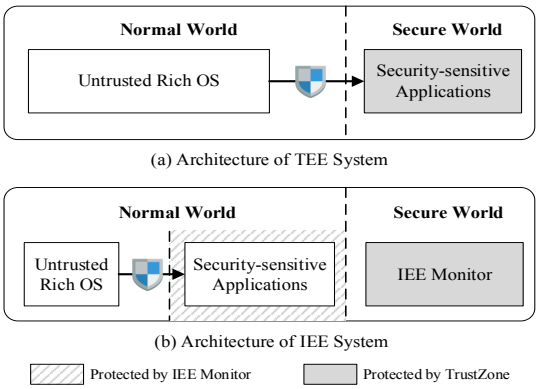

Figure 1: TEE System vs. IEE System

To address the above problem in TEE systems, researchers propose to create *Isolated Execution Environments* (called *IEEs*) in the normal world [13, 54, 55], as shown in Figure 1(b). The key idea is to use a trusted reference monitor (i.e., the IEE monitor) in the secure world to ensure that only the authorized IEE application (i.e., the security-sensitive application running in the IEE) can access the related IEE sensitive resources, protecting the security-sensitive application in one IEE from other IEEs and the rich OS. For example, TrustICE [54] relies on the IEE monitor to protect IEE memory via dynamically controlling the security attribute of the IEE memory. Ginseng [55] constructs the IEEs to protect secrets of third-party applications in the normal world without deploying any application-specific logic in the secure world. SANCTUARY [13] allocates its IEEs as the per-core environments to protect the memory designated for IEEs (hereinafter referred to as IEE memory) in the normal world from being accessed by any other non-secure cores. In this paper, we use "switch out" to denote the process of one core's context switching from IEE to untrusted rich OS, and "switch in" to represent the reverse process on each core.

Compared with TEE systems, the IEE systems can minimize the TCB of the secure world by moving security-sensitive applications into the normal world and only installing an IEE monitor in the secure world. Also, IEE systems can achieve a better portability since the security-sensitive applications are installed and executed in the normal world. However, since the security-sensitive applications are isolated and protected by a software component (i.e., the IEE monitor), IEE systems may not achieve the same level of security protection as TEE systems.

In this paper, we conduct a systematic study on the existing IEE data protection models and the ARM cache attributes, and discover three new cache-based attacks called CITM that can be leveraged to manipulate the sensitive data protected in the IEEs (hereinafter referred to as IEE data). First, the attackers may manipulate the IEE data through cross-core cache operation during concurrent execution. On multi-core platforms, it is not secure to only ensure core-wise isolation on the IEE memory to defend against the concurrently running untrusted rich OS [13], since the cache may be still open for the cross-core access and thus manipulated by attackers. Second, when a security-sensitive application is suspended or finished (i.e., during the "switch out" process), the IEE system must conduct several security measures to protect the IEE data from later being accessed by the rich OS [55]. However, the attackers may bypass these security measures by manipulating the non-secure cache mapping to the IEE memory used for the security measures. Third, when the IEE memory protection is achieved by dynamically controlling the security attribute of the memory (e.g., configuring it as non-secure before "switch in" and as secure before "switch out" [54]), due to incomplete security protection on cache, attackers may steal sensitive IEE data during the "switch out" process and tamper with IEE data during the "switch in" process.

We conduct case studies of CITM attacks against three IEE systems including SANCTUARY [13], Ginseng [55], and TrustICE [54]. The experimental results show that attackers may successfully steal and modify IEE data of SANCTUARY system via cross-core L1 cache manipulation during concurrent execution, steal IEE data of Ginseng system through bypassing the security measures enforced during the "switch out" process, and steal and tamper with the IEE data of TrustICE system by manipulating the non-secure cache of IEE memory which is not well protected during the IEE's context switching processes, respectively. Our case studies show the wide existence of CITM attacks in IEE systems and point out the importance of securely protecting the cache in addition to the main memory for ensuring the security of IEE data.

Finally, we propose a countermeasure to resolve the attacks, whose root causes are the incoherence of security-related attributes between cache and memory and desynchronized read and write operations between cache and memory. The main idea of our solution is to securely configure the cache attributes for the IEE memory and/or clean the cache mapping to the IEE memory during context switching. We develop a prototype of the defense system on the i.MX6Quad Sabre development board and the experimental results show that our countermeasure has a small system overhead on the rich OS and security-sensitive applications.

In summary, we make the following contributions.

- We conduct a systematic study of cache attributes and their security implications on IEE systems, and discover new cache-based attacks called CITM.
- We perform case studies of CITM attacks against three recently proposed IEE systems. Our attack prototypes show that the CITM attacks may be misused to steal and tamper with the sensitive data in IEE systems, which raises the importance of considering memory and cache together when designing IEE systems.
- We present a countermeasure to mitigate the CITM attacks after identifying the root causes. The prototype shows it can effectively remove CITM attacks from IEE systems with small system overhead.

2 BACKGROUND

We first introduce the ARM TrustZone hardware security extension and then discuss the cache architecture on the ARM processors. We also provide a brief description on three IEE systems including SANCTUARY [13], Ginseng [55], and TrustICE [54].

2.1 ARM TrustZone

TrustZone is a security extension since ARMv6 architecture to provide a hardware-based isolation environment for secure code

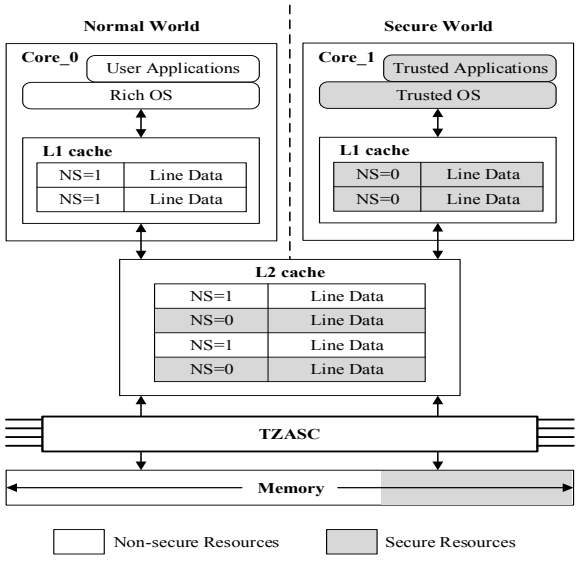

Figure 2: Architecture of ARM TrustZone

execution. The security is achieved by partitioning the resources including processors, memory, and peripherals into one of two worlds, i.e., the secure world and the normal world. As illustrated in Figure 2, normal user applications run on a rich OS in the normal world, and the secure world is preserved for running a small number of trusted applications and a trusted OS.

TrustZone enables the separation of the physical DRAM (main memory) into two partitions, namely, *secure memory* and *non-secure memory*. The non-secure memory is accessible to both normal world and secure world, while the secure memory can only be accessed by the secure world. The memory separation is achieved through a hardware peripheral called *TrustZone Address Space Controller (TZASC)*, which can split the entire memory address space into several memory regions. TZASC allows each region to specify its own security attribute as either secure or non-secure. The latest TZASC model TZC-400 [5] introduces a new security-control feature called *Identity-based Filtering* to separate the non-secure memory regions. In the ARM system, each device (e.g., CPU, GPU, DMA controller etc.) acts as a bus master and is assigned with a unique Non-Secure Access IDentifier (NSAID). TZC-400 can configure one non-secure memory region as only accessible to specific devices based on their NSAIDs, and block other devices' accesses to this region.

2.2 ARM Cache Architecture

Cache is a component on the processors used for buffering the memory's data. Most modern ARM processors [6] are equipped with two levels of caches, i.e., level-one (L1) cache and level-two (L2) cache. The L1 cache is further categorized into instruction cache (I-cache) and data cache (D-cache). The L2 cache is unified and holds both instructions and data. Both L1 and L2 caches are organized as *N-way Set Associative Cache*. Specifically, the entire cache space is divided into N equally-sized pieces, called ways. Further, each way is indexed with k cache lines, and each line serves as the unit of data saved in the cache. Meanwhile, the cache lines from all ways with the same index compose one cache set. To map the memory into cache, the memory is divided into blocks, and each block has the same size as a cache line. The memory's block i can be loaded into any cache line belonging to the cache set $i \bmod k$. The cache lines are mostly Physically Indexed, Physically Tagged (PIPT).

The cache hierarchy is divided into two domains, i.e., *inner cacheability domain* and *outer cacheability domain*. The *inner cacheability domain* means the cache equipped inside a specific CPU core (e.g., the L1 cache which is usually exclusively owned by one core), and the *outer cacheability domain* means the cache equipped outside the CPU cores (e.g., the L2 cache which is usually shared among the cores). Each cacheability domain can be configured using four attributes, i.e., *non-cacheable*, *write-back*, *write-through*, and *write-allocate*, which define the caching behavior of the memory accessing operations. And the attributes are configured at the granularity of memory page. When a memory page is configured as *non-cacheable* for a cacheability domain, any reading and writing operation on the memory page will not go through that cacheability domain. Correspondingly, the memory page is *cacheable* for the cacheability domain when it is configured as *write-through* or *write-back*. The *write-through* attribute additionally means to forward any writing on the current level cache immediately to the next level storage. For example, writing on the L1 cache will be forwarded to L2 cache, and L2 cache will be forwarded to main memory. In contrast, the *write-back* attribute means the changes are only buffered in the current level cache, and the next level storage can only be updated when the cache eviction happens. When a cache miss happens for a write transaction, if the cacheability domain is set with the *write-allocate* attribute, a new cache line will be allocated to save the write result. Otherwise, the cache-missed write will make modifications to the next level storage. In the following, when both the *inner cacheability domain* and the *outer cacheability domain* are set with the same attributes, we omit the *cacheability domain* attribute for brevity. For example, we use *write-back, write-allocate* to represent *inner write-back write-allocate, outer write-back write-allocate*.

Besides the settings of caching attributes, ARM cache's status are also affected by other maintenance operations, such as *invalidation* and *cleaning* instructions. When the *invalidation* instruction is executed, it directly invalids the data saved in the cache. And the *cleaning* instruction forwards the contents of the target cache to the next level cache or main memory. For ARM processors with TrustZone support, caches in all levels are extended with an additional tag bit (i.e., the NS-bit in Figure 2) to record their security state. When accessing memory from normal world, the corresponding cache lines will be set as non-secure; when accessing memory from secure world, the corresponding cache lines will be set as secure. In addition, the cache line's NS-bit is set automatically by the hardware and cannot be modified by the software.

2.3 IEE Systems

The IEE system aims to construct an isolated execution environment (i.e., IEE) in the normal world through a trusted IEE monitor in the secure world, as illustrated in Figure 1. The security-sensitive application running inside the IEE could be a code snippet, a function, an application, or a system, whose sensitive data is well protected against the untrusted rich OS. However, existing IEE systems focus

more on protecting the main memory that is commonly utilized to accommodate the sensitive IEE data, while the security of cache in IEE systems has not been well studied. In the following, we provide a brief description on three recently proposed IEE systems, i.e., SANCTUARY [13], Ginseng [55], and TrustICE [54].

SANCTUARY supports to run sensitive apps and a micro kernel in an IEE concurrently with the rich OS on multi-core platforms. Each IEE is allocated to run on a dedicated core with core-isolated memory, and the execution of the IEE could not be interrupted by other non-secure cores. To prevent the cache-based attacks, the micro kernel will clean the L1 cache during the IEE context switch processes (i.e., before an IEE is terminated or before the sensitive apps are loaded). Also, the L2 cache is disabled for the core running the IEE. Since the L1 cache locates inside each core and cannot be directly accessed from the other cores, SANCTUARY provides no extra protection on L1 cache during the run time of an IEE. Ginseng is an IEE system that protects sensitive data of selected functions on multi-core platforms. To defend against the concurrently running malicious OS, the sensitive data is stored and processed only in the registers, which is inaccessible from the other cores. Since the sensitive data are only stored in registers instead of core-isolated memory, Ginseng provides no protection on the cache. TrustICE is an IEE system implemented on single core platforms, where the IEE and untrusted rich OS could not run concurrently. The security-sensitive application running in one IEE consists of a user program and a micro kernel. Memory is leveraged to store and process the sensitive data, and the sensitive data protection is achieved by dynamically configuring the security attribute of IEE memory, i.e., memory allocated for an IEE will be set as non-secure when the IEE is running and as secure otherwise. TrustICE also lacks of protection on the cache.

3 THREAT MODEL

In this paper, we focus on investigating the attacks against the sensitive data in Isolated Execution Environments (IEEs), as illustrated in Figure 1(b). We assume the rich OS in the normal world cannot be trusted, and the attacker with the root privilege aims to break the confidentiality and integrity of the sensitive data in the IEE, i.e., stealing and tampering with the sensitive data in the IEE. We assume the ARM TrustZone technique can be trusted to provide secure isolation between the normal world and the secure world. The software running inside the secure world (e.g., the IEE monitor) can be trusted and cannot be compromised by the rich OS. We assume the security-sensitive application running in the IEE will not deliberately disclose its sensitive data to the outside, and its code is well protected by the IEE monitor.

4 CITM ATTACKS

We first abstract two generic data protection models in IEE systems. Next, we uncover three types of cache-based CITM attacks that can compromise the security of the IEE data against these two data protection models. Also, we introduce the cache lockdown technique which is frequently leveraged in the CITM attacks.

4.1 IEE Data Protection

Two generic data protection models have been adopted to protect the IEE data in two scenarios, namely, (i) allowing untrusted procedures to run concurrently with one security-sensitive application in the normal world and (ii) suspending all untrusted procedures when a security-sensitive application is running in the normal world.

Model 1: Untrusted procedures are allowed to run concurrently with a security-sensitive application on two (or more) different cores in the normal world. On multi-core platforms, when the security-sensitive application is running on one core in the normal world, untrusted procedures (e.g., the untrusted rich OS) may run concurrently on different cores [13] and/or run on the same core in a time-sharing manner [55]. Such IEE systems usually have three security measures in place to protect its sensitive data. First, the core-isolated storage (e.g., memory allocated for one core and inaccessible to the other cores [13], or the on-core storage like registers [55]) is allocated for each security-sensitive application to process its sensitive data during concurrent execution. Second, when the execution of a security-sensitive application is suspended or finished, all its sensitive data is protected against untrusted procedures by cleaning the core-isolated storage [13] during the "switch out" process. Third, when the execution of a security-sensitive application is resumed or started, the IEE monitor is responsible for restoring the core-isolated storage [55] or allocating blank core-isolated storage [13] during the "switch in" process.

Model 2: Untrusted procedures are *NOT allowed to run concurrently with security-sensitive applications in the normal world.* On single-core platforms, when one security-sensitive application is running in the normal world, all untrusted procedures are always suspended [54]. On multi-core platforms, all untrusted procedures are suspended by the IEE monitor even if there are cores available [22]. Since all cores can only run either the security-sensitive application or the untrusted procedure at any time in the normal world, there is no need to allocate core-isolated storage. Meanwhile, it still requires to enforce the security measures that should be performed during the IEE's context switching processes. Besides the security measures introduced in model 1, when concurrently running is not allowed, the protection could also be achieved through configuring the IEE memory as inaccessible to the normal world [54] during the "switch out" process, and restoring it as accessible to the normal world [54] during the "switch in" process.

4.2 CITM Attack Types

Existing IEE systems focus more on protecting the memory, but the security of cache in IEE systems has not been well studied. After conducting a comprehensive investigation of the cache features on the ARM platforms, we find both data protection models enforced in the IEE systems might be compromised via manipulating the cache in the normal world by the untrusted rich OS. From the attacker's point of view, the two IEE data protection models could be defeated from two main directions, i.e., *manipulating the core-isolated memory during concurrent execution* and *tampering with the context switching of the IEE*. In the following, we introduce three types of CITM attacks identified on the IEE systems.

Type I. *Manipulating core-isolated memory during concurrent execution.*

In multi-core systems, when main memory is used as the core-isolated storage for the security-sensitive application, the concurrently running malicious OS may steal or modify the IEE data in the core-isolated memory by manipulating the cache in the normal world. When the core-isolated memory is set as *cacheable*, the IEE data in the memory will pass through the cache when the memory is accessed. Since the memory is accessed by the security-sensitive application in the normal world, the corresponding cache lines are tagged as non-secure (see Section 2.2) and may be manipulated by the untrusted rich OS. For example, by crafting a page table entry with the physical address pointing to a core-isolated memory page, the malicious OS could manipulate its cache lines through accessing the corresponding virtual address. As an example, we illustrate how the IEE data in SANCTUARY [13] might be compromised by the concurrently running malicious OS through manipulating cache in Section 5.1. Note modern ARM platforms provide hardware features to ensure memory isolation, but they lack similar features to ensure cache isolation. For example, the *Identity-based Filtering* feature on the TZC-400 (see Section 2.1) can be used to isolate memory, but it does not guarantee the isolation of cache.

Type II. *Bypassing security measures during IEE "switch out" process.*

The attackers may aim to bypass the security measures that are enforced to ensure data protection during IEE's context switching processes. Since the rich OS cannot be trusted, the security measures during the "switch in" process are always initiated and accomplished by the trusted IEE monitor in the secure world. Therefore, since the associated cache lines are tagged as secure and could not be manipulated in the normal world, it is difficult to bypass those security measures by manipulating the cache lines. For security reason, the security measures enforced during "switch out" process should be performed either by the IEE itself in the normal world or the IEE monitor in the secure world. However, we uncover the security measures could be bypassed in both cases.

When the security measures are performed by the IEE in the normal world, data cleaning could be achieved through overwriting the IEE memory with random data or all zero data if memory is leveraged to store IEE data. Since the memory cleaning is performed in the normal world, the corresponding cache lines are non-secure. Thus, it is possible to constrain the memory writings in the cache and retain the memory unchanged by controlling the non-secure cache. For example, when the memory is set as *write-back, write-allocate*, all memory writing will be buffered in the corresponding cache set until the cache set is evicted (see Section 2.2). The attackers can leverage the cache lockdown technique (see Section 4.3) to prevent the cache eviction, so that the sensitive data on IEE memory might not be securely cleaned after the security-sensitive application has been suspended or finished.

When the security measures are performed by the IEE monitor in the secure world, the IEE should be able to directly transfer the control to the IEE monitor in the secure world without involving the rich OS. Otherwise, the attacker may manipulate the cross-world context switching to bypass the security measures. However, the context switching from the normal world to the secure world is normally initiated by invoking a high-privileged instruction called *Secure Monitor Call (SMC)* in the rich OS kernel. When the security-sensitive application is a function or an application [55], the *SMC* instruction cannot be directly invoked inside the IEE. To solve this problem, some IEE systems choose to trigger the context switching by raising an external abort in the secure world via intentionally accessing secure memory from the security-sensitive application. However, since the cache lines mapping to the secure memory are non-secure when being accessed in the normal world, the malicious rich OS may manipulate the corresponding cache lines to bypass the control switching and the security measures (e.g., IEE data cleaning). We show how the security measures during the "switch out" process in the Ginseng system [55] can be bypassed via controlling the non-secure cache in Section 5.2.

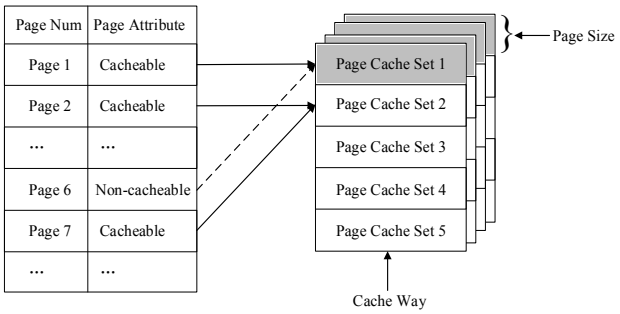

Figure 3: Locking Specified Cache Set via Page Table Control

Type III. *Misusing incomplete security measures during IEE's context switching.*

In some IEE systems, the memory protection during context switching is achieved by dynamically controlling the security attribute of the IEE memory, i.e., configuring it as non-secure during "switch in" and secure during "switch out". In those systems, even if the security measures during context switching are securely enforced, the cache may still be misused by attackers to manipulate the IEE data. This is because the memory configuration is achieved through TZASC (see Section 2.1), but the corresponding cache lines might be still non-secure in the normal world. Therefore, inappropriate cache cleaning during "switch out" might lead to IEE data leakage. Similar, malicious data might be loaded into the cache before "switch in" and later be fed to the security-sensitive application rather than the original sensitive data in the IEE memory. Note Type III attack is due to the incomplete security measures on cache, which is different from Type II attack that focuses on bypassing the existing security measures during context switching. We illustrate how TrustICE [54] system becomes vulnerable by manipulating cache during the "switch in" and "switch out" processes.

In summary, when the IEE systems use Model 1 for data protection, they may suffer from all three identified attacks. When using Model 2, the IEE systems are vulnerable to Type II and III attacks, but not Type I attack since the concurrently running is not allowed. Also, Type I and III attacks work only when memory is employed to store IEE data, and Type II attack has this requirement only when the security measures for "switch out" process are performed in IEEs.

4.3 Cache Lockdown Technique

Cache lockdown is a feature that enables a program to load code and data into cache and mark it as exempt from eviction [51]. The main purpose of locking the code or data in cache is to provide faster system response and avoid the unpredictable execution times due to the cache line eviction. Attackers may misuse this technique to launch the CITM attacks, e.g., locking the writing operation for memory in the cache to invalidate the memory-cleaning operations.

Three approaches may be adopted to achieve cache lockdown. First, some ARM development boards (e.g., i.MX53 and i.MX6Quad development boards) allow the users to lock certain L2 cache ways by configuring the *L2 auxiliary cache control register* [58]. However, this hardware-based locking control register is not supported on ARMv8 processors. Second, by setting the memory regions controlled by the attackers as *outer cacheable* and all other memory regions as *outer non-cacheable*, attackers can exclusively occupy the L2 cache. This method may introduce huge performance overhead on the normal execution of the system due to the exclusive usage of L2 cache. Third, attackers may exclusively occupy some L2 cache sets by conducting a fine-grained control on each memory page's caching attributes. Specifically, since the caching attributes are set at the granularity of memory page, we divide each cache way into page-sized blocks. Blocks with the same index compose a page cache set (see the cache lines marked gray in Figure 3). To lock the data of a specified memory page on the L2 cache, attackers can configure that memory page as *outer cacheable*, and set all other memory pages sharing the same page cache as *outer non-cacheable*. For example, in Figure 3, when page 1 and page 6 share the same page cache set, attackers can lock the data of page 1 on the L2 cache by setting page 1 as *outer cacheable* and page 6 as *outer non-cacheable*. This technique has been leveraged in the *SecTEE* [60] system to prevent cache-based side-channel attacks. Considering the small size of L1 cache, we choose to lock only L2 cache using the third method in our attacks.

5 CASE STUDY OF CITM ATTACKS

We conduct case studies of the CITM attacks on three well-known IEE systems including SANCTUARY [13], Ginseng [55], and TrustICE [54] to illustrate how they could be utilized to compromise IEE systems on real hardware testbeds. Since SANCTUARY system is achieved through the ARM Fast Models virtualization tools rather than on the actual development board, we simulate the cache operations of SANCTUARY on the i.MX6Quad development board, and successfully steal and modify the sensitive IEE data via the cross-core L1 cache manipulation. The cache operations of SANCTUARY are mainly obtained through carefully studying the published paper [13]. The CITM attacks of the Ginseng and TrustICE systems are implemented with the source codes shared from their authors on two real hardware development boards, namely, HiKey620 and i.MX6Quad SABRE.

5.1 SANCTUARY: Manipulating L1 Cache

In the following, we first introduce the security measures of SANCTUARY on IEE data protection. Next, we elaborate that SANCTUARY suffers from Type I Attack due to the lack of protection on the L1 data cache at the run time of an IEE. Then, we detail the attacking procedure that can leak and tamper with sensitive data in IEE of the SANCTUARY system.

5.1.1 Data Protection Mechanisms. SANCTUARY includes a number of data protection mechanisms to protect the sensitive data in IEE. When the IEE finishes its running (i.e., during the "switch out" process), the sensitive data will be cleaned by the micro kernel running inside the IEE, which overwrites all-zero data to the protected core-isolated memory and invalidate the L1 cache (L2 cache is disabled for the core-isolated memory in SANCTUARY). Before booting up one IEE (i.e., during the "switch in" process), the IEE monitor in the secure world constructs a clean environment for the IEE. In addition, the micro kernel invalidates the L1 cache before the sensitive apps are loaded. Therefore, SANCTUARY is immune to Type III attack, since the core-isolated memory and L1 cache are safely cleaned by the micro kernel during both "switch in" and "switch out" processes and L2 cache is disabled. Also, SANCTUARY is immune to Type II attack, since the data cleaning operation is accomplished by the micro kernel running inside IEE. To bypass the data cleaning operation, the attackers should lock the memory overwriting on the L1 cache (i.e., preventing the data on L1 cache from being evicted to memory). However, the eviction actions of the L1 cache are determined by the memory operations on that core and the core-isolated memory pages' cache attributes. The attackers cannot control the eviction by manipulating another core. In addition, the cache attributes of the core-isolated memory are controlled and protected by the micro kernel running in the IEE.

SANCTUARY defends against the concurrently running malicious OS by protecting both memory and L2 cache. It allocates core-isolated memory for each IEE. The memory isolation is achieved through the *Identity-based Filtering* feature (see Section 2.1). Particularly, it assigns each core with a unique Non-Secure Access IDentifier (NSAID), and allocates isolated memory regions for each NSAID by configuring TZC-400. Since existing ARM platforms do not support to assign the NSAID at the granularity of CPU core, i.e., all the CPU cores share the same NSAID, the memory isolation is achieved through the ARM Fast Models virtualization tools rather than on the actual development board. Protection of the L2 cache is achieved by configuring the protected memory region as *outer non-cacheable*. Thus, the sensitive data do not pass through the L2 cache, which is usually shared among all cores on the ARM mobile devices. Since the L1 cache locates inside each core and cannot be directly accessed from the other cores, SANCTUARY provides no extra protection on L1 cache during concurrent execution. However, we identify one Type I attack in SANCTUARY by manipulating the L1 data cache on a different core.

5.1.2 Type I Attack in SANCTUARY. After investigating the cache features on the ARM platform, we discover a cache attribute named *shareability* that can be configured to read/write one core's L1 data cache via operating another core's L1 data cache [4] [1]. The shareability attribute defines the range of the cache to be ensured of value coherency. There are two types of shareability domain, i.e., the *inner shareability domain* and the *outer shareability domain*. The former ensures the value coherence among the cores inside one cluster, and the later ensures the value coherence among the cores

[1] The shareability attribute works only for L1 data cache, but not L1 instruction cache.

in all clusters. When the ARM platforms have more than one group of cores, each group represents a *cluster*. For instance, the Juno r2 development board is equipped with two clusters, one cluster with a quad-core Cortex-A53 processor and another cluster with a dual-core Cortex-A72 processor [8].

Since value coherency is naturally ensured on the single-core platforms, the *shareability* attribute is only configurable on the multi-core platforms [7]. It only works when the processors run in the *Symmetric Multi-Processing (SMP)* mode [7], which is set by default on most multi-core platforms. When the shareability attribute is set, the value coherency can be ensured by the *Snoop Control Unit (SCU)*, which contains buffers that handle direct cache-to-cache transfers between cores [6]. If the value on one core's L1 data cache is modified, SCU synchronizes the changes to the L1 data cache of other same-cluster cores if the corresponding memory page is set as *inner shareable*. The changes will be synchronized to all other cores if the memory page is set as *outer shareable* [7]. For security reason, the data on non-secure cache will not be synchronized to secure cache, and vice versa.

We conduct a series of experiments to better understand the impacts of the shareability attribute on the non-secure L1 data cache [4] (see Appendix A). The results show that the data on one core's L1 data cache could be leaked out to and tampered by another core, when both cores run in the normal world (it ensures the corresponding cache lines to be non-secure) and access the same physical memory address with that memory page's cache attribute set as *inner shareable* or *outer shareable* for both cores. The cache attribute is configured separately for each core since it is configured in the page tables, and each core has its own set of page tables. In contrast, data on one core's L1 data cache could not be leaked out to and tampered by another core, when the corresponding memory is set as *non-shareable* (i.e., *inner&outer non-shareable*) for that core.

5.1.3 Attacking Procedure. We simulate the cache operations of Sanctuary on the i.MX6Quad SABRE development board. As illustrated in Figure 4, core_0 and core_1 are running in the normal world, and core_2 is running in the secure world. The security-sensitive application is running on core_0, and the untrusted rich OS is running on core_1. The *Static Trusted App* (i.e., the IEE monitor in SANCTUARY) runs on core_2, and it configures the TZC-400 to allocate an isolated memory region for core_0 and disable the usage of L2 cache for core_0. Though the memory region assigned to core_0 is non-secure memory, it is protected by the *Static Trusted App* to block the access from core_1. The L1 cache lines accessed by core_0 are non-secure cache, since they are accessed in the normal world. According to our experimental results listed in Appendix A, it is possible to affect one core's L1 data cache by manipulating another core's L1 data cache, when both caches are non-secure. Therefore, the sensitive data in core_0's L1 data cache might be stolen or modified through controlling core_1's L1 data cache.

In Linux kernel, all the cacheable memory is by default set as *shareable* (*inner* or *outer shareable*). According to the description in the paper [13], the SANCTUARY system prevents the cache based attacks by invalidating the L1 cache during the IEE context switching processes and changing the *cacheability* attribute of the core-isolated IEE memory to be *inner cacheable, outer non-cacheable* during the runtime of an IEE. Since the *shareability* attribute is not

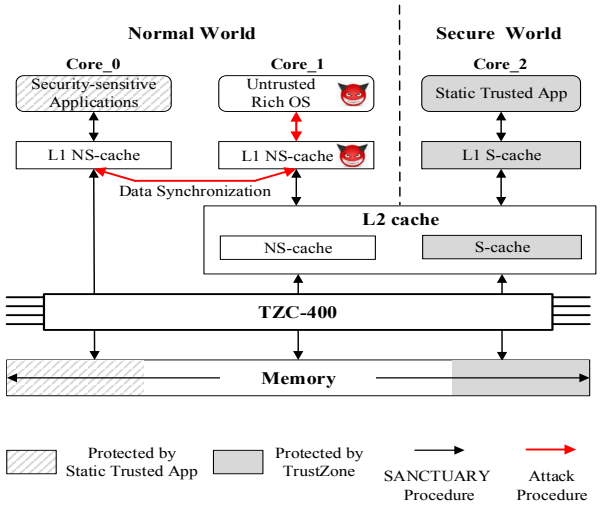

Figure 4: CITM Attack on SANCTUARY

configured, the protected memory will be by default set as *shareable*. Therefore, we can read and write the sensitive data residing in the L1 data cache during the concurrent execution (i.e., when an IEE is running currently with the untrusted rich OS) by leveraging the *shareability* attribute. As illustrated by the red lines in Figure 4, we first craft a page table entry for core_1, configuring the memory page's cache attributes as *shareable* and making its physical address point to a memory page of core_0 (i.e., a memory page protected by SANCTUARY). Then, when we access the corresponding virtual address on core_1, sensitive data in core_0's L1 data cache could be stolen or modified due to the value coherency ensured by the *shareability* attribute.

In the above attacking procedure, we need to identify physical addresses of IEE memory (i.e., memory allocated for core_1). Since the page tables associated with the IEE memory is maintained in the IEE, the attackers cannot directly obtain its physical address range. However, the entire physical memory is divided into three parts, i.e., IEE memory, TEE memory (memory allocated for the secure world), and unprotected memory (memory allocated for the untrusted rich OS). The address range of the unprotected memory is naturally known to the malicious rich OS. The remaining two memory regions can be distinguished since the cache corresponding to IEE memory is non-secure while the cache of TEE memory is secure. Though reading the TEE memory will always return zero or generate an exception (depending on the configuration of TZASC), reading of IEE memory can obtain real data when the memory data is buffered in the cache. Therefore, we can identify the IEE memory through probing the memory region apart from the unprotected memory. Since the size of cache is normally smaller than IEE memory, we may need to probe several times to identify the entire IEE address range. The probing times depend on the size of IEE and TEE memory, the time durance of IEE data in L1 data cache, etc.

Note since the cache shareability attribute is not well-known and never mentioned in SANCTUARY paper [13], when other developers follow the paper for reimplementation, there is a high probability that their systems are prone to the same problem.

5.2 Ginseng: Mapping to Non-Secure Cache

As described in Section 2.3, Ginseng is an IEE system that protects the sensitive data by storing and processing them in registers. As such, Ginseng is immune to the Type I attack. Also, since the contents in registers do not pass through cache, it is immune to the Type III attack too. However, we discover that Ginseng suffers from the Type II attack. Ginseng relies on the TEE monitor running in the secure world to perform the data cleaning operations during the "switch out" process. Since the rich OS cannot be trusted, the control flow is transferred directly from the IEE to the TEE monitor by accessing the secure memory in the IEE in order to trigger a secure interrupt. However, by manipulating the non-secure cache (that maps to the secure memory) in the normal world, the attackers can block the control flow switching from the IEE to the IEE monitor and thus bypass the data cleaning operations.

```
1  /*sensitive function*/
2  int genCode(sensitive long key_top,
3              sensitive long key_bottom) {
4      // operations for insensitive data
5      ...
6      // invoke sensitive function
7      hmac_sha1(key_top, //sensitive data
8          key_bottom,    //sensitive data
9          challenge,     //insensitive data
10         resultFull);   //insensitive data
11     // truncate 20-byte hmac_sha1() result to 4-byte
       ↪ truncatedHash
12     ...
13     // invoke insensitive function
14     printf("OTP: %06d\n", truncatedHash);
15     return truncatedHash;
16 }
17
18 /*sensitive function*/
19 void run(){
20     // mark the protected data as sensitive
21     sensitive long key_top, key_bottom;
22     // read keys from TEE secure world
23     ss_read(UUID1, UUID2, key_top);
24     ss_read(UUID3, UUID4, key_bottom);
25     // invoke sensitive function
26     genCode(key_top, key_bottom);
27 }
```

Listing 1: A Sample Program Protected by Ginseng

5.2.1 Data Protection Mechanisms. We illustrate the working flow of Ginseng system through a sample program shown in Listing 1. The program is to perform a `hmac_sha1` operation based on two keys obtained from the secure world. Specifically, two local variables are marked to be protected as *sensitive*, i.e., `key_top` and `key_bottom` (line 21). Ginseng provides a compiler to perform static taint analysis for identifying all variables that may carry sensitive data and allocating them in the registers. The functions involving sensitive data are identified as sensitive functions, and will undergo code integrity check before being executed. In this example, the functions `run()` (line 19), `genCode()` (lines 2 and 26), and `hmac_sha1` (line 7) are sensitive functions. Function `printf()` (line 14) is an insensitive function, whose code integrity is not guaranteed. All sensitive data associated operations are executed in the registers instead of in the memory.

To defend against malicious kernel, Ginseng introduces six secure API functions to transfer the control flow directly from user space of the normal world to GService (i.e., the IEE monitor in Ginseng). Two of the secure API functions are provided for the programs to securely interact with GService i.e., `ss_write()` and `ss_read()`. Another four secure API functions will be inserted to the program automatically by the compiler. The `ss_saveCleanV()` and `ss_readV()` are two secure API functions inserted before and after each function invocation inside the sensitive functions, where the former is responsible for encrypting the sensitive data, storing the encrypted data in memory, and cleaning the corresponding registers, and the later takes charge of decrypting the sensitive data and restoring them into registers. For example, the `ss_saveCleanV()` and `ss_readV()` will be inserted before and after the `printf()` function (line 14) inside `genCode()`. Another two secure API functions `ss_start()` and `ss_exit()` are inserted at the begin and end of each sensitive function to conduct preparation work (e.g., performing code integrity check for the sensitive function) and clean all sensitive registers to prevent data leakage, respectively.

Before "switching out" of the IEEs, Ginseng achieves a cross-world context switching directly from the IEEs that run in the user space of the normal world to the GService that runs in the secure world, and performs the security-sensitive operations (e.g., encrypting the sensitive data, cleaning the sensitive registers etc.) in the GService. The normal way to trigger the cross-world context switching is invoking the high privileged *SMC* instruction from kernel space. However, the approach is not applicable in Ginseng, since its IEEs run in the user space and are not able to invoke the *SMC* instruction. Ginseng resolves the problem by triggering a security violation in the IEEs. Specifically, each secure API function is assigned a unique secure memory by configuring TZASC. Then, the invocation of a secure API function will trigger a security violation since it attempts to access secure memory from the normal world. By default, the processor raises an external abort (EA) in the normal world when handling the violation. To raise the EA in the secure world, GService sets the `external abort` bit of the Secure Configuration Register, so that GService can obtain the EA and handle the requests sent by secure APIs without the attendance of the malicious kernel.

5.2.2 Type II Attack in Ginseng. However, this triggering solution could be manipulated since the cache lines corresponding to the secure memory are non-secure. We use the secure API `ss_saveCleanV()` as an example to illustrate the problem (lines 11 to 19 in Listing 2 are the implementation of `ss_saveCleanV()` in Ginseng). It first loads the address of `__channel_save_clean` (the secure memory assigned to `ss_saveCleanV()`) into register `x4` (line 16). Then it loads the data from the secure memory located at `x4` to the register `x0` (line 18). Figure 5 illustrates the detailed execution flow. ① When the "secure memory loading" instruction (i.e., line 18) is executed, the processor tries to load data from cache. ② The cache fetches data from secure memory due to cache miss. ③ An external abort is raised since accessing secure memory from normal

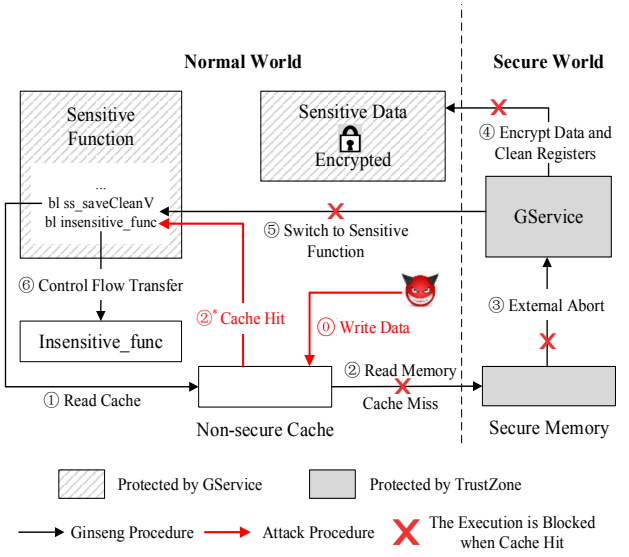

Figure 5: CITM Attack on Ginseng

```
1  /* Attack preparation: fill in the corresponding cache in
   ↪  advance*/
2  writeSM:
3      /* __channel_save_clean:
4      virtual address of the secure memory
5      assigned to ss_saveCleanV*/
6      ldr    x4, =__channel_save_clean
7      /* store data to secure memory */
8      str    x0, [x4]
9      ret
10
11 /*Implementation of the secure API ss_saveCleanV in Ginseng*/
12 ss_saveCleanV:
13     /* __channel_save_clean:
14     virtual address of the secure memory
15     assigned to ss_saveCleanV*/
16     ldr    x4, =__channel_save_clean
17     /* load data from secure memory */
18     ldr    x0, [x4]
19     ret
```

Listing 2: Exploiting the Cache of Secure Memory

world, and is captured by GService. ④ GService encrypts the sensitive data and cleans the corresponding registers. ⑤ The control flow is transferred back to the secure API `ss_saveCleanV()`. ⑥ The insensitive function `insensitive_func()` is invoked. Though the secure memory is protected against malicious kernel, the cache accessed in step ① is non-secure cache, since it is accessed from the normal world. Therefore, it could be manipulated by the attackers.

5.2.3 Attacking Procedure. Based on Ginseng's open source code [24], we implement a prototype of Ginseng on a HiKey620 development board with a 8-core ARM Cortex-A53 processor. Our attack can successfully steal the sensitive data through the attacking procedure illustrated by the red lines in Figure 5. Particularly, we introduce a step ⓪, which fills in the cache lines mapping to the secure memory before step ① is executed. Then, when the processor tries to load data from cache through the "secure memory loading" instruction (i.e., step ①), it encounters a cache hit rather than cache miss. As such, the step ②* will be executed while the normal execution of steps ②, ③, ④ and ⑤ are blocked. Finally, in step ⑥, the insensitive function `insensitive_func()` could be manipulated to read the uncleaned sensitive registers, since it is executed on the same core as the sensitive function and is not protected through code integrity check. For example, we can change the control flow of `printf()` (an insensitive function invoked by the sensitive function `genCode()` in Listing 1 through modifying the `libc.so` library. Since the clean process is interrupted unnoticeably, the function `insensitive_func()` can read the sensitive data (e.g., keys) from the registers.

The `writeSM` function in Listing 2 is used to accomplish step ⓪, i.e., writing data to the cache of `__channel_save_clean` (the secure memory assigned to `ss_saveCleanV()`). First, the virtual address of `__channel_save_clean` (line 6) is loaded into register `x4`. Since the protected program's (e.g. the `hmac_sha1` program in Listing 1) page tables including the virtual address of `__channel_save_clean` are maintained in rich OS kernel, it could be obtained by the attackers. For security reason, Ginseng hooks all page table update operations and ensures the page tables are read-only to rich OS kernel. Then, data of register `x0` is stored to the secure memory located at `x4` (line 8). The data will be first written to the cache of the secure memory `__channel_save_clean`. Since the secure memory is set as *write-back, write-allocate* in the Ginseng system, the data is buffered in cache and will only be evicted to memory when the cache set is full. We leverage the cache lockdown technique introduced in Section 4.3 to prevent the cache eviction, i.e., we set the memory pages sharing the same page cache set with the secure memory `__channel_save_clean` as *outer non-cacheable*. Since some data is locked in advance in the cache lines mapping to the secure memory `__channel_save_clean`, memory access in step ① will encounter a cache hit and avoid the data fetching from secure memory (i.e., step ②). Then, no security violation will be triggered, neither does the cross-world context switch. As such, by invoking the `writeSM` function before `ss_saveCleanV()` is executed, we can successfully bypass the context switching from IEEs to the GService and the data protection conducted in GService. The secure API function `ss_exit()` for cleaning the sensitive registers at the end of each sensitive function could be attacked similarly.

5.3 TrustICE: Incomplete Cache Cleaning

In the following, we first illustrate data protection mechanisms in TrustICE. Next, we point out that TrustICE suffers from Type III attack due to incomplete cache cleaning during the context switching processes. Then, we detail the attacking procedure.

5.3.1 Data Protection Mechanisms. TrustICE statically divides the entire physical memory into three separated regions for the rich OS in the normal world, the IEEs in the normal world, and a Trusted Domain Controller (i.e., the IEE monitor in TrustICE) in the secure world, respectively. Sensitive data protection is achieved by dynamically configuring the security attribute of IEE memory. The IEE memory is set as secure by the Trusted Domain Controller when

the system boots up. Before launching a new IEE, the Trusted Domain Controller allocates it a memory region from the IEE memory and sets the memory region as non-secure. When the IEE finishes, the micro kernel inside it transfers the control flow directly to the secure world by invoking the *SMC* instruction. The Trusted Domain Controller then configures the corresponding IEE memory region as secure, before transferring the control back to the rich OS.

5.3.2 Type III Attack in TrustICE. Since TrustICE follows Model 2 to achieve data protection, it is immune to Type I attack. The IEE memory is set as secure when the malicious OS is running and set as non-secure when the security-sensitive application is executing. The data protection during the "switch out" process is achieved by dynamically setting the associated memory as secure. It cannot be bypassed, since it is enforced in the secure world and the control flow is transferred directly from micro kernel in IEE to the secure world by invoking the *SMC* instruction. As such, the IEE system is immune to Type II attack. However, although the IEE memory is protected during the context switching processes, the corresponding cache is non-secure and not cleaned correctly. Therefore, the attackers could leverage Type III attack to compromise the security of sensitive data.

5.3.3 Attacking Procedure. We implement a TrustICE prototype on the i.MX6Quad SABRE development board. Since i.MX6Quad is a multi-core platform, we suspend all the other cores when an IEE is running. Particularly, before booting up an IEE, the *Trusted Domain Controller* sends to the other non-secure cores (except the core allocated to the IEE) an inter-core interrupt, which is configured as a secure interrupt. The cores switch to the monitor mode when receiving the secure interrupt, and thus the procedures running on them are suspended.

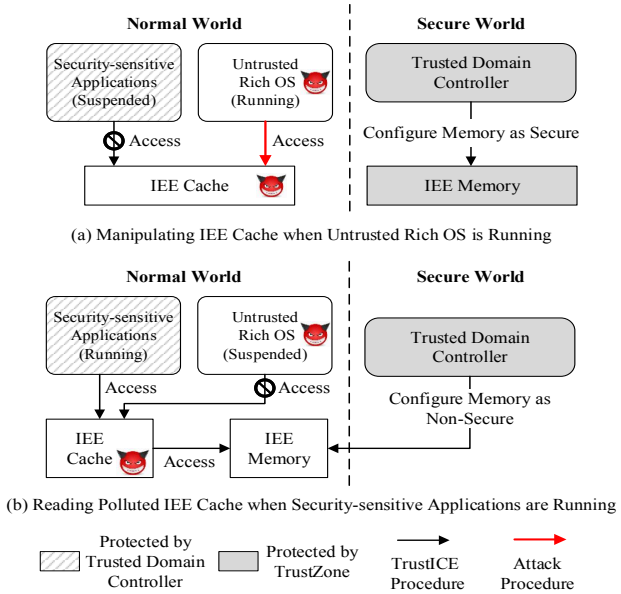

Figure 6: CITM Attack on TrustICE

The attacking procedure is as follows. As depicted in Figure 6(a), we craft a page table entry in the normal world, where its physical address is pointed to an IEE memory page (the address range of IEE memory can be obtained as described in Section 5.1.3) and its cache attributes are set as *write-back, write-allocate*. Although the IEE memory page is configured as secure during the "switch out" process, we can still access the sensitive data residing in the corresponding cache lines that are non-secure, since TrustICE does not clean the data in cache lines. During the "switch in" process, the rich OS can write malicious data to the cache lines corresponding to the IEE memory page and lock the data in cache lines using the cache lockdown technique depicted in Section 4.3. Then, when the IEE is executing, it will first read the malicious data residing in the cache lines rather than the legal data in the IEE memory page, as shown in Figure 6(b).

6 COUNTERMEASURE

Intuitively, the most straightforward defense strategy is completely disabling the cache for all IEE memory. However, it is impractical due to the huge performance overhead without using cache. Based on the analysis of the root causes of CITM attacks, we propose to prevent them by correctly configuring the cache attributes of IEE memory and/or cleaning cache of the IEE memory during context switching. Experiments show that our countermeasure has a small overhead over both rich OS and the security-sensitive applications.

6.1 Defense Approaches

We observe that one main reason for all CITM attacks is the incoherence between two levels of memory architecture, cache and main memory. Thus, our defense focuses on removing those incoherences. First, the memory isolation does not automatically guarantee the cache isolation, and it is the root cause of Type I attack. Particularly, when a memory region is isolated for a dedicated core via the identity-based filtering feature of the TZC-400 in SANCTUARY [13], the data of corresponding L1 cache may still be shared among the cores. Thus, we can eliminate Type I attack by configuring the cache attributes as *outer non-cacheable, non-shareable* (i.e., *inner&outer non-shareable*) for the core-isolated memory.

Second, the main reason for Type II attack is that the reading and writing operations are not synchronized between memory and cache. For instance, the cross-world switching in Ginseng [55] is bypassed by constraining the reading and writing of the secure memory in the cache. After preloading and locking malicious data in the cache corresponding to the secure memory in advance, the future reading of secure memory inside the IEE will hit the preloaded malicious cache. We can defeat Type II attack through synchronizing the reading and writing operations between memory and cache. Specifically, cache attributes of the IEE memory (e.g., the secure memory in the Ginseng system) should always be configured as *write-through, non write-allocate*. As such, the reading and writing operations will not be constrained in the cache.

Third, the memory region is configured as secure or non-secure through TZASC, but the security attribute of a cache line is determined by the status of the core who accesses it. In other words, the cache lines are automatically identified as non-secure if being accessed by a core running in the normal world, and identified as secure if being accessed by a core running in the secure world. This is the main reason for Type III attack. For example, the CITM

attack on TrustICE (see Section 5.3) is achieved by reading the IEE memory's non-secure cache after "switch out", and writing and locking malicious data on the non-secure cache before "switch in". We can defeat Type III attack by cleaning the cache lines during both "switch in" and "switch out" processes, so that attackers could not read residual sensitive data or retain malicious data in the cache.

In summary, the three CITM attacks identified in this paper could be eliminated through (i) configuring the cache attributes of the IEE memory as *inner write-through non write-allocate, outer non-cacheable, non-shareable* and (ii) cleaning the cache of IEE memory during context switching. The approach to enforce the cache attributes varies on different IEE systems. When the IEE memory's page tables are maintained in the IEEs (e.g., SANCTUARY [13]), correct cache attributes could be enforced by the security-sensitive applications running inside the IEEs. When the IEE memory's page tables are constructed in the secure world (e.g., TrustICE [54]), the enforcement of the cache attributes could be achieved by the IEE monitor. When the IEE memory's page tables are maintained in the malicious OS (e.g., Ginseng [55]), all page table update operations of the malicious OS should be interposed and forwarded to the IEE monitor, which then enforces the cache attributes are correctly configured. Particularly, we block the direct manipulation from malicious kernel by setting the IEE memory's page tables and the kernel-privileged codes as read-only to the kernel. Then, we replace all instructions in the kernel codes for updating the page table entries and the related registers, making them trap into the secure world and undergo security checking (i.e., the cache attributes of the critical memory are correctly configured) before being executed. Finally, we configure the system to set the *Privileged Execute Never (PXN)* attribute by default on any newly allocated pages, so that no executable kernel-privileged instructions can be inserted when the system is running. The cache cleaning could be achieved via invoking the *invalidation* and *cleaning* instructions inside the IEE. Particularly, we could directly clean the cache during the "switch in" process by invoking the *invalidation* instruction. To prevent the loss of data during the "switch out" process, we could first invoke the *cleaning* instruction to synchronize data from cache to memory and then invoke the *invalidation* instruction to clean the cache.

6.2 Defense Overhead

We implement a prototype of our countermeasure solution on the i.MX6Quad SABRE development board, which is equipped with a quad-core ARM Cortex-A9 processor running at 1.2GHz with 1GB DDR3 SDRAM. Then, we evaluate the system overhead introduced by our defense. To minimize the noise in the experiments, we run each test with 1,000 iterations and report the average.

We first explore the overhead on security-sensitive applications due to the enforced cache attributes. Particularly, we run an AES encryption application in one IEE, and evaluate its execution time when the memory is configured with different cache attributes. The experimental results show that our defense system (i.e., with the cache attributes set as *non-shareable, inner write-through non write-allocate, outer non-cacheable*) introduces around 90% overhead comparing to the default setting (i.e., with the cache attributes set as *shareable, inner write-back write-allocate, outer write-back write-allocate*), and it is mainly caused by disabling L2 cache. We also observe that for the IEE systems that disable the L2 cache for protection (e.g., SANCTUARY), our defense system only introduces negligible additional overhead. Also, we evaluate the overhead on the rich OS introduced by the additional cross-domain context switches enforced on each page table updating operation. The results show that 2.65% overall overhead is introduced on the execution of rich OS. In addition, we evaluate the overhead on the operations that involves frequent page table updating, i.e., the system booting and application loading. It shows the overall loading overhead for both kernel and applications is less than 10% in all evaluation scenarios. The evaluation details can be found in Appendix B.

7 DISCUSSION

Besides ARM TrustZone, the technologies such as *Software Guard Extensions (SGX)* [44] and virtualization [9] have also been adopted to construct IEEs for protecting the security-sensitive applications against malicious OS. In this section, we show that SGX is immune to CITM attacks by design, and the virtualization-based IEE systems are more difficult to be attacked.

SGX-based Solutions. In the SGX-based solutions [12, 19, 23, 52], the IEEs (also called *enclave* in SGX) are constructed in the user space of an untrusted OS. When an enclave is setup, a specific memory region named *enclave page cache (EPC)* is allocated for it. The sensitive data is only processed and stored in the EPC pages. The SGX-based IEE solutions are immune to the CITM attacks since the hardware-based security measures enforced on the EPC pages. First, the EPC pages and corresponding cache lines are only accessible when the processor is running in the *enclave* mode, i.e., when an enclave is being executed. Second, each physical EPC page could be allocated to only one enclave, i.e., the EPC pages allocated for any two enclaves are not overlapped [12]. The former prevents the direct manipulation on the cache lines of the EPC pages from malicious OS, and the later deters the indirect attacks through manipulating the EPC pages' cache lines from another crafted enclave.

Virtualization-based Solutions. In the virtualization-based IEE solutions [18, 21, 32, 33, 35, 40, 42, 43, 56], the hypervisor is assumed to be secure and the memory is managed through a two-stage address translation mechanism. The stage-1 translation translates a virtual address (VA) to an intermediate physical address (IPA), and the stage-2 translation translates the IPA further to a real physical address (PA). The stage-2 translation is achieved in the hypervisor, which usually provides well protection on the IEE memory by controlling the IPA-to-PA page table mappings (e.g., ensuring separated physical memory regions are allocated for the IEE and rich OS). Although the attackers can manipulate the VA-to-IPA mappings, they can hardly control the access to the real PAs without compromising the hypervisor. Therefore, it is difficult to launch CITM attacks on the virtualization-based IEE solutions, where the cache lines of the IEE memory are indexed through real PAs.

Though the SGX and virtualization based solutions are more secure against the CITM attacks, they have their own limitations. First, the SGX technique is only available on the Intel platforms, but most mobile devices are equipped with the ARM processors. Second, the virtualization-based solutions rely on a trusted hypervisor in the normal world as the reference monitor, which may also be

compromised [11, 15, 45]. In this paper, we focus on the TrustZone-based IEE systems that rely on a small-sized IEE monitor in the secure world to protect the security-sensitive applications against untrusted OS and hypervisor in the normal world. The attacking target of CITM is the IEEs running in the normal world, while the secure world (e.g., the IEE monitor) is immune to CITM since the associated cache is secure cache, which could not be manipulated in the normal world.

8 RELATED WORK

There is a line of research works using ARM TrustZone extension [16] to protect security-sensitive resources against untrusted OS. In general, they can be divided into two categories, i.e., protecting the sensitive resources directly in the secure world, or protecting the sensitive resources in the normal world through a reference monitor running in the secure world. Traditional *Trusted Execution Environment (TEE)* systems usually follow the first implementation model, i.e., implementing the security-sensitive applications as TAs in the secure world, including *OP-TEE* [47], *Qualcomm's QSEE* [28], *Huawei's Secure OS* [34] and *SAMSUNG's KNOX* [25] etc. Research works such as *TrustShadow* [30], *TrustOTP* [53], *TEEv* [39], *PrOS* [38], *Trusted Language Runtime (TLR)* [49], *CaSE* [58], *Komodo* [26], *SecTEE* [60], *MIPE* [17] etc. also fall into the first category. For example, *Komodo* [26] and *SecTEE* [60] implement a SGX-like system in the secure world. *PrOS* [38] constructs multiple isolated TEEs in the secure world. *TrustOTP* [53] provides trusted one-time password functions in the secure world. *Trusted Language Runtime (TLR)* [49] ensures the confidentiality and integrity of the .NET mobile applications by deploying their code and security-sensitive application components in the secure world. Since cache lines accessed in the secure world are secure, these systems are secure for the CITM attacks. However, they increase the TCB by introducing partial or entire execution codes into secure world.

There are two groups of solutions to protect the sensitive resources in the normal world. The first group is the IEE systems such as SANCTUARY [13], Ginseng [24] and TrustICE [54], which are proposed to protect the third-party security-sensitive applications. As depicted in our paper, they are vulnerable to the CITM attacks. The second group includes the solutions such as *TZ-RKP* [10], *SPROBES* [27] and *SeCReT* [37] etc., which shield only certain specific data/codes rather than third-party applications. For example, *TZ-RKP* [10] and *SPROBES* [27] focus on protecting the critical kernel codes. *SeCReT* [37] constructs a secure cross-domain communication channel with the help of *TZ-RKP* [10]. They achieve the data protection through interposing the page table updating operations and preventing the malicious OS from manipulating the cache and memory associated with the sensitive data/codes. Therefore, they are immune to the CITM attacks.

Besides ARM TrustZone, the virtualization technology has also been widely adopted to protect the security-sensitive resources. Solutions such as *OSP* [21], *PrivateZone* [35], *vTZ* [33] and *TFence* [36] target at the ARM platforms. The former three schemes focus on constructing the IEEs, and *TFence* [36] utilizes the hypervisor to create a secure cross-domain communication channel between applications and the TEE. The virtualization technology is also frequently used to protect the security-sensitive data on the x86 platforms. For example, *Flicker* [43], *TrustVisor* [42], *InkTag* [32] and *Minibox* [40] protect the security-sensitive data in an IEE constructed through hypervisor. *Overshadow* [18] protects the security-sensitive applications by illustrating different memory views for the applications and malicious OS, respectively. *CloudVisor* [56] uses a similar idea to protect virtual machines in the cloud platform. *NICKLE* [48] achieves the real-time integrity protection of kernel codes. The *Intel Software Guard Extensions (SGX)* [44] technology has also been adopted in many solutions to secure the security-sensitive applications [12, 19, 23, 52] on the Intel platforms. As discussed in Section 7, both hardware-assisted virtualization and SGX based solutions are more secure against the CITM attacks.

Researchers also investigate on how the cache may be manipulated in developing various cache-based attacks. For instance, cache-based side-channel attacks have been developed on both ARM platforms [20, 31, 41, 59] and Intel platforms [14, 29, 46, 50]. Also, *CacheKit* [57] can hide malicious codes in the normal world and bypass the detection of both the secure and normal worlds since the values of cache in two worlds can be different even they are mapped to the same physical address. We focus on attacking the IEE systems through the incoherence between cache and main memory.

9 CONCLUSIONS

ARM TrustZone extension has been widely adopted in the IEE systems, which constructs a secure IEE in the normal world against malicious OS. However, existing IEE systems focus more on protecting the memory, while the security of corresponding cache has not been well studied. In the paper, we first summarize the data protection measures enforced in the IEE systems into two generic models. After performing a comprehensive investigation of the cache features on the ARM platforms, we identify three Cache-in-the-Middle (CITM) attacks which might compromise both data protection models enforced in the IEE systems. To illustrate how to exploit them on real hardware testbeds, we conduct three case studies on three well-known IEE systems. After analyzing the primary reason for the CITM attacks (i.e., the incoherence between memory and cache), we propose a defense scheme to defeat them. The experimental results show that a small overhead is introduced by our defense system.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Fengwei Zhang for their insightful comments on improving our work. This work is partially supported by U.S. ONR grants N00014-16-1-3214 and N00014-18-2893, NSF CNS-1815650, the National Natural Science Foundation of China under GA No. 61802398, the National Science and Technology Major Project of China under GA No. 2018ZX03001-010, the National Cryptography Development Fund under Award No. MMJJ20180222 and MMJJ20170215. Lingguang Lei is the corresponding author of this paper.

REFERENCES

[1] Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *ARM white paper* 3, 4 (2004).
[2] Android. 2017. Trusty TEE | Android Open Source Project. https://source.android.com/security/trusty/.

[3] ANTUTU. 2019. Aututu Benchmark. http://www.antutu.com/en/index.html.

[4] ARM. 2014. ARM Architecture Reference Manual. https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf.

[5] ARM. 2015. ARM CoreLink TZC-400 TrustZone Address Space Controller. https://static.docs.arm.com/100325/0001/arm_corelink_tzc400_trustzone_address_space_controller_trm_100325_0001_02_en.pdf.

[6] ARM. 2016. ARM Cortex-A53 MPCore Processor Technical Reference Manual. https://developer.arm.com/docs/ddi0500/g.

[7] ARM. 2016. ARM Cortex-A9 Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.100511_0401_10_en/arm_cortexa9_trm_100511_0401_10_en.pdf.

[8] ARM. 2016. Juno r2 ARM Development Platform Technical Reference Manual. https://developer.arm.com/docs/ddi0515/f/juno-r2-arm-development-platform-soc-technical-reference-manual.

[9] ARM. 2017. ARM virtualization. https://developer.arm.com/docs/100942/0100/aarch64-virtualization.

[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] 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.

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

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure:{SGX} cache attacks are practical. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*.

[15] Robert Buhren, Julian Vetter, and Jan Nordholz. 2016. The threat of virtualization: Hypervisor-based rootkits on the ARM architecture. In *International Conference on Information and Communications Security*. Springer, 376–391.

[16] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*. 18–20.

[17] Rui Chang, Liehui Jiang, Wenzhi Chen, Yang Xiang, Yuxia Cheng, and Abdulhameed Alelaiwi. 2017. MIPE: a practical memory integrity protection method in a trusted execution environment. *Cluster Computing* 20, 2 (2017), 1075–1087.

[18] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 2–13.

[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] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2018. Prime+ count: Novel cross-world covert channels on arm trustzone. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 441–452.

[21] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. 2016. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 565–578.

[22] Dawei Chu, Yuewu Wang, Lingguang Lei, Yanchu Li, Jiwu Jing, and Kun Sun. 2019. OCRAM-Assisted Sensitive Data Protection on ARM-Based Platform. In *European Symposium on Research in Computer Security*. Springer, 412–438.

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

[24] ECG. 2019. Source code of Ginseng. http://download.recg.org.

[25] Samsung Electronics. 2013. Samsung KNOX. http://www.samsung.com/global/business/mobile/solution/security/samsung-knox.

[26] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 287–305.

[27] 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*.

[28] Google. 2012. QSEEComAPI.h. https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/QSEEComAPI.h.

[29] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[30] 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.

[31] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 38–55.

[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] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *Proceeding of usenix security symposium*.

[34] Huawei. 2017. Huawei SecureOS. http://developer.huawei.com/cn/consumer/devunion/ui/server/SecureOS.html.

[35] Jinsoo Jang, Changho Choi, Jaehyuk Lee, Nohyun Kwak, Seongman Lee, Yeseul Choi, and Brent Byunghoon Kang. 2016. Privatezone: Providing a private execution environment using arm trustzone. *IEEE Transactions on Dependable and Secure Computing* 15, 5 (2016), 797–810.

[36] Jinsoo Jang and Brent Byunghoon Kang. 2018. Retrofitting the partially privileged mode for TEE communication channel protection. *IEEE Transactions on Dependable and Secure Computing* (2018).

[37] 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)*.

[38] Donghyun Kwon, Jiwon Seo, Yeongpil Cho, Byoungyoung Lee, and Yunheung Paek. 2019. PrOS: Light-weight Privatized Secure OSes in ARM TrustZone. *IEEE Transactions on Mobile Computing* (2019).

[39] Wenhao Li, Yubin Xia, Long Lu, Haibo Chen, and Binyu Zang. 2019. TEEv: virtualizing trusted execution environments on mobile platforms. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2–16.

[40] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. 2014. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 409–420.

[41] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*. 549–564.

[42] 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.

[43] 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.

[44] 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).

[45] Saeed Mirzamohammadi and Ardalan Amiri Sani. 2018. The Case for a Virtualization-Based Trusted Execution Environment in Mobile Devices. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*. 1–8.

[46] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. Cachezoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 69–90.

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

[48] 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.

[49] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM TrustZone to build a trusted language runtime for mobile applications. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 67–80.

[50] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 3–24.

[51] Andrew Sloss, Dominic Symes, and Chris Wright. 2004. *ARM System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

[53] 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)*.

[54] 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)*.

[55] Min Hong Yun and Lin Zhong. 2019. Ginseng: Keeping Secrets in Registers When You Distrust the Operating System.. In *NDSS*.

[56] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. 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*. 203–216.

[57] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Y Thomas Hou. 2016. CacheKit: Evading memory introspection using cache incoherence. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 337–352.

[58] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. 2016. Case: Cache-assisted secure execution on arm processors. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 72–90.

[59] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. 2016. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive* 2016 (2016), 980.

[60] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. 2019. SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1723–1740.

A SHAREABILITY ATTRIBUTE ON ARM PROCESSORS

We conduct a series of experiments to better understand the impacts of the *shareability* attribute on the non-secure L1 data caches [4]. We conduct the experiments on the i.MX6Quad sabre development board, which equips with a quad-core ARM Cortex-A9 processor running at 1.2GHz and 1GB DDR3 SDRAM. Since the board has only one cluster, the *inner shareability domain* is equal to the *outer shareability domain*. In other words, the *inner shareable* has the same effect as the *outer shareable*. Since the *shareability* attribute is utilized to enforce value coherence when the same data is accessed by multiple cores, we make all four cores access the same physical memory (hereinafter referred to as *test memory*). We make all cores run in the normal world, so that the L1 cache accessed by them will all be non-secure. To eliminate the potential impacts on L1 cache introduced through L2 cache and memory, we disable the L2 cache and set the *test memory* as secure memory. We opt to set the *test memory* as secure rather than utilizing the memory isolation scheme in SANCTUARY, since that scheme is only simulated through the ARM Fast Models virtualization tools and is not achievable on the actual development boards. To prevent the raising of external abort due to accessing secure memory from the normal world, we set the *test memory* as *write-back, write-allocate* and all other memory as *non-cacheable*. As such, the writing of *test memory* will be buffered and locked in the L1 cache, and will not be synchronized to memory. The *test memory* and the corresponding L1 data cache are all initialized to `zero`.

Table 1: L1 Cache When Enabling Shareable Attribute

Shareability Attribute of the Cores	Value on the Core's L1 Data Cache	
	After Writing 0xffff to Core_0	After Writing 0xdddd to Core_1
Core_0 (Shareable)	0xffff	0xdddd
Core_1 (Shareable)	0xffff	0xdddd
Core_2 (Shareable)	0xffff	0xdddd
Core_3 (Shareable)	0xffff	0xdddd

Table 2: L1 Cache When Disabling Shareable Attribute

Shareability Attribute of the Cores	Value on the Core's L1 Data Cache	
	After Writing 0xffff to Core_0	After Writing 0xdddd to Core_1
Core_0 (Non-shareable)	0xffff	0xffff
Core_1 (Shareable)	0x0	0xdddd
Core_2 (Shareable)	0x0	0xdddd
Core_3 (Shareable)	0x0	0xdddd

We first investigate the impacts of data coherency when enabling the shareability attribute. Specifically, we set the *test memory* as *shareable* for all the four cores. Then, we store (i.e., write) `0xffff` to core_0's L1 data cache mapping to *test memory*, and load (i.e., read) L1 data cache of each core addressed through *test memory*. As illustrated in Table 1, value `0xffff` on core_0's L1 data cache is synchronized to the other three cores. Thereafter, we store `0xdddd` to core_1's L1 data cache, and find all cores' L1 data caches are synchronized again. It shows that the data on one core's L1 data cache could be leaked out to and manipulated by another core, when the two cores run in the normal world and the corresponding memory is set as *shareable* for both cores. Table 2 illustrates the results obtained by disabling the shareability attribute. Particularly, we modify the *test memory*'s cache attribute as *non-shareable* (i.e., *inner&outer non-shareable*) for core_0 and repeat the experiment. It shows that data on the core's L1 data cache could not be leaked out to or manipulated by another core, when the corresponding memory is set as *non-shareable* for that core.

To set different cache attributes for multiple cores when accessing the same physical memory region (i.e., the *test memory*), we construct four page table entries, which map to the same physical memory region and define different cache attributes for this physical memory region. After assigning one entry to each core, the four cores can access the same memory region with different cache attributes.

B EVALUATION OF THE DEFENSE SYSTEM

We evaluate the overhead introduced by our defense system based on the prototype implemented on the i.MX6Quad SABRE development board, which is equipped with a quad-core ARM Cortex-A9 processor running at 1.2GHz with 1GB DDR3 SDRAM. To minimize the noise in the experiments, we run each test with 1,000 iterations and report the average.

We first explore the overhead on security-sensitive applications due to the enforced cache attributes. Particularly, we run an AES encryption application in one IEE, and evaluate its execution time when the memory is configured with different cache attributes. In Table 3, the default configuration for most memory region is shown in the second column, where `S`, `WB`, `WA`, `With L1&L2` means setting the cache attributes as *shareable* [2], *inner write-back write-allocate, outer write-back write-allocate*. The attributes enforced by our defense system are shown in the fourth column, where `non-S`, `WT`, `non-WA`, `Without L2` represents *non-shareable, inner write-through non write-allocate, outer non-cacheable*. The cache attributes illustrated in the third column are similar to the ones in our defense system, but with L2 cache enabled. The experimental results show that our defense system introduces around 90% overhead comparing to the default setting, and it is mainly caused by disabling L2 cache. We also observe that for the IEE systems that disable the L2 cache for protection (e.g., SANCTUARY), our defense system only introduces negligible additional overhead.

Table 3: AES Encryption Time (in Milliseconds)

Payload (Bytes)	S, WB, WA, With L1&L2	non-S, WT, non-WA, With L1&L2	non-S, WT, non-WA, Without L2
1024	3.9	4.5	6.8
2048	7.5	8.8	13.4
4096	15.4	18.1	28.7

* `"S,WB,WA,With L1&L2"`: *shareable, inner write-back write-allocate, outer write-back write-allocate*;
`"non-S,WT,non-WA,With L1&L2"`: *non-shareable, inner write-through non write-allocate, outer write-through non write-allocate*;
`"non-S,WT,non-WA,Without L2"`: *non-shareable, inner write-through non write-allocate, outer non-cacheable.*

[2] Since the i.MX6Quad board has only one cluster, *shareable* is equal to *inner shareable* or *outer shareable*.

Then, we evaluate the overhead on the rich OS introduced by the additional cross-domain context switches enforced on each page table updating operation. We first study the overall overhead through a comprehensive benchmark suite, i.e., AnTuTu 2.9.4 [3]. It measures the performance in integer computation, float point operation, 2D and 3D graphic rendering etc. The results are illustrated in Table 4, which shows 2.65% overall overhead is introduced on the execution of rich OS. The primary reason for the 17.74% overhead on the `Database I/O` operations is the need of building a mass of page table mappings when copying data from the disk to the memory.

Table 4: Benchmark Results on Rich OS

Test Item	Protection Disabled	Protection Enabled	Overhead
RAM	486	475	2.26%
CPU Integer	698	692	0.86%
CPU Float-point	567	564	0.53%
2D Graphics	282	281	0.35%
3D Graphics	861	852	1.05%
Database I/O	310	255	17.74%
SD Card Write	38	36	5.26%
SD Card Read	186	182	2.15%
Total	3428	3337	2.65%

We also evaluate the overhead on the operations that involves frequent page table updating, i.e., the system booting and application loading. As illustrated in Table 5, the test item `Kernel` records the loading time from the hardware booting to the starting of the `init` process. `Android Home` refers to the initialization time of the Android `Launcher` process. We also test the loading time of four Android applications, including `Calculator`, `Calendar`, `Music` and `Settings`. Overall, the loading overhead for both kernel and applications is less than 10% in all evaluation scenarios.

Table 5: Loading Time Results on Rich OS (in Seconds)

Test Item	Protection Disabled	Protection Enabled	Overhead
Kernel	22.26	23.71	6.51%
Android Home	87.42	89.81	2.73%
Calculator	3.01	3.22	6.98%
Calendar	3.14	3.34	6.37%
Music	1.26	1.37	8.73%
Settings	3.77	3.95	4.77%