TEEv: Virtualizing Trusted Execution Environments on Mobile Platforms

Wenhao Li
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
China
liwenhaosuper@sjtu.edu.cn

Yubin Xia
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
China
xiayubin@sjtu.edu.cn

Long Lu
Northeastern University
l.lu@northeastern.edu

Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
China
haibochen@sjtu.edu.cn

Binyu Zang
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
China
byzang@sjtu.edu.cn

Abstract

Trusted Execution Environments (TEE) are widely deployed, especially on smartphones. A recent trend in TEE development is the transition from vendor-controlled, single-purpose TEEs to open TEEs that host Trusted Applications (TAs) from multiple sources with independent tasks. This transition is expected to create a TA ecosystem needed for providing stronger and customized security to apps and OS running in the Rich Execution Environment (REE). However, the transition also poses two security challenges: enlarged attack surface resulted from the increased complexity of TAs and TEEs; the lack of trust (or isolation) among TAs and the TEE.

In this paper, we first present a comprehensive analysis on the recent CVEs related to TEE and the need of multiple TEE scheme. We then propose TEEv, a TEE virtualization architecture that supports multiple isolated, restricted TEE instances (i.e., vTEEs) running concurrently. Relying on a tiny hypervisor (we call it TEE-visor), TEEv allows TEE instances from different vendors to run in isolation on the same smartphone and to host their own TAs. Therefore, a compromised vTEE cannot affect its peers or REE; TAs no longer have to run in untrusted/unsuitable TEEs. We have implemented TEEv on a development board and a real smartphone, which runs multiple commercial TEE instances from different vendors with very small porting effort. Our evaluation results show that TEEv can isolate vTEEs and defend all the known attacks on TEE with only mild performance overhead.

CCS Concepts • **Security and privacy → Mobile platform security**;

Keywords TEE; TrustZone; Virtualization; Mobile security

ACM Reference Format:
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 (VEE '19), April 13–14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3313808.3313810

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VEE '19, April 13–14, 2019, Providence, RI, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313810

1 Introduction

Trusted Execution Environment (TEE)[1] is getting widely used, especially on mobile devices. Since its debut, TEE development has gone through the following stages. In the first stage, TEE is mainly used for secure booting which checks the validity of the loaded OS in the Rich Execution Environment (REE), e.g., in Motorola X/G/E mobile phones. In the second stage, TEE starts supporting more functionalities, including encryption, fingerprint authentication, mobile payment, trusted user interface (TUI), Digital Rights Management (DRM), etc. Some phone manufacturers also leverage TEE's high privilege to perform runtime protection of the REE, e.g., TrustZone Integrity Measurement Architecture (TIMA) in Samsung's KNOX [39]. Up to this stage, the functionalities of a TEE are typically fixed (i.e., cannot change after manufacturing). For example, if a third-party company

[1] In this paper, by TEE, we mean the software running in the secure world of ARM TrustZone except the trusted applications (TAs). TEE offers an execution environment for TAs.

Table 1. Real-world mobile commercial TEE providers and products
.

	Vendor	TEE Name	Commonly Used Arch
Chip Vendor	Qualcomm	QSEE [15]	ARM32, ARM64
	Spectrum	Spectrum TEE	ARM32
	HiSilicon	TrustedCore [38]	ARM32
TEE Vendor	Apple	Enclave [37]	ARM32
	TrustKernel	T6 [18]	ARM32
	Trustonic	Kinibi [19]	ARM32
	Google	Trusty [11]	ARM32
	Linaro	OP-TEE [12]	ARM32
	SierraWare	SierraTEE [16]	ARM32
	Proven&Run	ProvenCore [14]	N.A.

needs to deploy a Trusted Application (TA) for mobile payment in TEE, it has to pre-install the TA before devices leave factories.

Today, the third stage of TEE development is unfolding. New TEEs now support dynamic (post-manufacture) installations of TAs. There has already been a few TA app stores from which phone users can easily download and install TAs, just like installing an ordinary app, e.g., Samsung's trustlets [39] and TrustKernel's TEEReady [50]. GlobalPlatform [10] has proposed a set of APIs for communication between TEE and REE, which nowadays most commercial TEEs follow as a de facto standard. ARM also leads a group of TEE vendors to create the Open Trust Protocol (OTrP) [22], which combines a secure architecture with TA management. The aim of these efforts is to enhance the compatibility and deployability of TAs with different TEEs.

However, this increasing openness and flexibility make TEE more complex and enlarge the attack surface. To support various TAs, TEEs are growing with more functionalities, which lead to significant increase of the size of Trusted Computing Base (TCB). Meanwhile, as TEEs need to support dynamical installation of new TAs, it is no longer feasible for manufacturers to perform complete security tests in factory. At present, there are more than ten TEE vendors in the market (as shown in Table 1).

The larger TCB/attack surface of TEE and the more dynamic TEE ecosystem bring two challenges: weakening security and increasing distrust. First, TEE usually has the highest privilege of the system, which means that if the TEE is compromised, the security of the entire system can be breached. For example, an attacker can leverage a bug of TEE to write arbitrary memory of REE which is known as Boomerang attack (CVE-2016-8764). Meanwhile, if a TEE is compromised, it may further attack TAs and cause leakage of secrets like fingerprint data (CVE-2015-4422) or keys (CVE-2015-6639). It is expected that the number of TEE vulnerabilities will keep increasing in the near future due to the enlarged attack surface.

Second, since there is only one TEE in each device currently, all TAs have to unconditionally trust the TEE. However, this trust is becoming more and more difficult to establish as TAs from different sources are entering into TEEs. TA vendors may require a higher security standard or more security features than some TEEs provide. It is also possible that a TA vendor is in conflict of interest with a TEE vendor and prefers to trust/run in another TEE.

In this paper, we first present a comprehensive analysis on the recent CVEs related to TEE and the need of multiple TEE scheme. We then propose TEEv, a TEE virtualization architecture that supports multiple isolated, restricted TEE instances (i.e., vTEEs) running concurrently. Relying on a tiny hypervisor (we call it TEE-visor), TEEv allows TEE instances from different vendors to run in isolation on the same smartphone and to host their own TAs. Therefore, a compromised vTEE can affect neither its peers nor the REE; TAs no longer have to run in untrusted/unsuitable TEEs. Meanwhile, our system also supports controlled interaction between TAs running on different vTEEs (e.g., between a fingerprint authentication TA and a payment TA), which makes our system more practical and deployable.

The design and implementation of TEEv overcome several challenges. First, there is no hardware support for virtualization in the secure world. Thus, we have to run the TEE-visor and vTEE in the same privilege while still keeping the isolation between them. Second, the virtualization should be completely transparent to REE applications and TAs, which have been deployed on billions of devices. Third, we also need to multiplex different peripherals to support TAs like fingerprint, frame buffer, etc., which are not designed for multiplexing. Finally, we need to minimize the modification to existing TEEs and also keep the overhead as low as possible to make TEEv practical.

We have implemented the system on a development board and a real mobile phone. It can run multiple commercial TEE instances from different vendors with very small porting effort. Our evaluation results show that TEEv can isolate vTEEs and defend all the known attacks on TEE with only mild performance overhead.

This paper makes the following contributions:

- A TEE virtualization architecture without hardware virtualization support to enable multiple real-world vTEE instances running simultaneously.
- A novel design enforcing the isolation between different vTEEs, as well as between vTEE and REE and is transparent to existing REE and TAs, and only needs small modification to the existing TEEs.
- An implementation of TEEv on real mobile phones for security and performance evaluation, which shows that the system can defend all the existing vulnerabilities of TEEs and the performance overhead is small.

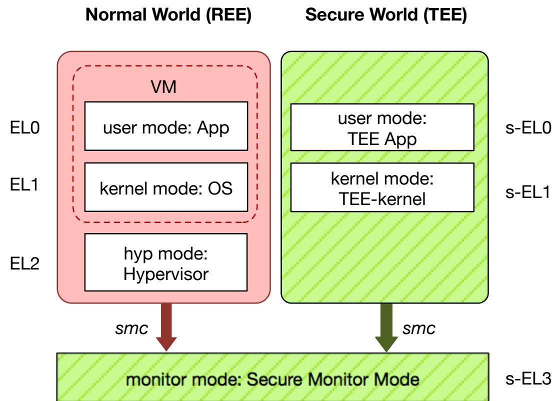

Figure 1. ARM TrustZone architecture.

2 Background & Motivation

2.1 TEE Architecture

ARM TrustZone [21] is a hardware-based security mechanism available on most mobile devices today. It consists of a set of CPU extensions for isolating processors, memory and peripherals between two execution modes, namely the Normal World and the Secure World (as Figure 1). The normal world has three execution levels: EL0 for user applications, EL1 for operating system, and EL2 for hypervisor. The secure world also has three execution levels but with a subtle difference: s-EL0 for user-level trusted applications, s-EL1 for trusted operating system, and s-EL3 for monitor mode. Note that the secure world does *not* have s-EL2 yet for hypervisor [2].

The secure world can control all the computing resources, while the normal world can only access resources that are assigned to it. The normal world is used to run a functionally rich and complex commodity OS (REE). In contrast, the secure world is for hosting security-critical services (TEE). The world switching is done through a special instruction named "secure monitor call" (`smc`), which traps the processor to the monitor mode for context saving and restoring. The code running in the monitor mode is called a *Secure Monitor*, which is usually provided by chip vendors by default, e.g., ARM Trusted Firmware (ATF). Similarly, memory can also be split to two parts: the normal part and the secure part. The normal world cannot access the secure part but the secure world can access all the memory.

This two-world isolation also expands to all the peripherals and interrupts. Once an interrupt occurs, if it is assigned to the secure world, TrustZone will switch the processor to the secure world mode to handle it. A peripheral can be assigned to both worlds at the same time or secure world accessible only. The normal world cannot access the secure world's protected peripherals. The secure world controls the assignment of all the peripherals. Such assignment can be changed during runtime.

[2]ARMv8.4-A has proposed s-EL2 [2], but has no any implementation yet.

Currently, TAs are not designed as standalone applications and TEE follows a request-response execution model: a TEE executes only when receving requests from REE to TA. Thus the scheduling model of TEE's main thread follows the calling application of REE OS.

2.2 Vulnerabilities of TEE

We collect published vulnerabilities of TEEs, which are listed in Table 6 in the appendix. Many vulnerabilities are critical to a large number of devices. A post of Google's Project Zero [1] shows how to exploit two major TEEs, Qualcomm's QSEE and Trustonic's Kinibi, on real mobile devices. It concludes that " *... despite their highly sensitive vantage point, these operating systems currently lag behind modern operating systems in terms of security mitigations and practices.*

Many of the CVEs are not well documented and have long delay before publishing. For example, CVE-2016-10238 was first uncovered in 2016 but was not released until March 2017, and the description is brief: "*... The technical details are unknown and an exploit is not publicly available.*"[3] Still, we try to analyze each vunlerability with best effort.

We find that there are three basic categories of TEE-related vulnerabilities: First, TEE vulnerabilities can lead to secret data leakage or arbitrary code execution, e.g., CVE-2017-0518/0519, CVE-2016-2431/2432, etc. Second, TEE vulnerabilities allow one TA to affect the security of other TAs, e.g., CVE-2016-0825 and CVE-2015-6639/6647. Third, TEE vulnerabilities can be leveraged to achieve privilege escalation in the REE, e.g., CVE-2016-8762/8763/8764. The causes of these attacks include the lack of isolation and the semantic gap between different execution environments.

One of the major reasons leading to the above vulnerabilities is that there are a lot of interactions between applications running in REE and TEE, which are usually done through shared memory using pointers. As stated by Machiry et al. [44]: "*TEE has very limited visibility into the untrusted environment's security mechanisms*", which is referred to as the "semantic gap" between TEE and REE. Thus, a malicious CA (client application running in REE) may request the TEE to overwrite data in the REE kernel. Although the CA itself cannot do so since it has no privilege, the TEE has higher privilege and may violate the REE's security mechanism by logical mistakes.

2.3 Needs for Multiple Isolated TEEs

Currently, all TAs installed on a device have to trust the only TEE kernel available on that device. This forced trust is being increasingly questioned as the TA ecosystem becomes more diverse and open. Consider a mobile payment TA running in a TEE offered by the phone manufacture. If an attacker leverages a bug of TEE to steal the private key of the payment TA, she can actually steal user's money directly. In practice, the

[3]https://vuldb.com/?id.101419

payment company ends up compensating the user for the fault of TEE, like AliPay [3]. The point here is that currently a TA has to trust the only available TEE on a device, despite that the TEE might not meet the TA's security requirements or is not trustworthy to the TA. In contrast, if the system supports multiple TEEs, then the phone manufacturer can offer a default TEE (aka., *system TEE* in this paper) for running manufacturer's TAs. At the same time, TAs with different security requirements can install TEE instances they trust.

Protecting TEE from attacks by hardening TEE itself could be one research direction for the problem we address. However, as shown in Table 1, there are various TEE OS products from different vendors widely deployed in billions of devices, each of them has diverged design and implementation. It is hard to protect them one by one in practice.

3 System Overview

We summarize our observations in the above sections:

- More and more TAs are getting deployed and TEE now supports dynamical installation.
- Different TEE vendors have different designs and implementations of TEE.
- Both TA and TEE OS are premature in security which already leads to critical vulnerabilities. A compromised TEE may affect the security of REE and TAs.
- Different TA providers are not willing but has to trust the only TEE on the device.

By introducing a thin hypervisor, called TEE-visor, our system allows multiple isolated TEE instances (vTEEs) to run simultaneously. The design brings about three benefits: isolation, restricted interaction and multiple vTEEs. First, TEE-visor can enforce two types of isolation: among vTEEs, between vTEE and REE. Thus, even if a vTEE has vulnerabilities and gets compromised, it cannot affect its peers, which localizes TEE attacks, including 0-day exploits, and prevents them from breaching the security of the entire system. Second, TEE-visor restricts the interaction between different execution environments by requiring the use of designated communication channels, interfaces, and semantics. Third, by supporting multiple vTEEs, the TAs can choose their trusted and suitable TEE instance instead of having to trust the default TEE.

3.1 Threat Model and Assumptions

In TEEv, the TEE-visor is globally trusted as the TCB of the system. We assume that the boot process is protected by the hardware and the TEE-visor cannot be compromised after bootup.

A vTEE does not trust its neighbouring vTEEs. From a vTEE's perspective, other vTEEs may be controlled by an attacker or be vulnerable. Within each vTEE, the kernel does not trust any TAs and each TA does not trust other TAs by default. The system-vTEE(introduced in Section 3.3) is *partially* trusted by other vTEEs. A vTEE can choose to trust a service provided by the system-vTEE, e.g., fingerprint authentication, instead of trusting the entire system-vTEE. In another word, a compromised system-vTEE cannot affect the security of those vTEEs that do *not* rely on its services.

Similarly, the REE also *partially* trusts vTEEs if it needs services provided by these vTEEs. Such trust is at application level instead of system level, which means that even if a vTEE is malicious, it cannot illegally access REE's data; only the REE app that relies on vTEE's services might be affected.

We do not consider physical attacks between vTEEs or between REE and vTEE. We also do not consider logic bugs within the apps, e.g., a bug may mistakenly map all of the app's memory as shared with a TA, which allows a TA to access all the app's data. Side channel attacks that could break the isolation of TrustZone are out of scope in this paper, as discussed in Section 5.

3.2 Challenges

There are several technical challenges that our design overcomes.

1. Unlike the *hyp* mode in the normal world, there is no hardware virtualization support in the secure world (i.e., no EL2 in secure world). Thus, TEE kernel is supposed to be running with the highest privilege to control all the computing resources. There is no higher privilege to restrict the behavior of TEE instance. We need a way to ensure the isolation between vTEE and TEE-visor and to impose restrictions on each vTEE.
2. Our system should be compatible with most devices. Different mobile devices usually have different peripherals, which require their own drivers. In the previous mobile payment example, the third-party vTEE may need to use fingerprint reader to authenticate the user for quick payment. However, it is not practical for all the vTEE vendors to have drivers for all existing fingerprint readers.
3. The TCB of our system should be kept small. TEE-visor should contain only essential functionalities to minimize its attack surface.
4. The system should be non-intrusive to existing systems. It should not require major modification to existing TEE implementations or incur significant performance overhead.

3.3 Overview of TEEv

Our system aims to enable multiple isolated vTEEs (virtualized TEEs) in one secure world on a single device. The goals entail strong isolation between REE and vTEE, as well as isolation among vTEEs. Using our system, TA vendors can encapsulate trusted services into their own vTEEs without having to use or trust the phone manufacturer's TEE. This new capability allows the TA ecosystem to be more dynamic and open without sacrificing the security of each TA or the

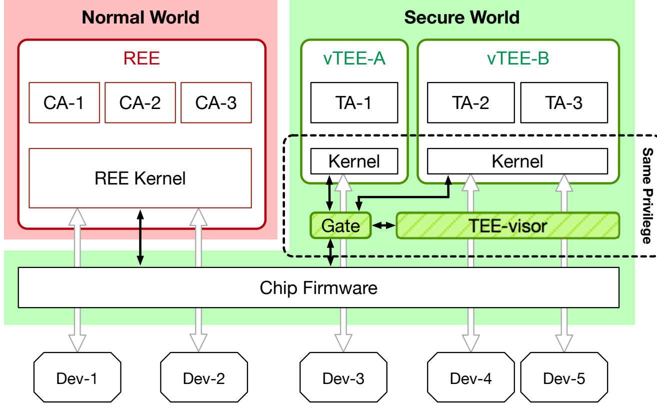

Figure 2. Architecture of TEEv. In the secure world, TEE-visor and the kernel of vTEE run in the same privilege but from different address spaces. The "gate" is a piece of code mapped in both the TEE-visor and vTEE instances at the same address.

entire device. Meanwhile, a vulnerable vTEE cannot affect the REE or other vTEEs. Our system contains the following components, as shown in Figure 2:

- a TEE-visor running as a hypervisor for managing and securing multiple vTEEs;
- a gate mechanism for secure context switches between different execution environments and the TEE-visor;
- a secure communication channel mechanism for different execution environments to safely interact.

Specifically, the TEE-visor is in charge of resource management in the secure world. It controls the scheduling of vTEEs, the mapping of memory, as well as the assignment of all the peripherals. It also manages the installations, updates, and removals of third party vTEE instances. The TEE-visor runs at the same privilege level as the vTEEs, but is isolated from other environments to ensure its security. Details can be found in Section 4.1.

The interaction between REE and vTEE is only allowed through the secure communication channel established by the TEE-visor. Either vTEE or REE can create a communication channel with another execution environment by sending a request to the TEE-visor. The request contains the identifications of both environments as well as allocated memory pages from the requester for sharing. Upon such a request, the TEE-visor first authenticates both environments and then maps the shared memory pages by adjusting the page table in each environment. The TEE-visor provides well-defined interfaces for vTEE/REE to operate on communication channels, including creating/deleting, attaching/detaching memory pages, etc.

There is one system-vTEE, which includes an TEE OS that contains device drivers for peripherals like fingerprint readers, as well as several common TAs, e.g., KeyMaster and GateKeeper TA required by Android OS. The system-vTEE itself is pre-installed and restricted, which does not allow new TAs to be installed dynamically. Apps in the REE can require the TEE-visor to install third-party vTEEs, which could be a full-fledged TEE supporting multiple TAs, or just a specialized TEE with only a few functionalities. A vTEE is activated only when it receives a service request from other environments or devices (e.g., handling interrupts).

We now use an example to illustrate the interactions among the components of our system. Consider an off-the-shelf mobile phone with only one vTEE (system-vTEE) initially. The TEE-visor assigns the fingerprint reader to the system-vTEE by default. Later, the phone owner installs one mobile payment app. The payment app is delivered with its own vTEE and TA, which is installed by the TEE-visor during the app installation. The signature of the vTEE image will be checked by the TEE-visor in advance. In most of the time, the pay-vTEE is not active. When the user needs to pay and authenticate herself using her fingerprint, the app sends a request with the pay-vTEE's ID to the TEE-visor to establish a communication channel with the pay-TA in pay-vTEE. As a result, the pay-vTEE is activated. It establishes a communication channel with the system-vTEE, which is asked to verify the user's fingerprint. The system-vTEE then reads and authenticate user's fingerprint and return the result to the pay-vTEE through the channel. The pay-TA then gets the result and continues the process of payment.

4 Design

4.1 Isolation Enforcement

Isolation between TEE-visor and vTEE: The first challenge we tackle while designing our system is to isolate the TEE-visor from vTEEs. Our solution is inspired by the prior work on *same privilege isolation* [23, 32, 34]. Specifically, we introduce a small TEE hypervisor named TEE-visor, that runs at the *logically* lowest execution level (i.e., highest privilege) and maintains an exclusive control over the MMU (memory management unit). Each time a CA (Client Application) in REE needs to communicate with a TA in TEE, the CA has to explicitly send a request to the TEE-visor, which in turn grants the TA access to one or more CA's memory pages.

In order to ensure that only the TEE-visor can configure the MMU, all the related instructions must be protected. These instructions include those accessing SCTLR (system control register), TTBR (translation table base register), VBAR (vector base register), DACR (domain access control register), TLB flush and MMU enable/disable register. Our design disallows vTEE instances to use these instructions. We enforce this by scanning the binary of vTEE and ensuring no such instructions exist during vTEE installation or loading. With vTEE instances deprived of direct memory management rights, the TEE-visor gains full and exclusive control over the page table. As a result, the TEE-visor can protect the integrity and confidentiality of its own code and data. Furthermore, leveraging the control over MMU and peripherals,

vTEE's View		TEE-visor's View		REE's View	
vTEE Data	RWNX	vTEE Data	RWNX	vTEE Data	NULL
vTEE Code	RX	vTEE Code	RWNX	vTEE Code	NULL
Gate Data	RWNX	Gate Data	RWNX	Gate Data	NULL
Gate Code	RX	Gate Code	RX	Gate Code	NULL
TEE-visor Data	NULL	TEE-visor Data	RWNX	TEE-visor Data	NULL
TEE-visor Code	NULL	TEE-visor Code	RX	TEE-visor Code	NULL
REE Data	NULL	REE Data	RWNX	REE Data	RWNX
REE Code	NULL	REE Code	RWNX	REE Code	RX
World Shared Mem	RWNX	World Shared Mem	RWNX	World Shared Mem	RWNX

Figure 3. Memory Mapping of vTEE, TEE-visor and REE

the TEE-visor also isolates vTEE instances and protects the entry gates.

Isolation between vTEEs and vTEE/REE: TEE-visor is the TCB of our system. It can access all the physical memory. Upon booting, TEE-visor will identify and manage every memory usage permissions for further memory management. TEE-visor restricts each vTEE to its own memory view (i.e., a partial view of the entire memory). Specifically, a vTEE can only access its own data and code. It cannot access the memory space assigned to other vTEE instances or REE or TEE-visor without explicitly authentication. Figure 3 shows the memory views of a vTEE, the TEE-visor, and the REE, respectively. This design ensures that a vTEE has no higher privilege than the REE, establishing a mutual distrust connection between vTEE and REE.

vTEE kernel life-time code integrity: Ensuring each vTEE kernel life-time code integrity is essential for TEEv, otherwise attackers may break our isolation guarantee by exploiting one vTEE and inject code in vTEE, which leads to executing arbitrary privileged code in the same privileged level as TEE-visor. Since TEE-visor controls page tables of all vTEEs, TEE-visor enforces that all vTEE kernel code section is mapped as read-only and except the vTEE kernel code section, all other pages mapped by vTEE are set to privileged execution never (*PXN*) to prevent them from executing in kernel privilege.

4.2 Gate for Context Switching

A vTEE can call services provided by TEE-visor through a *gate*. A gate is used for switching the execution between TEE-visor and a vTEE. It is a piece of code mapped in the TEE-visor and all vTEE instances at the same address. Any vTEE or the TEE-visor can issue a context switch through the same code.

The gate mechanism controls the process of switching between a vTEE and the TEE-visor. It is designed to achieve the following goals. First, the gate should be the only entry point for context switch and cannot be bypassed. Second, the switching process should be atomic to avoid partial executions or shortcuts. Third, the integrity of the gate should be ensured. We list the critical part of the gate code in Listing 1.

To unify the gate entry, the gate code is executed in monitor mode through *SMC* intruction call with interrupt disabled. The data used by the gate code (for example, teevisor_pt in Line 14 and Line 28) is mapped as read-only for all vTEE instances to avoid directly tamper the page table pointer of TEE-visor by vTEEs. The exit gate routine from TEE-visor to vTEE is similar to the entry gate code.

A potentially compromised vTEE can attempt to attack the gate in two ways: interrupting the execution of the gate to pollute general register values or jumping to the middle of the gate. We will describe how we defend these attacks in the following. Note that the integrity of vTEE TTBR register values (Line 2 to 12) will be further checked after entering TEE-visor.

Listing 1. The code snippet of entry gate

```
1  /* Temp save vTEE TTBR regs */
2  cpsid     if                 /* disable irq and fiq */
3  mrc       p15, 0, r1, c12, c0, 0 /* read VBAR */
4  push      {r1}
5  mrc       p15, 0, r1, c1, c0, 0  /* read SCTLR */
6  push      {r1}
7  mrc       p15, 0, r1, c2, c0, 1  /* read TTBR1 */
8  push      {r1}
9  mrc       p15, 0, r1, c2, c0, 0  /* read TTBR0 */
10 push      {r1}
11 mrc       p15, 0, r1, c2, c0, 2  /* read TTBCR */
12 push      {r1}
13 /* Load TEE-Visor TTBR Regs */
14 ldr       r0, =teevisor_pt
15 ldr       r1, [r0], #4
16 mcr       p15, 0, r1, c2, c0, 2  /* write TTBCR */
17 ldr       r1, [r0], #4
18 mcr       p15, 0, r1, c2, c0, 0  /* write TTBR0 */
19 ldr       r1, [r0], #4
20 mcr       p15, 0, r1, c2, c0, 1  /* write TTBR1 */
21 ldr       r1, [r0], #4
22 mcr       p15, 0, r1, c1, c0, 0  /* write SCTLR */
23 ldr       r1, [r0], #4          /* SP */
24 ldr       r1, [r0], #4
25 mcr       p15, 0, r1, c12, c0, 0 /* write VBAR */
26 /* Check jump-to-the-middle attacks from vTEE */
27 isb
28 ldr       r0, =teevisor_pt
29 ldr       r1, [r0], #4
30 mrc       p15, 0, r3, c2, c0, 2  /* read TTBCR */
31 cmp       r1, r3
32 bne       .
33 ldr       r1, [r0], #4
34 mrc       p15, 0, r3, c2, c0, 0  /* read TTBR0 */
35 cmp       r1, r3
36 bne       .
37 ldr       r1, [r0], #4
38 mrc       p15, 0, r3, c2, c0, 1  /* read TTBR1 */
39 cmp       r1, r3
40 bne       .
41 ldr       r1, [r0], #4
42 mrc       p15, 0, r3, c1, c0, 0  /* read SCTLR */
43 cmp       r1, r3
44 bne       .
45 ldr       r1, [r0], #4          /* SP */
46 ldr       r1, [r0], #4
47 mrc       p15, 0, r3, c12, c0, 0 /* read VBAR */
48 cmp       r1, r3
49 bne       .
50 /* Flush TLB */
```

```
51      mov       r0, #0
52      mcr       p15, 0, r0, c7, c5,  0
53      /* Jump to tee-visor main() */
54      dsb
55      isb
56      bl        teevisor_main
```

Defending Jump-to-the-middle Attack: Note that in Listing 1 there are two critical instructions in Line 18 and 20, which set `TTBR0` and `TTBR1` to point to the corresponding page tables. The code page containing these two instructions is mapped to the address space of vTEE , which could potentially be abused: a malicious or compromised vTEE may initialize *r1* with a crafted value and then jump directly to Line 18 (thus skipping Line 14-17) to set `TTBR0` with the crafted value. That means the malicious vTEEs can change the page table, which violates our security requirement that the page table can only be controlled by the TEE-visor. We call it *jump-to-the-middle attack*.

In order to defend against this attack, we add a piece of code (Line 27 to 49) to reload TTBRs to ensure that their values should be the physical addresses of TEE-visor's page table. However, if the malicious vTEE has successfully changed the page table, this piece of checking code may not exist in the new address space.

In TEEv, we leverage a hardware feature called *TLB lockdown* [17] to solve this problem. When a TLB lockdown is in place, the translation of cached addresses remains fixed. Therefore, switching page table has no effect on memory translation during a TLB lockdown. Using this hardware feature, TEE-visor locks the TLB for translating the gate code. Thus, even if a malicious vTEE issues jump-to-the-middle attack and change the page table to a forged one, it cannot violate the control flow of the gate, since the processor always executes the rest of the code (Line 19 to 56), which can detect such attacks.

Defending Interrupt-Execution Attack: The gate is assumed to execute in interrupt disabled state (for example execute in *SMC* handler which is interrupt disabled), However, a malicous vTEE may execute the gate code directly and pollute the general register by carefully controlling the interrupt timing. Similar to *jump-to-the-middle attack*, we need to ensure that interrupt is disabled during the gate transition. We guarantee the gate is executed in interrupt disabled state by intercepting the vector base register (see Section 4.1) of vTEE. The real exception base address is held by TEE-visor and upon receving an interrupt, TEE-visor will firstly check that the execution context is not in the gate during the interrupt occurs before forwarding the interrupt to the vTEE.

Fixing The Address Space ID Problem: It should be noted that an attacker may still use the processor's *tagged TLB* feature to bypass our TLB lockdown technique. Tagged TLB is introduced for performance optimization. Each TLB entry can be associated with a specific ASID (Address Space ID) so that after a context switch there will be no need for a TLB flushing. Abusing this feature, a malicious vTEE may set the TTBR with an ASID different from the ASID of locked TLB entries (TTBR has a field of ASID) through reusing the gate code (Line 18 and 20) by jump-to-the-middle attack or interrupt-execution attack. Now the processor thinks that a different address space is used since the ASID in TTBR is different from the ASID in TLB, and consequently, ignores our locked TLB. Thus, the next instruction will trigger a TLB miss and the processor will use the malicious page table in memory.

One straightforward solution to this problem is to simply disable the tagged TLB feature. This can work because all the page tables are controlled by the TEE-visor from the beginning. However, this solution will affect the global performance of TEE because now after every context switch there will be a TLB flush. In our system, we propose another solution by leveraging another TLB feature named "global entries" to retain the performance benefit of tagged TLB. There is a bit in each locked TLB entry (the 9th bit) indicating whether the entry is used for a single address space or globally. Once the bit is set, the TLB entry is valid across all address spaces. Thus, we set all the TLB entries for gate code pages as global entries. To avoid the locked TLB entries to be unlock by vTEEs, all TLB unlock/flush operations in vTEEs are delegated to TEE-visor.

Design Consideration and Limitation: The secure gate design described above is based on ARM32 for compatiability reason: all real-world TEE products support executing in ARM32 mode even in 64-bit ARM chips, but few support ARM64 mode (see Table 1). Note that this design could be easily applied to ARM64 mode. However, preventing *Jump-to-the-middle* attacks requires TLB lockdown, which we found out it may be not available in some 64-bit ARM chips: Cortex-A57 [8] and Cortex-A72 [9]. This limitation could be eliminated by removing explicitly instructions that update TTBR0 and TTBR1 in the gate and only activating TTBCR.N to provide deterministic switch, similar to SKEE [24]. In that case, we need to guarantee that vTEE does not use TTBR1 but only TTBR0. In real-world TEE, this guarantee does not hold by default and thus may require TEE providers to change the way how TEE manages memory. We argue that our assumption holds in most cases and this solution is useful in practical because currently billions of mobile and IoT devices support TLB lockdown feature.

4.3 Interaction between vTEEs and vTEE/REE

Upon loading a vTEE, TEE-visor allocates a secure memory region for vTEE execution and ensures it is not conflicted with other vTEE instances. TEE-visor keeps memory view of each vTEE and keep track of each memory mappings. If two vTEEs, or one vTEE and the REE, need to exchange data, they have to establish a secure communication channel with the help of the TEE-visor. The initiator sends a request to the

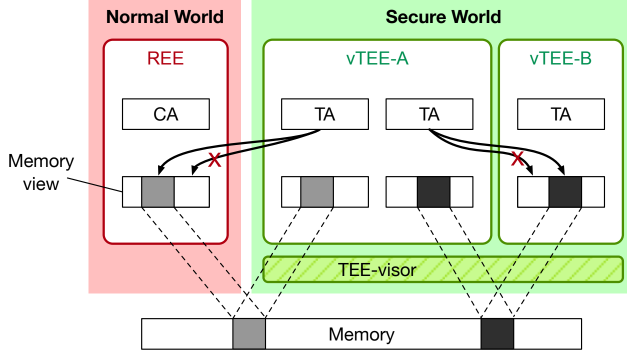

Figure 4. Interaction between vTEE/REE & between vTEEs.

TEE-visor with the identities of both environments as well as pre-allocated memory pages for sharing. The TEE-visor first authenticates the two environments and then maps the pages as shared for both environments.

Since there are more than one vTEEs in the system, an app in REE needs to explicitly specify which vTEE it intends to interact with. Specifically, the app first establishes a channel with the intended vTEE by sending a request to the TEE-visor with one or more shared memory pages and the ID of the vTEE as parameters. The TEE-visor then loads the vTEE instance corresponding to the ID, establishes a communication channel between the app and the vTEE instance, and switches to the vTEE to execute through the gate. Interaction can also happen between different vTEE instances, e.g., a vTEE asks for a service provided by the system-vTEE, as shown in Figure 4.

4.4 I/O Device Management

A TEE may control various peripheral devices, such as fingerprint reader, input device and framebuffer for TUI (Trusted UI), random number generator, secure storage like RPMB (Replay Protected Memory Block), various SEs (secure elements), etc. Traditionally, TEE kernel has the highest privilege and can control any device by assigning it to the secure world, as shown in Figure 5-(a). In our system, since the vTEEs are not fully trusted, we introduce a few new device management modes. Before describing these modes, we first share two observations on I/O devices' usage of TEE, which inspired our design.

The first observation is that different TAs usually do not share devices at the same time. TEE is inactive for most of the time and it is rare, if not impossible at all, that two TAs need to share one peripheral. In contrast, a virtual machine hypervisor (as opposed to TEE-visor) usually needs to multiplex I/O devices like disks and network cards among multiple VMs. The second observation is that the peripheral device drivers used in TEE are quite diverse. This brings challenges to our design, since it is not practical to require *all* third-party vTEE vendors to have all the device drivers.

Based on the two observations, we designed three modes of I/O management for different scenarios. The first mode is *delegation mode*, as shown in Figure 5-(b). In this mode, a device is controlled by the system-vTEE. A TA in system-vTEE offers an interface to TAs running in other vTEEs, acting as a proxy. The interface usually contains high-level APIs. For example, the system-vTEE directly controls a fingerprint reader. A fingerprint TA offers a service of fingerprint check. A TA in a third-party TEE may ask the fingerprint TA to authenticate the current user, and will get a result of "pass" or "fail". In this case, only the fingerprint TA will get the user's fingerprint data, so the diversity of different fingerprint readers is hidden from the third party vTEEs.

The second mode is *frontend/backend driver mode*, as shown in Figure 5-(c). This mode resembles the I/O para-virtualization mode in traditional hypervisor's architecture, such as *virtio* in KVM. The system-vTEE controls a peripheral and uses a back-end driver to offer general abstraction for each type of device. Thus, a third-party vTEE only needs to have one front-end driver for each device category. The communication between back-end and front-end driver is done through the secure channel mechanism. This mode is useful for those TAs that have higher security requirements. Take 2D facial authentication for example, [4] where a phone vendor pre-installs a 2D facial recognition algorithm in system-vTEE. Since different algorithms may have very different performance and accuracy, a phone vendor may tend to chose those with low latency and low false positive rate (e.g., for better user experience of unlocking). However, it is likely that such algorithms may have high false negative rate. A payment TA may not trust the pre-installed algorithm. Using the frontend/backend mode, the TA can ask for the raw data of the video stream and then use its own algorithm for facial recognition.

The third mode is *passthrough mode*, which is shown in Figure 5-(d). In this mode, the device is assigned to and controlled by one vTEE exclusively. The assignment as well as the revocation are done by the TEE-visor. Once a vTEE needs to access a device, it sends a request to the TEE-visor, which checks whether the device is in use or not. If some other vTEE (e.g., the system-vTEE) is using the device, the TEE-visor will ask it to release the device. After that, the TEE-visor will assign the device to the requester vTEE. The requester will first re-initialize the device before using it, so that any residual states within the device will not affect the usage. If one vTEE fails to release a device (e.g., return fail or timeout), the TEE-visor will revoke it eventually. In practice, we find that in most cases a vTEE just needs to use a device for a short time, like generating a random number.

Our system supports all the three modes for different scenarios. A third-party vTEE can chose which mode to use for different peripherals and scenarios. In real cases, a small company can use delegation mode to ease its development.

[4] 3D facial recognition is usually considered more secure than 2D and can use the delegation mode.

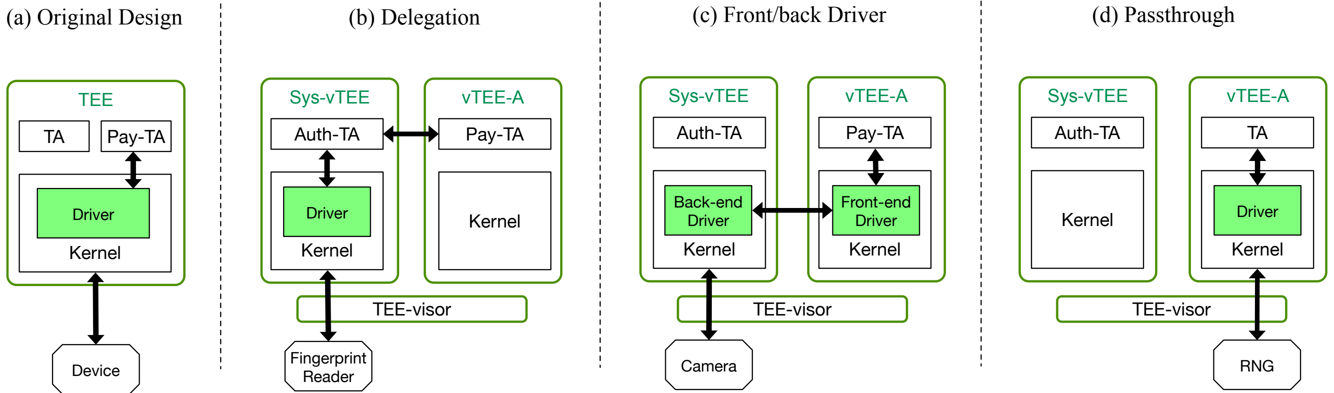

Figure 5. Different I/O device management modes of TEEv. (a) shows the original design where all the devices are controlled by the only TEE. (b) shows the delegation mode in which the system-vTEE controls the device and offers interface for other vTEE to access. (c) shows the front/back driver mode, in which a driver is split into two parts, and the system-vTEE offers a low-level interface to other vTEEs. (d) shows the passthrough mode, in which a vTEE controls the device exclusively using its own device driver.

A security critical TA may chose the frontend/backend or passthrough mode to get full control over the execution. Thus, this is a tradeoff between security and availability/deployability.

5 Security Analysis

5.1 Defending Against Existing Attacks

We show how our system can defend against existing attacks of TEE. Since most of the TEE implementations are not open sourced, especially those have many CVEs, we cannot do evaluation on real devices. However, since our system is expected to be secure by design, we think such analysis can reflect the effectiveness of our design. We select following CVEs which have sufficent details for our analysis.

CVE-2016-5349: According to the description of Boomerang attack [44], there are three TAs that accept user input: KeyMaster, WideVine, and PlayReady. These three TAs all have vulnerabilities that allow a malicious application in the non-secure world to read any memory from an arbitrary address in the non-secure world, including memory of all other applications and the kernel. For example, the malicious application can ask the KeyMaster to read one byte of arbitrary address of the REE and generate a cryptographic signature as a result. To recover the original data, the result is checked against a pre-computed table of signatures for all of 256 possible values of a byte with a known key. By using our system, the TEE instance will be restricted to only access its own memory and the memory shared with it explicitly. Thus the attack will be defeated.

CVE-2016-8762/8763/8764: According to the description of Boomerang attack [44], the TrustZone driver has a flawed validation mechanism for user's input, which enables an attacker to locate both arbitrary read and write functionalities on any address of REE. The attacker then can get the root privilege of REE and do further attacks to enable code execution within the TEE. Although the paper does not have details on the second step (TEE code execution), we can know that by using our system, the attacker will fail on the first step and cannot read/write REE's data through sending out-of-range pointers to the TEE services.

CVE-2016-0825: This attack is done by leveraging Widevine TA's vulnerability. Once got compromised, the attacker will send request to the TEE kernel through the TA to leak data stored in TEE's secure storage. Since the secure storage is used by all the TAs within the TEE, such leakage will also affect other TAs' security, which is known as "fate sharing". By using our system, a critical TA can have its own TEE with a dedicated secure storage to avoid such leakage.

5.2 Isolation Enforcement

The isolation between vTEE and REE, as well as between different vTEEs, are enforced by TEE-visor's exclusive control of MMU.

vTEE Impersonation: If an app is interacting with vTEE-A, it needs to send the ID of the vTEE-A to the TEE-visor to establish a communication channel. However, it is within our threat model that an attacker may change the ID (in the untrusted normal world) to some already compromised vTEE instance, vTEE-E, to impersonate vTEE-A and redirect all the communication. Such attack can only attack the app (which is in the normal world and not protected) but will not cause secret data leakage in vTEE-A.

Compromised vTEE: A compromised vTEE may try to break the isolation of TEE-visor through injecting privileged instructions. Since TEE-visor controls access permissions of all vTEE instance memory, it prevents attacks that attempt to inject unverified code into kernel mode to violate the isolation. Note that currently all TEE products do not support dynamic kernel code generation and thus TEE-visor could easily guarantee the compatibility in this case.

Secure DMA Attacks: Some specific hardware peripherals are able to do Direct Memory Access (DMA) inside the secure memory on many platforms. This feature could be

leveraged by compromised vTEE to read or write arbitrary secure memory and threaten the isolation enforcement of TEEv. To avoid this attack, TEE-visor will restrict any direct manipulation of secure DMA controller. Instead, TEE-visor only allows the front/back driver mode for the secure DMA controller and will carefully verify that the secure DMA request does not violate the isolation scheme of TEE-visor.

Side Channel Attacks: Like other TrustZone based secure systems, TEEv could be potentially vulnerable to side channel attacks [46] [43]. The SMC instruction in ARM synchronizes the pipeline, making speculative execution attacks across normal and secure world transitions impossible. However, a potentially malicious TA inside a premature vTEE may be able to launch speculative execution attacks to other TAs and its vTEE. Currently TEEv could not prevent these attacks inside vTEE. Besides, State of a device in a vTEE not properly reset in passthrough mode may potentially leak private device data to other vTEEs. Nevertheless, the effect of these attacks is limited to leaking information without altering operations or break the isolation of TEEv.

5.3 Discussion on Trust

Traditionally, there is only one TEE which is trusted by default. Now we have multiple vTEE instances, which means that we also need to consider the relationship between them. The system-vTEE is introduced mainly for hosting various device drivers. A 3rd party vTEE should not rely on the trust of system-vTEE. However, if I/O delegation mode is chosen for the fingerprint authentication, a compromised system-vTEE can return false results to other vTEEs, e.g., return "PASS" even if the finterprint does not match. In this case, a more secure way is to use passthrough mode so that the system-vTEE is fully bypassed. However, the 3rd party TEE should be able to control the fingerprint reader directly, which is a non-trivial. A better way is to minimize the system-vTEE to make it only host device drivers. For example, a phone vendor can pre-install two vTEEs: a system-vTEE and a ta-vTEE. The latter is used for dynamically installing new TAs. In this case, the attack surface of the system-vTEE is very narrow which makes it hard to attack.

On the contrary, the phone vendors also do not trust the 3rd party vTEE instances. It should not send user's sensitive information, e.g., fingerprint template, to the untrusted vTEEs, since most of the phone vendors promise to keep user's identity information within the phone, and a 3rd party vTEE may upload these info to some cloud. Our design can support full isolation between vTEEs, so this is considered as a policy issue instead of mechanism one.

TEEv relies on a system-vTEE to provide system wide services, which may be leveraged to launch confused deputy attacks. TEEv mitigates the problem by providing restricted and fixed services, and explicitly configuring shared memory access.

6 Implementation and Evaluation

We have implemented TEEv both on a TrustZone-enabled Samsung Exynos 4412 development board and a Mediatek MT6750 mobile phone, equipped with ARM Cortex-A9 processors (32bit) and Cortex-A53 processors (64bit), respectively. TEE-visor is implemented as an independent secure component running in secure world and interacts with different vTEEs and REE using well-defined APIs. We have ported a commercial TEE product and an open-source TEE running atop of TEE-visor: TrustKernel's T6 [18], which has been deployed in hundreds of millions of devices, and Linaro's OP-TEE [12], which is widely used by researchers. T6 is configured as a system-vTEE and OP-TEE as a ta-vTEE. The chosen vTEEs comply with the GlobalPlatform TEE API specifications, including TUI, crypto and storage. In our implementation, TEE-visor and all vTEEs are built into 32-bit executable applications for compatibility reason. This is because currently almost all TEE products run in ARM32 mode even on 64-bit processors.

6.1 TEE Porting Efforts

Although the two vTEEs have different code base, the porting efforts are quite similar. All modifications are within the original vTEE kernels. The rest of the code, including vTEE user space libraries and vTEE drivers in REE, is not changed. The porting of vTEE kernels can be split into two stages: boot configuration and runtime protection. In the boot configuration stage, vTEE kernels need to delegate the page table management and operation initialization to TEE-visor, including MVBAR, TTBR0, TTBR1, TTBCR, SCTLR, DACR, TLB flush and MMU enable/disable operations. Upon page table initialization of each vTEE, TEE-visor maps code/data pages of entry gates into vTEE's address with write-protection. We found that OP-TEE uses 1MB-sections to map its boot code and data, which is too coarse for memory protection and not well protected. Therefore, we refine its code to use 4KB-page granularity with page-level memory protection enabled. This refinement is necessary to avoid code injection attack inside vTEE kernel, which might expose a hole for compromised vTEE to inject privileged instructions in kernel mode to violate the isolation of TEE-visor and vTEEs. We added or modified about 270 lines of code in T6 and 210 lines of code in OP-TEE. After boot configuration, TEE-visor will lock down the entry gate and enable memory protection for vTEE. For runtime protection, we replaced all sensitive operations in vTEE with an SMC gate call to TEE-visor, which wraps 40 lines of code in T6 and 26 lines of code in OP-TEE. In our experience, it takes within three days porting efforts for a TEE kernel developer to run TEE atop of TEE-visor.

6.2 Implementation Complexity and TCB Size

Our design keeps the TCB of TEEv small to reduce the attack surface. The entry gate of TEEv, which runs in monitor mode

Table 2. Context switch overhead

Invocation Type	Latency (ms)
Ctx switch between TAs from different vTEEs	16.0
Ctx switch between TAs within a system-vTEE	10.6
Ctx switch between TAs within original TEE	6.8
Ctx switch from vTEE to TEE-visor	0.2

Table 3. Secure service invocation

TA Placement	CA-TA Invocation	Latency (ms)
Memory	Original TEE	116.0
Memory	TEEv	232.5
External storage	Original TEE	586.4
External storage	TEEv	745.3

Table 4. TEE boot time latency

TEE Type	Latency (ms)
Original TEE	2439.2
TEEv	5124.0

as a unique entry point to TEE-visor, contains about 350 lines of assembly code. TEE-visor contains about 3800 lines of C and assembly code. For each vTEE, it only needs to trust its own code base and TEE-visor.

6.3 Performance Impact

Context Switch between vTEEs: We measured the execution time of all kinds of context switches on the development board, including context switch from a vTEE to TEE-visor, context switch between TAs within a vTEE and across vTEEs. Table 2 shows the average performance result. We ran the test on the ARMv7 platform for ten times and calculate the average.

Secure Service Invocation: TAs usually provide services for client applications (CA). We measured the secure service invocation latency between a CA and a TA using system-vTEE on the development board, as shown in Table 3. The TA service we tested is a vTEE information query service. During the TA invocation, there are several round trips to TEE-visor, including entry call and parameter mapping. We measured the case when the TA is pre-placed in memory and in external storage. When placed in external storage, loading the TA requires file and crypto verification operations.

TEE Boot Time: TEEv modified the boot procedure of original TEE: in original TEE, bootloader loads and boots TEE, then switches to normal rich OS. In TEEv, early stage bootloader first loads and boots TEE-visor and then boots system-vTEE. After system-vTEE boots up, it switches to the normal rich OS. We measured these two types of latency on the development board, as shown in Table 4. Booting TEE-visor and vTEEs adds 2.7s, which takes a very small fraction (7%) of the total device bootup time (39.09s), and is a one-time effort for each reboot.

Performance Impact on REE: We used Antutu [7] to measure the overall system performance overhead. We firstly disabled TEEv, ran the test with original TEE, and then enabled TEEv. By comparing the results, we found that TEEv incurs nearly zero performance overhead. This is because normally in mobile device, most of the time the processors run in REE, and only run in TEE on demand in a request-response model. During the test, TEE has few tasks to run. This implies TEEv has no impact on performance of existing REE.

7 Real-world Case Study

We use the most popular real-world TAs deployed in more than a billion devices to demostrate how TEEv could be used using a MediaTek mobile phone.

Keymaster and GateKeeper TA: Keymaster TA [6] is a hardware-backed keystore required by Android, which implements various cryptographic functionality to assure the protection of cryptographic keys generated by Android OS and applications. GateKeeper TA [5] performs device pattern/password authentication in TEE for Android. These two TAs are essential for Android security.

Fingerprint TA: Android fingerprint HAL [4] provides stubs for third-party fingerprint vendor to implement fingerprint authentication in TEE.

Alipay TA: Alipay TA is a secure payment TA for Alipay using biometric authentication, which is widely used globally. During a payment transaction, Alipay app will firstly request biometric (e.g., fingerprint) TA in TEE to authenticate the user and then request Alipay TA to sign the payment transaction. Upon receving the payment request, Alipay TA will ask the biometric TA for the last authentication result and sign the transaction using its private key if the authencation passed. The signed response will be sent back to Alipay app, piggyback to remote server to finialize the transaction. In case of using 2D facial to authenticate, Alipay requires to install her own facial recognition algorithm TA and a risk TA to work together to ensure security.

Wechat Pay TA: Wechat Pay TA is for Wechat and its architecture is similar to Alipay TA, except that it reuses functionality provided by keymaster TA and only supports fingerprint authentication based payment.

These five TAs now have been pre-installed in almost all Android devices in China. Here we provide a case study on how to deploy TEEv to protect these existing TAs. As shown in Figure 6, in this case, the phone vendor pre-installs two vTEEs on phone during manufacturing, which is the same process that phone vendors currently follow to pre-install TEEs. The two TEEs are the system-vTEE and the ta-vTEE. The system-vTEE contains all the secure device drivers and offers services, such as fingerprint authentication

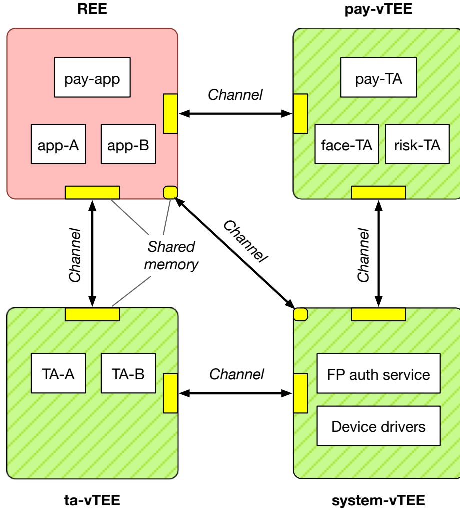

Figure 6. Four running domains: the system-vTEE and ta-vTEE are pre-installed by phone vendor. The pay-vTEE is provided by the payment app vendor. The pay-vTEE and ta-vTEE do not communicate directly.

Table 5. Device and payment authentication performance

Scenario	Ctx Switches	Cost(ms)
Dev unlock using pwd in orig TEE	90	19.8
Dev unlock using pwd in TEEv	90	21.0
Dev unlock using fp in orig TEE	858	178.2
Dev unlock using fp in TEEv	858	211.5
Alipay using fp in orig TEE	937	3130.4
Alipay using fp in TEEv	937	3564.2

and backend drivers, to other vTEE instances. Fingerprint TA, keymaster TA and gatekeeper TA are pre-installed in the system-vTEE. The system-vTEE is fully controlled by the vendor and does not support installation of any 3rd party TAs. The only attack surface exposed by the system-vTEE is the services it provides.

The ta-vTEE is used for installing and running 3rd party TAs. From REE's view, the ta-vTEE is the default TEE on the phone. If a client app in REE (i.e., CA) needs to install or interact with a TA without specifying a vTEE ID, the TEE driver in the REE kernel redirects the requests to the ta-vTEE by default. Wechat TA is installed in this ta-vTEE and leverages the fingerprint authentication service provided by the system-vTEE. Note that for establishing a communication channel, TEEv requires the caller to explicitly specify the memory pages it has allocated for the communication. The TEE-visor then ensures that the callee vTEE can only access these pages.

Now the Alipay app is installed in REE. It has stricter security requirements than that of the ta-vTEE. The payment app asks the TEE-visor to install a new signed vTEE named "pay-vTEE", which is designed for handling security critical operations related to mobile payments. The payment app then installs three TAs in the pay-vTEE: a *pay-TA* for processing payment, a *face-TA* for 2D facial authentication, and a *risk-TA* for risk analysis based on client-side information. During the initialization, the pay-TA generates a pair of private/public keys based on the device certificate and sends the public key to the payment server to bind the TA with user's account. When making a payment, the app asks the pay-TA to authenticate the user. In case of using facial recognition, the pay-TA will invoke the face-TA within the same vTEE. The face-TA then asks the system-vTEE to get user's face data (raw data) through the frontend/backend I/O model. It then runs its own facial recognition algorithm to authenticate the user. Throughout the payment process, the risk-TA collects various device-side information like GPS, time, user input, etc., and run it through a proprietary machine learning model to evaluate the risk of the transaction. The model is retrieved from a remote server and protected from REE and any other vTEEs.

The isolation between REE and the three vTEEs have several benefits. First, the separation of system-vTEE and ta-vTEE allows the former to have a very small code size and thus small attack surface. The system-vTEE does not host any 3rd party TAs. Second, the three TAs in the pay-vTEE are isolated with the TAs in the ta-vTEE. In this case, it only relies on the system-vTEE for fingerprint/face authentication. If the payment app has even higher demand on security, it can use the passthrough I/O mode. In either case, the private key of the pay-TA is secure since no other vTEE can access pay-vTEE's memory and storage. Third, the three vTEEs are restricted to access only the shared memory pages of the REE. These new abilities can significantly improve the security of the entire system.

We have conducted an evaluation on different scenarios: device unlock using password, device unlock using fingerprint, Alipay secure payment using fingerprint. The evaluation was done using the Mediatek mobile phone. We have registered five fingerprint templates before the test. Aliapy TA relies on system-vTEE to provide fingerprint authentication service in this case. We measured the times of context switch to TEE and the overall time cost of the entire operation, as shown in Table 5.

8 Related Work

Virtualization on ARM platform: Previous research on ARM virtualization focused mostly on the REE side, e.g., Xen/ARM [36] and KVM/ARM [30, 33]. vTZ [35] virtualizes the ARM TrustZone from the hardware's perspective. Like TEEv, vTZ also enables running multiple TEEs at the same time. However, unlike TEEv, the purpose of vTZ is to support one secure world for each virtual machine in the normal world for ARM server scenario, so that a virtual machine can use *smc* to switch to its own secure world, while TEEv's

purpose is to support multiple isolated vTEEs on mobile platform. In other words, the difference between vTZ and TEEv is that vTZ virtualizes TrustZone (hardware level) whereas TEEv virtualizes TEE (software level). In vTZ, the behaviors of virtualized secure world are exactly the same as native secure world, which means that TEE still has higher privilege than REE in the same virtual machine, and thus it cannot defend against all the attacks we discussed in the paper. ARM recently proposed s-EL2 on v8.4A, which is not available yet. While the new hardware feature can ease the design of TEEv, TEEv, as a pure software solution, can still benefit billions of existing ARM devices.

Same-privilege Isolation: Same-privilege isolating, also known as "intra-level isolation", has been extensively studied on both x86 [32, 34, 48] and ARM platforms [23, 29]. In our system, we leverage several techniques, such as TLB lockdown, global TLB entries, entry gates etc., to ensure the security and atomicity of context switching between different address spaces without using dedicated page table. Our design also minimizes the modification to existing software stack in the secure world.

Hypervisor-based TEE: There are a number of designs leveraging a hypervisor to provide an isolated execution environment with a small TCB [26, 27, 41, 42, 45, 47, 49, 51–54]. Compared to these designs, TEEv has a different goal and takes a different approach that results in smaller TCB. There are a few mainstream hypervisors ported and optimized for the ARM architecture [13, 20, 30, 40]. However, these hypervisors are not suitable to be used directly in the TrustZone environment.

Other Software-based TEE: There are many types of TEE that are based on software, like Linux kernel [24, 28, 34], or compiler [31, 32], to name a few. These researches are orthogonal to our work. The design of TEEv is inspired from these systems but solves a different problem and faces different challenges.

9 Conclusion

In this paper, we propose vTEE, a virtualization architecture for TEE to support multiple restricted TEE instances (aka., vTEEs). By introducing a tiny layer of hypervisor, vTEE instances from different vendors can run on a single mobile phone and each instance can host its own TAs. Different vTEEs are isolated from each other. The TEE-visor also restricts the capability of vTEEs so that they cannot access arbitrary memory or peripherals as before. Meanwhile, our system also supports controlled interaction between TAs running on different vTEEs, which makes our system more practical and deployable. The evaluation based on real devices shows the effectiveness and efficiency of our system.

Acknowledgments

This work is supported in part by the National Key Research & Development Program (No. 2016YFB1000104), and the National Natural Science Foundation of China (No. 61772335).

References

[1] 2017. Google Project Zero. https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html.

[2] 2017. Introducing 2017s extensions to the Arm Architecture. https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture.

[3] 2018. Alipay Member Protection. https://intl.alipay.com/ihome/user/protect/memberProtect.htm.

[4] 2018. Android Fingerprint HAL. https://source.android.com/security/authentication/fingerprint-hal.

[5] 2018. Android Gatekeeper. https://source.android.com/security/keystore.

[6] 2018. Android Hardware-backed Keystore. https://source.android.com/security/keystore.

[7] 2018. Antutu-benchmark. https://play.google.com/store/apps/details?id=com.google.android.stardroid&hl=en.

[8] 2018. ARM Cortex-A57 MPCore Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/CHDDDHFD.html.

[9] 2018. ARM Cortex-A72 MPCore Processor Technical Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.100095_0001_02_en/way1381846769141.html.

[10] 2018. GlobalPlatform. https://www.globalplatform.org/.

[11] 2018. Google Trusty. https://source.android.com/security/trusty/.

[12] 2018. OP-TEE. https://github.com/OP-TEE/.

[13] 2018. open virtualization. http://www.openvirtualization.org.

[14] 2018. Prove & Run. http://www.provenrun.com/.

[15] 2018. Qualcomm Security. https://www.qualcomm.com/products/snapdragon/security.

[16] 2018. SierraTEE. https://www.sierraware.com/open-source-ARM-TrustZone.html.

[17] 2018. TLB Lockdown Registers. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/Cihjdehg.html.

[18] 2018. TrustKernel T6. https://trustkernel.com.

[19] 2018. Trustonic Inc. https://www.trustonic.com/.

[20] 2018. Xen ARM with Virtualization Extensions. http://xenproject.org.

[21] Tiago Alves and Don Felton. 2004. TrustZone: Integrated hardware and software security. *ARM white paper* 3, 4 (2004), 18–24.

[22] ARM. 2016. Connected devices need e-commerce standard security say cyber security experts. https://goo.gl/1ePiQC.

[23] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 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*. ACM, 90–102.

[24] Ahmed M Azab, Kirk Swidowski, Jia Ma Bhutkar, Wenbo Shen, Ruowen Wang, and Peng Ning. 2016. Skee: A lightweight secure kernel-level execution environment for arm. In *Network & Distributed System Security Symposium (NDSS)*.

[25] Please! Bits. 2016. QSEE privilege escalation vulnerability and exploit (CVE-2015-6639). http://bits-please.blogspot.hk/2016/05/qsee-privilege-escalation-vulnerability.html.

[26] Haibo Chen, Fengzhe Zhang, Cheng Chen, Ziye Yang, Rong Chen, Binyu Zang, Pen-chung Yew, and Wenbo Mao. 2007. Tamper-Resistant Execution in an Untrusted Operating System Using A Virtual Machine Monitor. *Parallel Processing Institute Technical Report* FDUPPITR-2007-08001 (2007).

[27] X. Chen, T. Garfinkel, E.C. Lewis, P. Subrahmanyam, C.A. Waldspurger, D. Boneh, J. Dwoskin, and D.R.K. Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. ASPLOS*. ACM, 2–13.

[28] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained execution units with private memory. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 56–71.

[29] Yeongpil Cho, Donghyun Kown, Hayoon Yi, and Yunheung Paek. 2017. Dynamic Virtual Address Range Adjustment for Intra-Level Privilege Separation on ARM. In *NDSS*.

[30] Jason Nieh Christoffer Dall. 2014. KVM/ARM:The Design and Implementation of Linux ARM Hypervisor. In *ASPLOS*.

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

[32] John Criswell, Nicolas Geoffray, and Vikram S Adve. 2009. Memory Safety for Low-Level Software/Hardware Interactions.. In *USENIX Security Symposium*. 83–100.

[33] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. 2016. ARM virtualization: performance and architectural implications. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 304–316.

[34] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 191–206.

[35] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *USENIX Security*.

[36] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. 2008. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*. IEEE, 257–261.

[37] Apple Inc. 2016. iOS Security Guide. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[38] Huawei Inc. 2016. Built-in TEE chip for enhanced security for your private data. http://phoneproscons.com/716/huawei-honor-magic/352/built-in-tee-chip-for-enhanced-security-for-your-private-data/.

[39] Samsung Inc. 2018. Samsung KNOX. https://www.samsungknox.com/en/knox-platform/knox-security.

[40] J, S Hwang, S Suh, C Heo, J Park, S Ryu, C Park, and Kim. 2008. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *IEEE CCNC*.

[41] Youngjin Kwon, Alan M Dunn, Michael Z Lee, Owen S Hofmann, Yuanzhong Xu, and Emmett Witchel. 2016. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 277–290.

[42] 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.

[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[44] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. (2017).

[45] Nuno Santos, Rodrigo Rodrigues, Krishna P Gummadi, and Stefan Saroiu. 2012. Policy-sealed data: A new abstraction for building trusted cloud services. In *Usenix Security*.

[46] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* 15 (2015).

[47] A. Seshadri, M. Luk, N. Qu, and A. Perrig. 2007. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. SOSP*.

[48] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. 2017. Deconstructing Xen. In *NDSS*.

[49] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 279–292.

[50] TrustKernel. 2018. TrustKernel TEEReady. https://dev.trustkernel.com/ready.

[51] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. 2009. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 545–554.

[52] Jisoo Yang and Kang G Shin. 2008. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 71–80.

[53] 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*. ACM, 203–216.

[54] Fengzhe Zhang, Yijian Huang, Huihong Wang, Haibo Chen, and Binyu Zang. 2008. PALM: security preserving VM live migration for systems with VMM-enforced protection. In *2008 Third Asia-Pacific Trusted Infrastructure Technologies Conference*. IEEE, 9–18.

Appendices

Table 6. Vulnerabilities of TEEs. We select the CVEs that belong to TEE (either TA or TEE kernel). Some of the CVEs (e.g., 2015-4421, 2014-4322 and 2016-3931) are located in the TEE driver of the REE OS, which are not included in this table.

Vendor	Number	Component	Description
Qualcomm	CVE-2014-9932/9933 /9935/9936/9937/9945 /9947/9948/9949/9951	TA or TEE	Qualcomm did not publish details on these vulnerabilities. It only mentioned that they are caused by integer overflow, time-of-check time-of-use race condition, buffer overflow, improper authorization, improper validation of array index , information exposure, untrusted pointer dereference, information exposure through timing discrepancy.
	CVE-2015-6639/6647	TA	The Widevine QSEE TrustZone application allows attackers to gain privileges via a crafted application and execute arbitrary code within TEE. [25]
	CVE-2015-8995/8996 /8997/8998/8999/9000 /9001/9002/9003/9005 /9007	TA or TEE	Qualcomm did not release details on these vulnerabilities. It only mentioned that integer overflow, buffer overflow, time-of-check time-of-use race condition, untrusted pointer dereference, information exposure, out-of-range pointer offset, cryptographic issue, double free, etc.
	CVE-2015-9031	TA or TEE	A memory address of TrustZone can be exposed to the REE OS by HDCP.
	CVE-2016-0825	TA	The Widevine QSEE TrustZone application may leverage TEE kernel to get and leak data stored in TEE's secure storage.
	CVE-2016-2431/2432	TEE	allows attackers to gain privileges via a crafted application, TrustZone Kernel Privilege Escalation
	CVE-2016-5349	TA	When TAs receive memory addresses from REE such as Linux Android, those addresses have previously been verified as belonging to REE memory space rather than QSEE memory space, but they were not verified to be from REE user space rather than kernel space. This lack of verification could lead to privilege escalation within the REE.
	CVE-2016-10237	TA	If shared content protection memory were passed as the secure camera memory buffer by the REE to a TA, the TA would not detect an issue and it would be treated as secure memory.
	CVE-2016-10238	TA or TEE	In QSEE the access control may potentially be bypassed due to a page alignment issue.
	CVE-2016-10239	TA or TEE	In TrustZone access control policy may potentially be bypassed due to improper input validation an integer overflow vulnerability leading to a buffer overflow could potentially occur and a buffer over-read vulnerability could potentially occur.
	CVE-2016-10297	TA or TEE	A Time-of-Check Time-of-Use Race Condition vulnerability could potentially exist.
	CVE-2016-10333	TA or TEE	A sensitive system call of TEE was allowed to be called by the REE OS. No other details.
	CVE-2016-10339	TA or TEE	The REE OS can overwite secure memory or read contents of the keystore within the TEE.
	CVE-2017-0518/0519	TA or TEE	An elevation of privilege vulnerability in the Qualcomm fingerprint sensor driver could enable a local malicious application to execute arbitrary code within the context of the kernel. It first requires compromising a privileged process.
	CVE-2017-18125	TA	The memory of the buffer used by the secure camera may be reused by untrusted world, which may cause secret data leakage.
Hisilicon	CVE-2015-4422	TEE	When executing SMC instruction, a physical address pointed to *TC_NS_SMC_CMD* structure will be sent to TEE. A malformed *TC_NS_SMC_CMD* gives an attacker a chance to write one byte to almost any physical address. As there is no bound-checking, the attacker can modify any physical memory except the memory used by TEE kernel.
	CVE-2016-8762/8763 /8764	TA or TEE	The TrustZone driver has an input validation vulnerability. An attacker can leverage BOOMERANG and other techniques to obtain full root privileges of REE, as well as code execution within the TEE itself.
	CVE-2017-8142	TEE	A use-after-free bug affects the TEE module driver.
OP-TEE	CVE-2016-6129	TEE	The problem lies in the "LibTomCrypt" code in OP-TEE, that neglects to check that the message length is equal to the ASN.1 encoded data length. It makes OP-TEE vulnerable to "Bleichenbacher" signature forgery attack.
	CVE-2017-1000412	TEE	The encryption library is vulnerable to the bellcore attack resulting in compromised private RSA key.
	CVE-2017-1000413	TEE	The TEE is vulnerable a timing attack in the Montgomery parts of libMPA resulting in a compromised private RSA key.
Motorola	CVE-2013-3051	TEE	The kernel of TEE is hacked, which enables an attacker to unlock the bootloader and load any systems to run.
Other	CVE-2017-6296	TEE	Software contains a TOCTOU issue in the DRM application which may lead to the denial of service or possible escalation of privileges. This issue is rated as moderate.
	CVE-2017-6295	TEE	NVIDIA TrustZone Software contains a vulnerability in the Keymaster implementation where the software reads data past the end, or before the beginning, of the intended buffer; and may lead to denial of service or information disclosure.