PartEmu: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation

Lee Harrison[*1], Hayawardh Vijayakumar[*1], Rohan Padhye[2], Koushik Sen[2], and Michael Grace[1]

[1]Samsung Knox, Samsung Research America
{lee.harrison,h.vijayakuma,m1.grace}@samsung.com
[2]EECS Department, University of California, Berkeley
{rohanpadhye,ksen}@cs.berkeley.edu

Abstract

ARM's TrustZone technology is the basis for security of billions of devices worldwide, including Android smartphones and IoT devices. Because TrustZone has access to sensitive information such as cryptographic keys, access to TrustZone has been locked down on real-world devices: only code that is authenticated by a trusted party can run in TrustZone. A side-effect is that TrustZone software cannot be instrumented or monitored. Thus, recent advances in dynamic analysis techniques such as feedback-driven fuzz testing have not been applied to TrustZone software.

To address the above problem, this work builds an emulator that runs four widely-used, real-world TrustZone operating systems (TZOSes) - Qualcomm's QSEE, Trustonic's Kinibi, Samsung's TEEGRIS, and Linaro's OP-TEE - and the trusted applications (TAs) that run on them. The traditional challenge for this approach is that the emulation effort required is often impractical. However, we find that TZOSes depend only on a limited subset of hardware and software components. By carefully choosing a subset of components to emulate, we find we are able to make the effort practical. We implement our emulation on PartEmu, a modular framework we develop on QEMU and PANDA. We show the utility of PartEmu by integrating feedback-driven fuzz-testing using AFL and use it to perform a large-scale study of 194 unique TAs from 12 different Android smartphone vendors and a leading IoT vendor, finding previously unknown vulnerabilities in 48 TAs, several of which are exploitable. We identify patterns of developer mistakes unique to TrustZone development that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. We also demonstrate using PartEmu to test the QSEE TZOS itself, finding crashes in code paths that would not normally be exercised on a real device. Our work shows that dynamic analysis of real-world TrustZone software through emulation is both feasible and beneficial.

[*] These authors contributed equally to this work.

1 Introduction

ARM's TrustZone technology [2] is the basis for security of billions of devices worldwide, including Android smartphones [51,54] and IoT devices [55]. TrustZone provides two isolated environments: a rich execution environment (REE or "normal world") for running normal applications, and a trusted execution environment (TEE or "secure world") for running trusted applications. Only the secure world has access to sensitive data such as cryptographic keys and biometrics information. The secure world runs security-critical "trusted applications" (TAs) for cryptographic key management, attestation [41], device integrity maintenance [4], and authentication on top of a TrustZone operating system (TZOS). It is the responsibility of the TAs and TZOS to protect access to such sensitive data even if the normal world is fully compromised, for example, due to malicious apps or users who "root" their smartphones [63]. A vulnerability in a TA or the TZOS leads to a breakdown of this protection. Therefore, it is critical to be able to analyze the security of TrustZone software.

In spite of TrustZone software's importance to security, dynamic analysis of real-world TrustZone software is limited by TrustZone's locked-down nature. In real-world TrustZone deployments, only code that is authenticated (i.e., signed) by a trusted party can run. This restriction maintains the security of data accessible only by the secure world. However, it comes at a cost: the inability to instrument or monitor code in the secure world. This rules out applying dynamic analysis techniques such as feedback-driven fuzz testing [9,12,40,61], concolic execution [13,48], taint analysis [17,58], or debugging, on TrustZone software on real devices.

As a result, approaches to analyze real-world TrustZone software have been limited. Approaches to find TA vulnerabilities include static reverse-engineering of binaries [7,8] and blind fuzzing without feedback [6] on real devices. Approaches that attempt to emulate software by forwarding requests to real hardware [28,31,49,59] through interfaces such as JTAG or USB are not applicable, since TrustZone hardware does not export such interfaces and its software is locked down. Perhaps closest to our work is TEEMU, mentioned in a talk by Komaromy [30]. While they do not attempt full-system TZOS and TA emulation, they run TAs for a real-world TZOS (an older version of Trustonic's Kinibi [56]) by re-implementing a subset of the TZOS system calls. Since they do not run the original TZOS, this limits TEEMU to testing Kinibi TAs that use only those system calls that they re-implement, and does not allow testing the Kinibi TZOS itself. Furthermore, reproducibility is dependent on the fidelity of re-implementation of the TZOS system calls, which are often complicated.

In this work, we re-host[2] binary images of closed-source, real-world TZOSes in a full-system emulator to enable holistic dynamic analysis of TrustZone software - the TZOSes themselves and the TAs that run on these TZOSes. Specifically, we build an emulator that can run four widespread, real-world TZOSes: Qualcomm's QSEE [38], Trustonic's Kinibi [56], Samsung's TEEGRIS [43], and Linaro's OP-TEE [34][3]. As of 2019, Qualcomm's QSEE runs on more than 60% of Android phones [51, 62], Trustonic's Kinibi runs on over 1.7 billion devices, including 9 of the top 10 Android vendors [54], and Samsung's TEEGRIS runs on several of Samsung's non-Qualcomm smartphones, including its flagship Samsung Galaxy S10 [22, 50], making them the three most widely-used real-world TZOSes.

The obvious challenge for emulation is its practical feasibility. Android smartphones, the biggest users of TrustZone in the real-world, have a huge number of hardware and software components. The naïve approach of attempting to run an entire firmware image by emulating all required hardware is not practically feasible, especially given many vendor-specific, undocumented components. However, many components, such as a hypervisor, are unrelated to the TZOS. Therefore, to make the emulation effort practical, we start by excluding components unrelated to the TZOS.

However, even after excluding such unrelated components, we still find that supporting the remaining components is impractical. For example, the TZOS depends on the bootloader, which itself depends on a variety of storage controllers that are typically extremely complicated, vendor-specific and not sufficiently supported by any open-source emulator. Our insight is that, here, it is more practical to emulate the bootloader's APIs that the TZOS depends on than it is to support the entire, unmodified bootloader binary with all its dependencies. Thus, we re-implement the relevant functionality of the bootloader in a custom component that mimics, or emulates, the original bootloader to the TZOS. Our approach, therefore, is to study TZOS dependencies on each software component and determine whether it is more practical to reuse the original component or emulate it. In this process, we also identify dependency patterns on each component that may help similar future efforts for other closed-source TZOSes.

We implement our design on PartEmu, a modular framework that we built on QEMU [5] and PANDA [17]. We show that both the software and hardware emulation effort required to support these TZOSes is practically feasible: hardware required emulation of a total of 235 distinct registers using 8 access patterns, and additional support for only 3 devices, whereas software emulation of the bootloader and secure monitor required specifying 52 data values and 17 APIs, many again following simple patterns. We show the utility of PartEmu by integrating feedback-driven fuzz-testing using AFL as a module, and use it to test 194 unique TAs from 12 different Android smartphone vendors and a leading IoT vendor, finding previously unknown vulnerabilities in 48 TAs, several of which are exploitable. We identify patterns unique to TrustZone development that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. We also demonstrate using PartEmu to test the QSEE TZOS itself, finding crashes in code paths that would not normally be exercised on a real device.

In summary, the work makes the following contributions.

- We study the software and hardware emulation effort required to run four widespread, real-world TrustZone OSes - Kinibi, QSEE, TEEGRIS, and OP-TEE - in an emulator, showing that the emulation effort is practically feasible if we choose a suitable subset of components to emulate,
- We build PartEmu, a system that enables modular dynamic analysis of TrustZone by addressing additional challenges such as stability, performance, and TA authentication, and
- We use PartEmu to perform a large-scale study of 194 real-world TAs from 12 different smartphone vendors and a leading IoT vendor, finding several previously-unknown vulnerabilities and identifying patterns of causes.

To the best of our knowledge, we are the first to demonstrate that it is practically feasible to re-host real-world closed-source TZOSes in an emulator, and to perform a large-scale dynamic analysis of real-world TAs across Android smartphone vendors.

2 Problem

The problem we address in this paper is that dynamic analysis for real-world, deployed TrustZone software is extremely limited due to TrustZone's necessarily locked-down nature. TrustZone is often the foundation for smartphone security since it has access to critical cryptographic material. For example, it has access to a device-unique symmetric hardware

[2]Firmware re-hosting [23] is the process of migrating firmware from its original hardware environment into a virtual environment.

[3]OP-TEE can already be compiled to run in an emulator. However, we re-host an already-built binary image that runs on real hardware to an emulator.

key [1,41] that is used to ensure that data stored on disk encrypted by that key can only be decrypted on that particular device. As another example, on Samsung phones, TrustZone has access to a factory-installed per-device private key signed by the Samsung CA [42] for remote attestation. Thus, remote servers can verify that they are communicating with a valid, protected, Samsung device, and can decide to store enterprise data on such devices. By convention, on such devices, only authenticated TrustZone software that is signed by a trusted party can run. If arbitrary changes were possible to TrustZone software, then these keys and secrets could be leaked, thus completely compromising security.

A side-effect of the inability to change TrustZone software is that dynamic analysis is extremely limited for the community. Without the ability to instrument or monitor TrustZone software, the community cannot take advantage of advances in dynamic analysis such as feedback-driven fuzz testing [12,40,48,61] for TrustZone software. State-of-the-art for dynamic analysis on devices is limited to projects such as FuzzZone [6], which enables black-box fuzz testing of TrustZone on devices using a custom normal-world Linux kernel driver. Even here, if there is a crash, it is almost impossible to find the root cause. Devices typically just reboot and do not have TrustZone crash logs since such information may leak sensitive data. Researchers have been restricted to primarily static reverse-engineering of binaries [7,8] to find vulnerabilities in TrustZone software.

Thus, we are left with the status quo that TrustZone software, despite being the foundation of security on millions of smartphone devices across the world, has received limited scrutiny from the community.

2.1 Goals

To address the above problem, our goal is to build an emulator to enable dynamic analysis of real-world TrustZone software. In particular, our aim is to re-host closed-source binary images of four widely-deployed real-world TZOSes (and their TAs) in an emulator: Qualcomm's QSEE [38], Trustonic's Kinibi [56], Samsung's TEEGRIS [43], and Linaro's OP-TEE [34]. Specifically, we have the following goals:

- **Compatibility**. The emulator should be able to run the same TZOS and TA binaries that are deployed on real-world devices.
- **Reproducibility**. The emulator should have sufficient fidelity so that the issues discovered should be reproducible on the real device.
- **Feasbility**. We want to require practically feasible hardware and software emulation effort to build the emulator.

3 Challenge and Solution Overview

Our main challenge is that environments that use ARM TrustZone in the real world have a large number of software and

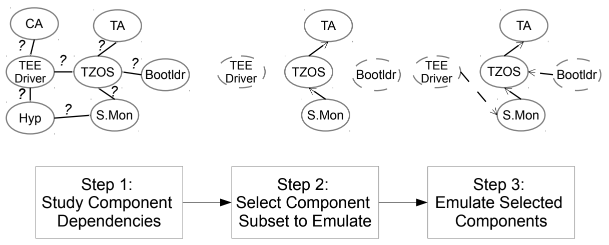
CA
TA
TEE Driver
TZOS
Bootldr
Hyp
S.Mon
?
TEE Driver
TZOS
Bootldr
TA
S.Mon
TEE Driver
TZOS
Bootldr
TA
S.Mon
Step 1: Study Component Dependencies
Step 2: Select Component Subset to Emulate
Step 3: Emulate Selected Components

Figure 1: Solution steps.

hardware components, thus making emulation non-trivial. Android smartphones, the largest users of ARM TrustZone, have software that includes the Android framework, Android apps, the Linux kernel, and a hypervisor in the normal world, and a secure monitor, TZOS, and TAs in the secure world. Hardware includes the system-on-chip, TrustZone address-space and protection controllers, cryptography co-processors, and peripherals such as a touchscreen, camera, GPS, and storage. Naïvely loading the entire firmware binary in an emulator and running it is practically infeasible due to the huge amount of hardware components, many custom and without documentation, that need to be emulated.

To make the emulation effort practically feasible, we first note that the TZOS only depends on a limited subset of all hardware and software components. For example, the TZOS usually has no dependency on the hypervisor. Therefore, such components can be excluded. Second, even for those components that the TZOS depends on, we find that we can sometimes further reduce effort by extracting out only the relevant parts of the component that the TZOS depends on into a stub, depending on how tightly the TZOS is coupled with the component. In general, we can more easily extract dependencies and emulate a component using a stub if the TZOS is loosely coupled with it. Otherwise, it is may be more practical to reuse the original binary component and support its dependencies.

Our solution approach, therefore, has three main steps. We start by studying the dependencies of the target component we want to emulate (Step 1 in Figure 1). In our case, our target, the TZOS depends on the secure monitor, the TEE driver in the Linux kernel, the TEE userspace, and the bootloader. We exclude components that the TZOS does not depend on. For each component the TZOS depends on, we estimate how tightly they are coupled, i.e., how complex the dependency is. Next, using this information, we decide whether to emulate components using a stub or reuse original components (Step 2 in Figure 1). Section 5 describes criteria for choosing whether to reuse or emulate a component, and Section 6 studies component dependencies in our target TZOSes, finding concrete patterns that suggest reuse or emulation.

Third, once we decide which software and hardware components to emulate, we need to emulate them (Step 3 in Figure 1), that is, replace the component with a stub that sufficiently mimics the original component. For most hardware components, we find that the TZOS binary itself gives sufficient information about the expected interaction, such as the result of reading a register. We find that simple register access patterns are sufficient to emulate most hardware (Section 7).

4 TZOS Background

In this section, we first present relevant background on ARM TrustZone (Section 4.1), and then study component dependencies in a typical system running ARM TrustZone (Section 4.2).

4.1 ARM TrustZone Background

ARMv8, ARM's 64-bit architecture that runs the majority of smartphone devices today, has two orthogonal privilege systems (Figure 2). First, it has four privilege levels called exception levels (ELs), similar to rings in x86. Typically, EL0, the lowest privilege level, runs userspace code, EL1 runs the OS, EL2 the hypervisor, and EL3, the highest privilege level, runs the secure monitor. For backwards compatibility, ARMv8 supports running 32-bit code as well. Therefore, it can support both the 64-bit TZOSes (QSEE, TEEGRIS), and 32-bit TZOSes (Kinibi).

Second, ARM TrustZone introduces another orthogonal privilege system. It allows code in any of the exception levels to run in either: (1) a trusted state, called the trusted execution environment (TEE) or "secure world", or (2) in an untrusted state, called the rich execution environment (REE), non-secure or "normal world"[4,5]. Transition from the normal to secure world is done using the secure monitor call (SMC) instruction, which calls into the secure monitor in EL3. SMCs can only be made from EL1 or EL2, and not directly from EL0.

When running in the secure world, software can access all memory and peripherals. When running in the normal world, software can only access non-secure memory and non-secure peripherals. This access control is enforced in hardware by the TrustZone address-space controllers (TZASC) for memory and protection controllers (TZPC) for peripherals.

4.2 TZOS Dependencies

TrustZone software components and their dependencies are implementation-defined. However, we observe that most implementations of TrustZone, including QSEE, Kinibi, TEEGRIS, OP-TEE, and Huawei's TEE [45], have similar software components and interactions. In the secure world, trusted applications (TAs) run in secure EL0 (S.EL0), the TZOS in secure EL1 (S.EL1), and the secure monitor in EL3. In the normal world, applications that communicate with TAs, called client applications (CAs) run in non-secure EL0 (NS.EL0)

[4]As of ARMv8.3, EL2 is only available in the normal world. ARMv8.4 removes this restriction

[5]EL3 runs in both secure and non-secure states

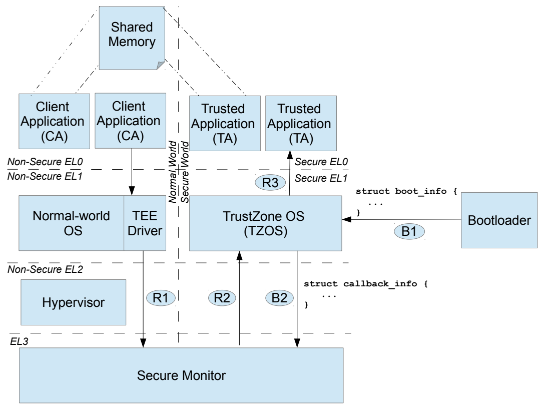

Figure 2: ARMv8 TrustZone architecture and a typical TZOS's software interactions.

alongside other apps. These CAs call into the TEE driver in the OS kernel in NS.EL1 (e.g., Linux) that transitions to the secure world using an SMC. Finally, during system startup, the TZOS is loaded into memory by a bootloader that runs either in EL3 or S.EL1. For ease of implementation, parts of the TEE driver may optionally exist in userspace (NS.EL0); these components form the TEE userspace.

We find that a typical TZOS's dependency on other software components can be broadly divided into dependencies with the bootloader and secure monitor at boot-time, and the secure monitor, and the TEE driver at run-time (Figure 2).

4.2.1 Boot-Time Dependencies

At boot-time, the TZOS depends on the bootloader and secure monitor. The bootloader supplies boot-time arguments to the TZOS through a *boot information* structure (Step B1 in Figure 2). Depending on the boot flow, this information may be passed to the TZOS through the secure monitor. Boot information structures contain hardware information such as the physical address ranges of RAM. The bootloader also sets up any data structures referenced by the boot information, and loads and starts execution of the TZOS binary.

Once the TZOS finishes boot successfully, it passes control back to the secure monitor with specific information about how to call back into the TZOS (Step B2). Thereafter, at runtime, if the secure monitor receives an SMC from the normal world that should be handled by the TZOS, it uses this information to pass control to the TZOS.

4.2.2 Run-Time Dependencies

At run-time, the TZOS typically interacts with the TEE driver in the normal world OS, the secure monitor, and TAs.

The main TZOS flow at run-time is to handle a request from the normal world - originating from a client application (CA) or the normal world OS itself. The normal world TEE driver invokes an SMC instruction to call the secure monitor (Step R1). The secure monitor determines if the request has to be handled by the TZOS. If so, it passes control to the TZOS (Step R2). The TZOS, in turn, handles the request. If the request is to be handled by a TA, the TZOS passes control to the TA (Step R3).

A special case is loading a TA. The TEE driver uses the SMC in Step R2 to send the TA binary and shared memory to communicate with the TA. The TZOS loads the TA and maps the shared memory into the TA.

4.2.3 Hardware Dependencies

The TZOS and secure monitor typically depend on hardware components that configure access control, such as the TZASC and TZPC, to set up secure and normal memory and interrupts. They also depend on the cryptography co-processor, which usually has access to a device-unique hardware key that is only available to TrustZone. One of either the TZOS or the secure monitor interacts with most hardware components; this is implementation-dependent.

5 Selecting Components to Emulate

In this section, we describe how we select a subset of components to emulate with the aim of making the emulation effort practically feasible. As noted, naïvely loading and running entire firmware images requires emulation of a huge number of hardware components, many custom and without documentation, thus making this approach infeasible.

To make the emulation effort practically feasible, our first insight is that TZOSes only depend only on a limited subset of all hardware and software components. For example, the TZOS and TAs have no dependency on the hypervisor or most of the Android framework. Therefore, such components can be excluded, reducing emulation effort. However, we find that even the emulation effort required to support the remaining components is often impractical. For example, supporting the bootloader requires emulation of particular storage controllers [26, 27] that are extremely complicated, and that no open-source emulators we know of support sufficiently. Thus, we need a different approach in such cases.

For a software component the TZOS depends on, we find we can sometimes make emulation more feasible by re-implementing only the relevant parts of the component in a stub. For example, the bootloader has several functions: it reads the TZOS binary image from storage, loads it into memory, sets up arguments, and jumps to the TZOS. However, the TZOS only directly depends on the bootloader setting up the arguments to the TZOS and jumping to it; if we can sufficiently mimic, i.e., emulate, this necessary functionality of the bootloader in a stub, we eliminate the need to support the entire original bootloader, and consequently, to emulate the storage controller. This approach is analogous to QEMU's

Component Type	Prefer to Emulate Component C if	Prefer to Reuse Component C if
Software	*C* and target component are loosely coupled	*C* and target component are tightly coupled
	C and other components are tightly coupled	*C* and other components are loosely coupled
	C is partially or fully open-source	*C* is closed-source
	C is encrypted	*C* is not encrypted
Hardware	*C* does not have interfaces to modify registers/memory	*C* has interfaces to modify registers/memory (e.g., JTAG)
	C locks down software that runs on it (e.g., using secure boot)	*C* does not lock down software that runs on it

Table 1: Criteria to decide whether to reuse or emulate a component C. As in object-oriented design, we use "loosely-coupled" to mean components that have well-defined interfaces with each other and work largely independently of each other, and "tightly-coupled" to mean the opposite, that is, components that need to know each others' internal data structure implementations, leading to complicated interfaces and deep dependencies.

user mode [5], which emulates an OS by re-implementing system call APIs required by a target user application. In contrast, QEMU's system mode runs entire OS binaries and instead emulates the hardware the OS depends on.

However, emulating the required software component APIs is not always more practically feasible. Sometimes, the coupling between two components is so tight that it is often more effort to understand and emulate the required dependencies than it is to reuse the original component and support all its dependencies instead. This is especially true because most TrustZone software is closed-source, and often the only way to determine dependency details is by high-effort binary reverse-engineering. Intuitively, it is preferable to emulate a software component if it is loosely coupled with the TZOS, but tightly coupled with other components itself. Sometimes, a component is tightly coupled with both the TZOS and other components. Our approach identifies that this would require significant emulation effort whether we emulate or reuse the component. Further, source-code availability makes understanding dependencies, and therefore emulation, easier. In addition, if a software component binary is encrypted, then the only option is emulating it. Thus, we have two choices for each component. First, we can *reuse* the original component as it is. Second, we can mimic, or *emulate* the component, that is, replace the component with a model or stub that sufficiently mimics the original component to the target.

For hardware components, we can reuse the original hardware component on the device if the component exposes an interface to interact with memory and registers (e.g., JTAG [47]), or if it is possible to run a custom software proxy on the device that allows a similar interface to hardware access. However, for real-world TrustZone environments, neither of these approaches is possible, since the hardware does not expose such interfaces, and is it not possible to run a custom software proxy for hardware access in the TrustZone because of mechanisms such as secure boot and code signing. Therefore, emulating any required hardware is the only possibility. Table 1 lists criteria to decide whether to reuse or emulate a hardware or software component.

6 Case Studies

In this section, we present results from our study of components that the TZOSes under consideration (QSEE, Kinibi, TEEGRIS, and OP-TEE) depend on, and use the criteria in Table 1 to determine whether to reuse or emulate each component. While the definitions of tight and loose coupling are subjective as in object-oriented design, we identify concrete patterns that indicate tight or loose coupling between components the TZOS depends on. We believe these findings and patterns would help focus and guide similar future efforts for other closed-source TZOSes.

6.1 Bootloader

Bootloader and TZOS Coupling. In all our cases, we found that the bootloader had three well-defined, loosely-coupled functionality relevant to the TZOS. First, the bootloader set up the *boot information* structure with boot-time arguments for the TZOS. This structure usually contains hardware information such as the physical address ranges of RAM and required peripherals. Second, the bootloader loaded the TZOS into memory. Third, the bootloader handed over execution control to the TZOS.

Bootloader and Other Component Coupling. All bootloaders we studied were tightly coupled with a hardware component - the storage controller (e.g., eMMC [26], UFS [27]). Since bootloaders have to read the TZOS and other images from storage, this dependency is expected. However, emulating such hardware faithfully is extremely complicated, and often requires supporting vendor-specific extensions.

Guiding Pattern. Check if the emulator already emulates the storage hardware that the bootloader uses. If so, reusing the bootloader binary is possible. Otherwise, it is preferable to the emulate the bootloader, as the coupling between the bootloader and the TZOS is generally much looser than the coupling between the bootloader and storage hardware.

6.2 Secure Monitor

Secure Monitor and TZOS Coupling. In general, the TZOS interacts with the secure monitor for two functions. First, the TZOS relies on the secure monitor for world switches - to yield control back to the normal world and to upcall into the TZOS. Second, the secure monitor offers APIs to hardware for the TZOS. It is usually the secure monitor, and not the TZOS, that interacts with hardware directly because the secure monitor is developed by the chip hardware manufacturers.

First, TEEGRIS's secure monitor was encrypted with a key derived from hardware. Therefore, our only option was to emulate its secure monitor by reverse-engineering the TZOS itself to find dependencies on SMC APIs. Second, in Kinibi's case, we found that only a limited number (5) of loosely coupled, well-defined SMC API calls between the TZOS and the secure monitor were required to get it to boot and run. These API calls either interact with hardware, store vectors of callback functions for upcalls, or yield control to the normal world. Third, QSEE's interaction with its secure monitor, however, was much more tightly coupled, involving multiple SMC calls and shared data structures that were challenging to reverse-engineer. We suppose this is because QSEE and its secure monitor are both developed by a single entity: Qualcomm. Likewise, our 32-bit OP-TEE's secure monitor and TZOS were compiled together into one binary, which we could not decouple.

Secure Monitor and Other Components Coupling. Kinibi's secure monitor was tightly coupled with hardware. It interfaced with hardware components such as a vendor-specific crypto co-processor and PRNG, which were challenging to emulate. In contrast, QSEE's secure monitor was loosely coupled with hardware; QSEE itself accessed most hardware directly, and did not go through secure monitor APIs. Again, we suppose this is because QSEE and the hardware are both developed by the same entity.

Guiding Pattern. Check if the TZOS and secure monitor are designed such that only the secure monitor directly interacts with most hardware. If this is the case, then it is typically more practical to emulate the monitor's APIs that the TZOS uses to access hardware than it is to emulate the hardware that the secure monitor depends on.

6.3 TEE Driver and TEE Userspace

TEE Driver and TZOS Coupling. The TEE driver in the normal-world OS (usually, Linux) enables communication between CAs and TAs. Broadly, the TEE driver interacts with the TZOS to: (i) start new TAs, (ii) set up shared memory between the CA and TA, (iii) send commands from the CA to the TA, and (iv) respond to requests from the TZOS, such as access to the normal-world filesystem.

We observed two designs of the interaction between TEE driver and the TZOS - synchronous and asynchronous - that gave a broad indication of the extent of coupling. In a synchronous design, the TEE driver specifies its request as arguments to an SMC call and blocks until the secure world completes the request. In an asynchronous design, the TEE driver and the TZOS set up a shared request-response queue that they operate in a producer-consumer relationship. Here, an SMC (or secure interrupt) is used to periodically transfer control to the TZOS. While not necessary, we observed that the asynchronous design generally correlated with tighter coupling because of queue synchronization requirements and because data structures in the queue needed to be consistent between the TZOS and TEE driver. In our case, QSEE and

OP-TEE followed a synchronous design, whereas Kinibi and TEEGRIS followed an asynchronous design.

TEE Driver and Other Components Coupling. The TEE driver optionally depends on the TEE userspace to handle functionality such as reading a file from the filesystem (to load persistent objects through the API `TEE_OpenPersistentObject` or to load TAs), and accessing the RPMB. In our environment, we found that neither QSEE nor OP-TEE required upcalls to the TEE userspace[6], whereas TEEGRIS and Kinibi did.

TEE Userspace. Kinibi, TEEGRIS, and QSEE images were extracted from Android smartphones in which userspace binaries were compiled for Android. Given the well-defined functionality expected of the TEE userspace, we found it much easier to emulate this functionality instead of introducing Android emulation to reuse the TEE userspace binaries.

Guiding Pattern. Check if the TEE driver interacts with the TZOS in an asynchronous manner, or if the TEE driver depends on the TEE userspace to handle significant functionality. In either of these cases, it is usually easier to reuse the TEE driver.

Table 7 in Appendix A.1 summarizes our choices for each component across all TZOSes.

7 Hardware Emulation

The TZOS depends on only a limited subset of all hardware components on a real device, instead relying on the normal world to interact with most hardware directly. This is a typical design choice to keep TZOS code as minimal as possible and not increase the trusted computing base with complicated hardware drivers. For example, to store data on the disk, the TZOS cryptographically "wraps" data using a key accessible only to the TrustZone secure world, and then sends it back to the normal world to store on the disk. This reduces the amount of hardware emulation required, since we do not need hardware models for such devices.

7.1 Ease of Hardware Emulation

For the hardware we need to emulate, we have a key finding: to get the TZOS to boot up and run in the emulator, we needed to emulate only simple access patterns for most hardware it interacts with. The TZOS interacts with hardware using memory-mapped I/O (MMIO), where hardware registers are accessed using memory addresses. We describe the patterns that the TZOS uses to interact with MMIO registers below.

- **Constant Read**. These MMIO registers return a constant value.

[6]OP-TEE generally requires upcalls to load a TA, but in our environment, the TAs were packaged into the OP-TEE binary itself.

```
# Constant read (CONSTANT_READ_REG)
v = read(CONSTANT_READ_REG);
if (v != VALID_VALUE)
    fail();

# Read-write (READ_WRITE_REG)
write(READ_WRITE_REG, v1);
v2 = read(READ_WRITE_REG);
if (v2 != v1)
    fail();

# Increment (INCR_REG)
v = read(INCR_REG);
if (read(INCR_REG) < v)
    fail();

# Poll (POLL_REG)
while (read(POLL_REG) != READY);
```

```
# Random (RAND_REG)
v1 = read(RAND_REG)
v2 = read(RAND_REG)
if (v1 == v2)
    fail();

# Shadow (SHADOW_REG1, SHADOW_REG2)
# Commit (COMMIT_REG)
# Target (TARGET_REG1, TARGET_REG2)
write(SHADOW_REG1, v1)
write(SHADOW_REG2, v2)
write(COMMIT_REG, COMMIT_VALUE)
v3 = read(TARGET_REG1)
v4 = read(TARGET_REG2)
if ((v1 != v3) or (v2 != v4))
    fail();
```

Figure 3: Register patterns we found in the TZOSes binaries. Variables in the binary are in lower case, and hard-coded constants in the binary are in upper case. Registers are identified by their MMIO addresses (e.g., RAND_REG).

- **Write-Read**. These MMIO registers store the value on a write operation and return the most recently written value on a read. This is the behavior of normal RAM.

- **Increment**. These MMIO registers return a monotonically increasing value each time (e.g., a timer). We found that the exact increment did not matter as long as it was non-zero.

- **Random**. These MMIO registers return a random value (e.g., a pseudo-random number generator).

- **Poll**. These MMIO registers are set when a particular operation is complete.

- **Shadow, Commit, and Target**. Shadow registers are used for atomic updates of multiple target registers. Shadow registers store new values to be written to other target registers. When a commit register is written to, all target registers atomically get the value in the corresponding shadow registers. For example, this is used when updating address range registers for access control in the TZASC or TZPC. Otherwise, there might be a tiny window during update where address ranges are configured incorrectly.

Figure 3 lists the corresponding code patterns. Importantly, we observe that the TZOS binary gives us sufficient information to determine both the address and expected values of particular MMIO registers. Given the simplicity of these patterns, we believe that it is possible to automate extracting relevant values from most, if not all, of these patterns.

Locating MMIO Regions. For Kinibi, we control the MMIO region through the boot information structure defined in our emulated bootloader. QSEE, TEEGRIS, and OP-TEE assume specific regions to be MMIO. For QSEE, we deduced these regions from their page tables in the binary. We assume that any region corresponding to a page table entry that has the non-cacheable attribute is MMIO. For TEEGRIS and OP-TEE, we obtained MMIO regions using the device tree used by the Linux kernel.

Other Hardware. Beyond these simple register patterns, the TZOSes required more complex emulation for only three more devices, and we were able to re-use standard implementations in all cases. First, all TZOSes required the ARM standard global interrupt controller (GIC). This hardware is standard and is already emulated in QEMU. Second, QSEE required limited emulation of cryptography hardware. QSEE uses a crypto co-processor, for example, to generate a hash of the TA binary for authentication before loading. Furthermore, it expects the hash of the root certificate signing the TA to be present in a specific memory location [39]. We discuss this in detail in Section 8.2. In particular, we only needed to implement the standard SHA-2 hash algorithm. All other TZOSes used software cryptography. Finally, TEEGRIS required a standard real-time clock (RTC), which was again already implemented in QEMU.

Interrupts. All TZOSes used the ARM-standard global interrupt controller (either GICv2 or GICv3), both of which are supported by QEMU. We did not have to add anything beyond these devices to handle interrupts.

8 PartEmu Implementation

We implemented our design on PartEmu, a framework that we built on QEMU [5] and PANDA [17]. We chose QEMU because it already has support for TrustZone. PANDA gives us an extensible and modular framework with already implemented modules such as taint analysis.

PartEmu adds to PANDA a *run management* API to unify the process of dynamic analysis (Table 2). The API is meant to be invoked by "driver" programs running in the emulator. One or more backend modules can register to receive callbacks when the driver calls into the API. This API is implemented using semihosting calls that call directly into QEMU. We have currently implemented two modules on this run management API: fuzz testing with AFL, and an LLVM run module that outputs an LLVM IR representation of a run of the target. This output could be fed to symbolic analysis engines such as KLEE, as in S2E [10, 13].

8.1 AFL PartEmu Module

We integrate feedback-driven fuzz testing using AFL [61] as a module to PartEmu. We base our code on TriforceAFL [24], which adds AFL support to QEMU system emulation and support for CPU and memory state duplication (forking), with one important difference. In [24], AFL runs QEMU as it does any normal process under test. In contrast, we start QEMU separately and interact with AFL through a proxy that behaves to AFL like the process under test. Thus, we are able to keep our modular structure, and allow AFL to be one among many backend modules for PartEmu's run management API.

Our implementation addresses some additional challenges. First, we need to identify the target being tested. For example, we might want to collect coverage feedback information from a particular TA. However, there are many components executing in TrustZone - other TAs, the TZOS, and the secure monitor. How do we identify our target TA so that we collect only the target's coverage information? Second, we need to ensure *stability*, i.e., that the same input to a component in a particular state results in the same output. This is non-trivial in full-system emulation with randomness and interrupts.

Depending on the TZOS implementation, we determined two different methods to identify the target under test. First, we found that Kinibi and TEEGRIS switched the address-space identifier (ASID) in the `TTBR0_EL1` register when they context-switched between TAs. While Kinibi returned the ASID to the normal-world CA as part of the TA descriptor, TEEGRIS used monotonically-increasing ASIDs. Thus, in both cases, we were able to determine the exact ASID to monitor and it to identify the target. Second, in contrast, neither QSEE nor the version of OP-TEE we ported changed the ASID when switching between TAs. However, we found we could identify the target using address ranges. Before loading a TA, QSEE requires the normal world to donate a region of physical address to load the TA. OP-TEE hardcodes such a region in its binary. Thus, for QSEE and OP-TEE, we identify the target if the program counter falls within this region. Once we identify that a particular basic block belongs to the target TA, we use the block's virtual address to populate AFL's coverage map. Selectively populating the coverage map using only the target's basic blocks can be viewed as an instance of domain-specific fuzzing [36].

Stability is another challenge. AFL defines stability as the property that a target returns the same feedback coverage when fed the same input [60]. We identified four sources of instability: interrupts, statefulness, randomness, and QEMU optimizations. First, interrupts cause different program paths to be executed. We handle this by simply disabling interrupts to the secure world during a run. Second, prior inputs to a stateful target program may drive it to a state where it responds differently to the same input. We solve this issue by forking PartEmu just before starting the test, which forks the entire CPU and memory state.

Randomness is another source of instability. Kinibi, TEEGRIS, and OP-TEE call into the secure monitor to obtain randomness, whereas QSEE accesses hardware PRNG using MMIO registers. We simply return a constant in response to these calls. Finally, the QEMU optimization of translation-block chaining [5] affects stability. When two or more basic blocks always occur only in the same sequence, QEMU chains them together into effectively one translation block. Therefore, if we track each translation block for coverage, we will miss these chained blocks. A simple way to solve this issue to

API	Description
`partemu_run_init(id, buffer)`	Register a client with id and buffer for input
`partemu_run_monitor_asid(id, asid)`	Identify target to monitor with asid
`partemu_run_monitor_addr_range(id, range)`	Identify target to monitor with address range
`partemu_fork()`	Fork a QEMU instance with the same CPU and memory state
`partemu_run_read_input(id)`	Read input from `partemu` module (e.g., AFL) into registered buffer
`partemu_run_start(id)`	Signal run start; `partemu` module starts monitoring target
`partemu_run_stop(id, ret)`	Signal run stop with ret value (e.g., crash) ; `partemu` module stops monitoring target
`partemu_exit(ret)`	Exit forked QEMU child with ret value
`partemu_run_debug(id, ret)`	Pause QEMU and wait for debugger when the target runs next

Table 2: PARTEMU Run Management API.

is to disable chaining, but we found that this reduced performance significantly. Instead, just before blocks are chained, we add an inline QEMU IR callback at the end of each block to the function that records the block. Therefore, blocks can still be chained but will call into our function inline.

8.2 TA Authentication

TAs have to pass two TZOS checks before they are loaded: (i) a signature check and (ii) a version check to prevent rollback. We describe below how we handle these checks for our TZOSes.

To pass QSEE's TA authentication [39] checks, we required additional hardware emulation. QSEE TAs contain a signature and the corresponding certificate chain. QSEE checks that the hash of the certificate matches what is stored in a specific memory area. On the device, this memory area is backed by one-time-programmable fuses that are programmed during device manufacture by the vendor. We faced the challenge of obtaining this value. This value can be either read directly from a real device or parsed from the TA binary. Due to our inability to modify QSEE, we could not extract this value directly from a device; neither would such an approach scale to devices from multiple vendors. Instead, we extracted this value by parsing the root certificate from the TA binary itself. Kinibi, OP-TEE, and TEEGRIS TA authentication, on the other hand, worked out-of-the-box. They had hardcoded public keys in the TZOS binary that it used to authenticate TA signatures.

Our next challenge was overcoming rollback prevention checks. When TA vulnerabilities are patched, TA version is increased. The minimum acceptable TA version is typically stored in secure storage (RPMB). We experimented with two different approaches to overcome this check. First, for QSEE and Kinibi, we re-signed the TA binary with a version number of zero using our own signing key. For Kinibi, we injected this signing key into the binary. For QSEE, we set the OTP-fuse memory area with the hash of this signing key. Second, for TEEGRIS, we emulated the RPMB interface so that it effectively returned zero as the minimum acceptable TA version. Finally, the version of OP-TEE we had did not enforce rollback checks.

In addition to passing rollback prevention checks, the ability to sign TA binaries gives other advantages. First, we can write and sign own custom TAs for testing. Second, it allows us to test TAs across multiple firmware images and vendors using the same TZOS image. Third, it allows us to instrument TA binaries for particular purposes, such as for performance optimizations.

8.3 Performance Optimizations

TA request processing loops are a potential source of inefficiency for testing. TA request processing passes through a lot of components - starting from the CA, to the Linux kernel, the secure monitor, the TZOS, the TA, and back. A shorter loop would enable TA fuzz testing to run much faster.

We found that the TA request processing loop for Kinibi could be optimized across all TAs. TAs in Kinibi have an infinite loop where they wait for a message from the normal world, process it, and return to the normal world [18]. Waiting for and returning to the normal world passes through a common library that we were able to instrument to call into PARTEMU to start and stop a test run, respectively. Thus, we were able to entirely cut out all non-TA components from the request processing loop, speeding up AFL's executions per second by $5\times$.

The TA request processing loop for TEEGRIS, OP-TEE, and QSEE TAs, however, was different, and could not be easily optimized without symbols in the TA binaries. In contrast to Kinibi, these TAs expect the OS to callback into a particular function to handle a request (e.g., the GlobalPlatform TEE Internal API [21] uses `TA_InvokeCommandEntryPoint`). While we could have instrumented the beginning and end of this function to indicate the start and stop of a run, finding the location of this function per-TA from the TA binaries we had was non-trivial in the absence of symbols.

9 Evaluation

In this section, we: (1) quantify the hardware and software emulation required to run the TZOSes, showing that it is prac-

Category	Difficulty	K	Q	T	O
Emulated Boot Information Structure					
Constants	Low	13	8	2	3
Any value	Low	1	3	0	0
Simple value	Low	2	1	14	2
Complex values	High	2	1[note a]	0	0
Total	-	**18**	**13**	**16**	**5**
Emulated Secure Monitor Calls[note b]					
Return simple value	Low	0	-	3	-
Return constant	Low	1	-	5	-
Store/retrieve values	Low	1	-	2	-
Control transfer	High	3	-	2	-
Total	-	**5**	**-**	**12**	**-**

Table 3: Table categorizing the number and difficulty of data fields in the emulated boot information structure, and of emulated secure monitor calls, for Kinibi (K), QSEE (Q), TEEGRIS (T), and OP-TEE (O).
[note a] To construct this complex value, we were able to use an open-source implementation [29].
[note b] Since we reused the secure monitor for QSEE and OP-TEE, we did not need to emulate them.

tical and feasible, (2) demonstrate the utility of emulation through the use-cases of finding real-world TrustZone vulnerabilities using AFL, and (3) evaluate the reproducibility of results found by emulation on a real device.

9.1 Extent of Emulation Required

In this section, we quantify the extent of software and hardware emulation we required to boot up and run the TZOSes. Our targets for emulation were QSEE v4.0, Kinibi v400A, TEEGRIS v3.1, and 32-bit OP-TEE based on v3.1.0. We obtained QSEE, Kinibi, and TEEGRIS binaries from Android firmware images, and OP-TEE from a leading IoT manufacturer's firmware image. Despite these TZOSes being full-fledged and real-world, by following our approach to select components to emulate, we found that the software and hardware emulation required was practically feasible. Across all these TZOSes, to emulate the required software components, we only needed to specify 52 data fields, many simple to determine, and implement 17 SMCs, many again following simple patterns. Hardware components required emulation of only 235 MMIO registers in 8 patterns (Section 7.1), and more precise emulation of 3 additional devices. In many cases, we were even able to re-use open-source components.

9.1.1 Software Emulation

Table 3 quantifies the amount and difficulty of software emulation required for the bootloader and secure monitor. First, we had to emulate the boot information structure passed in by the bootloader sufficiently to boot up and run the TZOSes. Table 3 categorizes the fields of this structure based on how difficult it was to determine their value. In summary, we only needed to specify 52 data fields, 49 of which were straightforward to determine. First, a majority of these values were constants that we obtained directly from the corresponding bootloader binaries. Second, some values did not matter - any value worked. Third, some values were not hardcoded constants but were straightforward to specify - the extent of RAM and the location of the normal-world software to transfer control to. Finally, the most challenging were complex data structures that the bootloader needed to setup. For Kinibi, we needed to setup page tables for a structure describing shared memory between the TZOS and the secure monitor. For QSEE, we needed to setup the SMEM data structure, which describes hardware such as RAM [32]. This task was simplified by the open-source version available in the Little Kernel project [29].

Second, for Kinibi and TEEGRIS, we had to emulate 17 calls to the secure monitor. Again, we found most of these values to be simple values (addresses), constants that we obtained from the binaries, or values from the TZOS that simply needed to be stored on one call and returned on the other. Most challenging to emulate were calls to either yield control to the normal world or to store and use TZOS callback vectors; these required careful saving and restoring of register contexts. For QSEE and OP-TEE, we did not have to emulate the secure monitor since we reuse the original secure monitor.

Register Type	Total (QSEE)	Unique (QSEE)	OP-TEE
Constant Read	478	219	3
Increment	1	1	0
Random	1	1	0
Poll	2	1	0
Shadow	54	4	0
Target	54	4	0
Commit	27	2	0
Total	**617**	**232**	**3**

Table 4: Table showing the total and unique number of types of registers we had to emulate to boot up and run QSEE and OP-TEE. We set the MMIO region to be write-read by default and initialized it to zero values unless otherwise specified.

9.1.2 Hardware Emulation

QSEE required the most hardware emulation. QSEE runs only on phones with a Qualcomm chipset, and hence expects certain hardware components to exist. In contrast, the other TZOSes may run on a variety of devices, and makes few assumptions about hardware. Instead, it is the secure monitor that interacts directly with most hardware.

Table 4 shows the number of types of registers we had to emulate to boot up and run QSEE categorized by the access patterns in Section 7.1. By default, we: (i) set the MMIO region to behave like normal RAM (write-read) so that a read gets the most recent value written, and (ii) initialize the MMIO region to return zeros unless otherwise specified. In total, there were 617 distinct MMIO addresses that required emulation beyond these defaults. One observation further simplified the emulation required. QSEE accessed certain MMIO regions in the same way across different iterations of a loop. We believe that these regions correspond to multiple instances of the same hardware components. We were able to repeat the same emulation for these regions. Discounting these duplicates, we get only 232 unique MMIO registers that we needed to emulate. For OP-TEE, we needed to emulate only 3 MMIO registers.

Table 8 in Appendix A.2 quantifies the amount of code added or modified for PARTEMU's core and for emulated software and hardware components across all TZOSes. In total, we had to add or modify around 14.5K lines of code.

9.1.3 Effort to Support TZOS Upgrades

We found that the upgrades we did only required incremental modifications, and that we were able to re-use most of our work for the previous version. In general, if there are drastic changes to hardware or software components, we would need to re-examine dependencies for the changed component. However, we find that such significant changes are rare for components that the TZOS depends on; they are more common for normal-world components. For Kinibi, we upgraded from version 310B to 400A; the only component we needed to change was the TEE driver. For QSEE, we upgraded between minor versions, and we only needed to add support for 3 additional MMIO registers.

9.2 Use Case: Fuzz Testing TAs

We collected TA binaries from 16 images across 12 leading Android smartphone vendors - Asus, Google, HTC, LG, LeEco, Motorola, Nokia, OnePlus, Razer, Samsung, Sony, and Xiaomi, and a leading IoT vendor. These are represented by Images A to P in a random order in Table 5. These devices run one of QSEE v4.0, Kinibi v400A, TEEGRIS v3.1, or OP-TEE v3.1.0 as the TZOS. In total, we collected 273 TAs. From their names, these TAs appear to encompass a wide variety of functionality such as key management, authentication, maintaining device state for purposes such as attestation, and monitoring device integrity. We found that several TAs were common among images from different vendors. These TAs either come bundled with the TZOS image itself, or are drivers for shared hardware such as fingerprint readers. After de-duplication, we obtained 194 unique TAs.

TAs should protect themselves even if the normal world is compromised. Consistent with this threat model, we wrote simple normal world driver programs to fuzz test TAs. These programs interact with the PARTEMU AFL module using the API in Table 2. They run as a Linux kernel driver (TEEGRIS), in userspace, or as a normal-world stub. The programs request the TZOS to load a TA and set up shared memory, then fuzz inputs to the TA, and finally yield control to the TA through an SMC. For Kinibi and QSEE, we set the contents of the shared memory using fuzzed input from AFL. For OP-TEE and TEEGRIS, which use the GlobalPlatform TEE Client API [20][7], we use the first few bytes of AFL's input to select the type of the 4 parameters - either a buffer or a value - and the command, which is a 32-bit value. We then use the rest of the input to determine the contents of the parameters. Crashes are detected using return values from the TZOS; all TZOSes indicate through specific return values that a TA has crashed.

Image	**Build Date**	**# TAs**	**# Crashing**	**# C**	**# I**	**# A**
A	Dec 2017	13	1	0	0	1
B	Jan 2019	3	0	0	0	0
C	Nov 2018	9	3	0	1	2
D	Dec 2018	15	3	2	0	1
E	Mar 2018	17	4	0	0	4
F	May 2018	13	0	0	0	0
G	Aug 2018	14	2	2	0	0
H	Sep 2018	22	4	2	0	2
I	Oct 2018	44	7	2	0	5
J	Oct 2018	11	2	2	0	0
K	Nov 2018	4	0	0	0	0
L	Oct 2018	38	12	1	4	7
M	Jun 2018	26	8	2	4	2
N	Sep 2018	24	5	2	2	1
O	Mar 2019	22	5	2	1	2
P	Mar 2019	2	0	0	0	0
Total		273	56	17	12	27
Unique		**194**	**48**	**9**	**12**	**27**

Table 5: Number of vulnerabilities found by image, categorized as affecting TA confidentiality (**C**), integrity (**I**), or availability (**A**). We ran AFL in non-deterministic mode on each TA for a total of 5 million executions or until we found a crash, whichever was earlier. We did not seed AFL with any meaningful input.

Table 5 shows the results of fuzz testing TAs. AFL found inputs that crashed 48 out of the 194 unique TAs. Surprisingly, 8 TAs crashed on single-byte inputs. All these single-byte input crashes, however, were because the TAs were not allocated sufficient shared memory for the command, and the TA tried to access unmapped pages. The GlobalPlatform TEE Internal API specification [21] does allow TAs to panic using a call to `TEE_Panic` on detecting exceptional conditions. However, these TAs did not detect exceptional conditions and relied on the TZOS to crash them if they accessed unmapped memory. This is a security issue if the address of such memory is attacker-controlled; however, we did not find this to be the case. On the other hand, some other TAs required long, specific sequences of inputs to crash them. For example, AFL found a specific 40-byte input to crash one TA. Blind fuzz testing has near-zero probability of finding such an input.

Next, we studied impact. AFL finds crashes which may or may not be exploitable. For each crash, we manually reverse-engineered the TA binary to determine how controllable pa-

[7]Kinibi also supports the GlobalPlatform TEE Client API [52]. However, the TAs we analyzed used Kinibi's own API.

Class	Vulnerability Types	Crashes
Availability	Null-pointer dereferences	9
	Insufficient shared memory crashes	10
	Other[note a]	8
Confidentiality	Read from attacker-controlled pointer to shared memory	8
	Read from attacker-controlled OOB buffer length to shared memory	0
Integrity	Write to secure memory using attacker-controlled pointer	11
	Write to secure memory using attacker-controlled OOB buffer length	2

Table 6: Crash classification. [note a]The "Other" availability type captures cases where attacker control of pointer or buffer length was insufficient to be exploitable, or if data read could not be leaked back through shared memory.

rameters related to the crash were, and classified them according to the descriptions in Table 6 as affecting TA confidentiality, integrity, or availability. In general, with vendors increasingly opening up access to the secure world to Android apps [25, 57], this could mean that a malicious Android app could potentially crash or exploit these TAs.

First, the impact of unavailability of a TA depends on whether each normal-world client gets its own instance of the TA or not. In QSEE, all normal-world clients share the same TA: the QSEE Linux kernel TEE driver does not launch a TA if one with the same name is already running [37]. In Kinibi, OP-TEE, and TEEGRIS, whether a single instance of a TA exists or not is controlled by property flags[8]. In the single-instance case, the impact of unavailability is potentially high: a client crashing a TA makes it unavailable to all other clients. For example, a malicious Android app with access to the secure world could crash a TA responsible for user authentication, thus locking users out of their phones [64]. Whether null-pointer dereferences are exploitable depends on what is mapped at low virtual TA addresses. None of the TAs that crashed had such mappings, however, so we classified them as availability issues.

Second, confidentiality and integrity issues can be exploited to leak or corrupt sensitive TA data depending on TA functionality. They can also be used as a step in privilege escalation to the TZOS [7]. We believe that most, if not all, of the crashes we found in these classes are exploitable. We were able to demonstrate three scenarios. First, we could get arbitrary code execution in a TA that controls access to the replay-protected memory block (RPMB) [3], which is persistent storage that increments a counter in hardware during writes to protect against replay attacks. Security-critical values stored here, such as minimum-allowed TA versions, are thus compromised. Second, we were able to leak arbitrary data from a digital-rights management (DRM) TA, thus compromising its keys.

[8]The GlobalPlatform TEE Internal API has a property (`gpd.ta.singleInstance` [21]) that specifies whether a TA should be single instance.

Third, we were able to compromise a one-time password TA, again leaking its keys. One of the arbitrary pointer dereference vulnerabilities we found was also found in parallel by another researcher, who developed an exploit to demonstrate arbitrary TA code execution [8]. Except this vulnerability, all other issues we found are previously unknown to the best of our knowledge.

We identified three patterns of developer mistakes specific to TrustZone development that caused several of these vulnerabilities. Further, two of these are specific to the TZOS APIs used. Such patterns highlight the need for TrustZone-specific and TZOS-API-specific developer education.

Assumptions of Normal-World Call Sequence. To minimize service time, TAs split work into small units; each unit has a sub-command that clients can call in sequence to achieve a bigger task. Thus, TAs are usually stateful: a typical session starts with an initialization call followed by other requests, and finally a close session call. TAs should not make any assumptions about the order of these calls, since a compromised normal world may issue these calls in any order. However, we found several TAs assumed a particular call sequence, resulting in using undefined data when a call was made out of sequence. While we only found null-pointer dereferences, confidentiality or integrity compromise is also possible.

Unvalidated Pointers from Normal World. Secure-world TAs communicate with normal-world client applications (CAs) using shared memory. In general, the normal-world CA does not know where such shared memory is mapped in the TA's virtual address space. However, Kinibi returns the virtual address of the base of this shared memory in the TA's address space to the CA [52], whereas QSEE identity-maps the shared memory. In both cases, the CA knows the virtual address of the shared memory in the TA's address space. The CA then constructs pointers to specific data in the shared memory that the TA can use. The TA developer should validate that these pointers refer only to addresses in shared memory before using them. However, we found such validation missing in some TAs. Thus, a normal-world CA can construct an arbitrary pointer into the TA's private data, call the TA, and have the TA either corrupt or leak this data depending on the call's functionality. While this issue is caused by developers missing the required security checks, and does not indicate a weakness in the TZOS itself, we found that this issue is more common in Kinibi TAs than QSEE TAs. This is perhaps because Kinibi requires such pointer construction to shared memory by design, whereas QSEE does not. This issue did not apply to either OP-TEE or TEEGRIS because they mapped shared memory buffers at a random virtual address, and because the GlobalPlatform TEE Client API they implement do not provide for shared memory pointers between the CA and TA.

Unvalidated Types. The GlobalPlatform TEE Client API [20], however, required a different check that was missing in some TAs. This API allows CAs to specify the type and content of four arguments for a command to a TA. The type can broadly be either a value or a buffer. We found that some TAs using these APIs implicitly assumed the types of arguments sent by the CAs. Thus, they interpreted a buffer address as a value, or worse yet, dereferenced a value. This results in vulnerabilities similar to those caused by unvalidated pointers from the normal world. We found instances of these that resulted in both confidentiality and integrity compromise.

9.2.1 Reproducibility and False Positives

We classify a TA crash as a false positive if it is not reproducible on a device. The general cause for false positives is a lack of fidelity in emulation of hardware or software components that TA interacts with. The only software component the TA interacts with is the TZOS, but this is unlikely to be a source of false positives since we reuse the original TZOS binary. Therefore, the most likely cause for false positives is insufficient hardware emulation. However, we found that only a few TAs interact with hardware and usually do not crash even if such hardware is unavailable, and thus PARTEMU's results have a high chance of being reproducible.

Our results are consistent with this intuition. We had devices corresponding to 24 out of the 48 unique crashes we found, and we were able to reproduce all 24 crashes on these devices. This included two TAs that accessed specialized hardware that we did not emulate. Out of the remaining crashes, only three other TAs accessed specialized hardware. If we conservatively assume that these three crashes are false positives, PARTEMU would have a true positive rate of 45/48 (93%), which we believe is sufficiently high to be useful.

9.3 Use Case: Fuzz Testing TZOS

Our second target for fuzz testing was the SMC API exported by the TZOS. We performed SMC API tests on one of our four target TZOSes - QSEE v4.0. Our aim was not to compare the security of TZOSes, but to show the utility of PARTEMU for TZOS testing: the reason we chose QSEE was because of its relatively simple and synchronous SMC calling convention [37]. In general, the normal world OS calls SMCs to request services from the secure monitor, TZOS, or TAs. This API is similar to the system call API: the caller specifies an SMC number and several arguments in registers.

The TZOS should protect itself from a compromised normal world that issues arbitrary SMCs. Consistent with this threat model, we used normal world driver programs to fuzz test the QSEE SMC API. The driver program gets the fuzz testing input from AFL, transforms these into SMC arguments, and sends the SMC. Crashes are detected if QEMU raises an abort. An additional challenge with APIs is that argument types can either be values or buffers. We use a part of the AFL input to determine argument types.

In total, AFL identified 124 distinct SMCs, and found crashes in 3 SMCs. These crashes only affected TZOS availability, and thus have limited security impact. However, interestingly, all these crashes tested QSEE code paths that would not normally be exercised on a real device, but those that could be triggered by an attacker who compromises the normal world. We discuss two cases below.

Normal-World Checks. One crash we found in QSEE that was independently fixed was an invalid pointer dereference triggered when the normal world requested the TZOS to load a TA that was already loaded. Interestingly, we found that this particular QSEE path was "shielded" by normal-world checks: QSEE's Linux kernel TEE driver [37], before sending a request to QSEE to load a TA, checked with QSEE if the TA was already loaded. If it was, the TEE driver did not send a request to load the TA at all. An attacker who compromises the Linux kernel itself, however, would not be restricted by this check, and could trigger this code path.

Assumptions of Normal-World Call Sequence. Another crash we found in QSEE was an uninitialized pointer dereference. This pointer was initialized by another SMC call that the Linux kernel on the device normally issued during boot. However, a compromised normal world would skip this SMC altogether, thus triggering this vulnerability. On a device, such a condition would normally not be triggered because the initialization would already have happened during boot.

10 Related Work

Closely related to our work are approaches that attempt to run real-world software in an emulator for dynamic analysis. Avatar [59], PROSPECT [28], Charm [49], and Surrogates [31] all attempt to enable dynamic analysis by running the target in a virtualized or emulated environment and forwarding accesses to real hardware. While Avatar, PROSPECT, and Surrogates target embedded device firmware, Charm targets Linux kernel device drivers running on mobile systems such as Android. These approaches work when the hardware exposes ways to interact with it, such as JTAG serial port, or USB. However, as we have seen, neither does TrustZone hardware exposes such interfaces, nor is it possible to run a software proxy for such hardware access in the TrustZone because of code signing.

Other approaches such as Costin *et al.* [15] and FIRMADYNE [11] attempt to emulate hardware to test embedded firmware. Hardware emulation was possible in these cases because the hardware was well-documented or standard. We study how to emulate the hardware required to run real-world TrustZone OSes, which is often non-standard and without documentation. Further, we show that it is possible to skip emulation of extremely complicated hardware by emulating other software components instead.

Firmware re-hosting [23] is the process of migrating firmware from its original hardware environment into a virtual environment. Pretender [23] attempts automated firmware re-hosting by generating hardware models using machine learning on runtime traces. P^{2}IM [19] uses manually-defined hardware register patterns and generates hardware models automatically on-the-fly by fitting different registers to these patterns at runtime. While these systems were tested on microcontrollers that are much simpler than our environment, they show the potential for automation of much of our work.

Concurrently with our work, Komaromy developed TEEMU [30], an emulator to run TAs for <t-base, an older version of the Kinibi TZOS. In contrast to our work, TEEMU does not re-host the <t-base TZOS itself. Instead, TEEMU emulates the TZOS by manually re-implementing specific <t-base system calls. This limits TEEMU to testing <t-base TAs that use only those system calls, and does not allow testing the <t-base TZOS itself. Furthermore, reproducibility is dependent on the fidelity of re-implementation of the TZOS system calls. Similar limitations apply to the Open-TEE [33,35] project, which is a virtual TZOS implementing the GlobalPlatform TEE API [21]. In contrast, PARTEMU supports full-system emulation by re-hosting unmodified TA and TZOS binaries, allowing holistic testing of TrustZone and making it significantly more likely that any issues found are reproducible on a real device.

PARTEMU enables using advances in dynamic analysis on real-world TrustZone software. Thus far, the main technique to analyze real-world TrustZone software has been static binary reverse-engineering of TAs and the TZOS [7,8]. Dynamic analysis for TrustZone software has been limited to blind fuzzing [6] and emulation of particular parts of TrustZone [30]. PARTEMU enables dynamic analysis techniques such as feedback-driven fuzz testing [9,12,40,61], symbolic and concolic execution [10,13,48], taint analysis [14,17,58], and debugging for real-world TrustZone software.

11 Discussion and Future Work

Dealing with Stateful TAs. On a random sample of 10 TAs, AFL had basic-block coverage varying from 0.2% to 45.6% with a median of 17.7%. We found that a major limiting factor for coverage was TA state: we noticed that several TAs had internal finite state machines and therefore required a sequence of multiple inputs to drive them to interesting states (e.g., connected, authorized, processing). Our driver currently sends a single message to a newly forked TA instance each time so that AFL does not have issues with stability (Section 8.1). Therefore, we cannot get past state checks, which require a sequence of inputs. We plan to handle TA state in future work. Even with such limited coverage, however, as we have seen, PARTEMU was able to find several non-trivial real-world vulnerabilities.

Hardware Roots of Trust. PARTEMU does not emulate hardware roots of trust. An example is the factory-installed per-device private key signed by the Samsung CA [42] and used for remote attestation. Thus, code paths in TAs that depend on remote attestation succeeding may not work. For example, Samsung Pay uses remote attestation for credit card enrollment; we cannot successfully enroll a credit card using a Samsung Pay TA [44] running on PARTEMU because we do not have access to the attestation key that would be present on a real device. While such TAs that depend on a valid root of trust require other techniques to test, they are few in number, and PARTEMU is able to test the vast majority of TAs.

Performance. Since we ran PARTEMU on an x86 machine, we could not take advantage of ARMv8 hardware virtualization [16]. AFL ran at around 10-25 executions per second for QSEE, OP-TEE, and TEEGRIS, while our performance optimizations for Kinibi (Section 8.3) enabled 125 executions per second. While even this was sufficient to find several non-trivial vulnerabilities, we believe PARTEMU would be even more useful if it could run faster. To this end, we plan to explore running PARTEMU directly on ARMv8 hardware.

12 Conclusion

In this work, we addressed the problem of the lack of dynamic analysis for real-world TrustZone software by building an emulator that runs four widespread, real-world, TZOSes - QSEE, Kinibi, TEEGRIS, and OP-TEE. We studied the software and hardware emulation effort required to run these TZOSes. We found that emulating the required hardware and software dependencies was feasible. We implemented our emulation on PARTEMU, enabling dynamic analysis of real-world TZOSes. We showed PARTEMU's utility by finding 48 previously-unknown vulnerabilities across 194 TAs from 12 different Android smartphone vendors and an IoT vendor. We identified patterns of developer mistakes unique to TrustZone development that cause some of these vulnerabilities, highlighting the need for TrustZone-specific developer education. This work shows that dynamic analysis of real-world TrustZone software using emulation is both feasible and beneficial.

Disclosure

We have notified each vendor of any relevant findings and are working with their security teams to address the issues.

13 Acknowledgements

We thank Stephen McLaughlin for automating parts of our analysis and interpreting results. We thank our anonymous reviewers for their suggestions that helped significantly improve the presentation and scope of this work. We thank Peng Ning, Kunal Patel, Laurent Simon, Luke Deshotels, and Stephen McLaughlin for helpful discussions and suggestions during various stages of this work.

References

[1] Apple. iOS Security. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf.

[2] ARM. ARM TrustZone. http://www.arm.com/products/processors/technologies/trustzone/index.php.

[3] J. S. S. T. Association. Embedded Multimedia Card eMMC. http://www.jedec.org/standards-documents/results/JESD84-A.

[4] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. 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*, 2014.

[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.

[6] G. Beniamini. FuzzZone. https://github.com/laginimaineb/fuzz_zone/tree/master/FuzzZone.

[7] G. Beniamini. TrustZone Kernel Privilege Escalation. http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html.

[8] D. Berard. Kinibi TEE: Trusted Application exploitation. https://www.synacktiv.com/posts/exploit/kinibi-tee-trusted-application-exploitation.html.

[9] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[11] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[12] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[14] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[15] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[16] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[17] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW*, 2015.

[18] J.-E. Ekberg. Trusted Execution Environments (and Android). https://usmile.at/sites/default/files/androidsecuritysymposium/presentations2015/Ekberg_AndroidAndTrustedExecutionEnvironments.pdf.

[19] B. Feng, A. Mera, and L. Lu. P^{2}IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[20] GlobalPlatform. TEE Client API Specification v1.0. https://globalplatform.org/specs-library/tee-client-api-specification/.

[21] GlobalPlatform. TEE Internal Core API Specification v1.2.1. https://globalplatform.org/specs-library/tee-internal-core-api-specification/.

[22] I. GlobalPlatform. GP TEE Certificate: TEEgris 2.5 on MT6737T. https://globalplatform.org/wp-content/uploads/2018/03/GP-TEE-2017_01_Certificate_MediaTek_GP170002_20171027_Gil.pdf.

[23] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna. Toward the analysis of embedded firmware through automated rehosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019.

[24] J. Hertz and T. Newsham. AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL.

[25] IETF. The Open Trust Protocol (OTrP). https://www.ietf.org/archive/id/draft-pei-opentrustprotocol-06.txt.

[26] JEDEC. e.MMC v5.1A. https://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc.

[27] JEDEC. Universal Flash Storage (UFS) 3.0. https://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs.

[28] M. Kammerstetter, C. Platzer, and W. Kastner. Prospect: peripheral proxying supported embedded code testing. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, 2014.

[29] L. Kernel. LK embedded kernel. https://github.com/littlekernel/lk.

[30] D. Komaromy. Unbox Your Phone - Exploring and Breaking Samsung's TrustZone Sandboxes. http://www.ekoparty.org/charla.php?id=756.

[31] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015.*, 2015.

[32] Linux. Qualcomm Secure Memory Manager binding. https://github.com/torvalds/linux/blob/master/Documentation/devicetree/bindings/soc/qcom/qcom,smem.txt.

[33] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan. Open-TEE – an open virtual trusted execution environment. Technical report, Aalto University, 2015.

[34] OP-TEE. Open Portable Trusted Execution Environment - OP-TEE. https://www.op-tee.org/.

[35] Open-TEE. Open-TEE. https://open-tee.github.io/.

[36] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), Oct. 2019.

[37] Qualcomm Android TEE Driver. https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/QSEEComAPI.h.

[38] Qualcomm. Qualcomm Security for Mobile Computing. https://www.qualcomm.com/solutions/mobile-computing/features/security.

[39] Qualcomm. Secure Boot and Image Authentication. https://www.qualcomm.com/media/documents/files/secure-boot-and-image-authentication-technical-overview-v2-0.pdf.

[40] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[41] Samsung. Knox Platform for Enterprise White Paper. https://docs.samsungknox.com/whitepapers/knox-platform/samsung-knox.htm.

[42] Samsung. Knox Platform Security. https://developer.samsung.com/tech-insights/knox/platform-security.

[43] Samsung. Samsung TEEGRIS. https://developer.samsung.com/teegris.

[44] Samsung. Secured Communication with the Payment Networks. https://developer.samsung.com/tech-insights/pay/secured-communication-with-the-payment-networks.

[45] D. Shen. Attacking your "Trusted Core" Exploiting TrustZone on Android. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf.

[46] A. Software. Trusted Firmware-A. https://github.com/ARM-software/arm-trusted-firmware.

[47] I. Standard. IEEE Standard Test Access Port and Boundary-Scan Architecture 1149.1-1990. https://ieeexplore.ieee.org/document/938734, 1990.

[48] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS*, 2016.

[49] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[50] A. Tarasikov. Reverse Engineering Samsung Exynos. http://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html.

[51] threatpost.com. Android Qualcomm Vulnerability Impacts 60 Percent of Devices. https://threatpost.com/android-qualcomm-vulnerability-impacts-60-percent-of-devices/118191/, Visited Aug 2019.

[52] Trustonic. Android Driver for the Trustonic Trusted Execution Environment. https://github.com/TrustonicNwd/tee-mobicore-driver.kernel.

[53] Trustonic. Android user space components for the Trustonic Trusted Execution Environment. https://github.com/TrustonicNwd/tee-mobicore-driver.daemon.

[54] Trustonic. Device Coverage: Trustonic Embeds Hardware Security in 9 of the Top 10 Android OEMs. https://www.trustonic.com/trustonic-device-coverage, Visited Aug 2019.

[55] Trustonic. Internet of Things. https://www.trustonic.com/markets/iot/.

[56] Trustonic. Trustonic Application Protection. https://www.trustonic.com/solutions/trustonic-application-protection-tap/.

[57] Trustonic. Trustonic Secured Platforms. https://www.trustonic.com/solutions/trustonic-secured-platforms-tsp/.

[58] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[59] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[60] M. Zalewski. AFL: Understanding the Status Screen. http://lcamtuf.coredump.cx/afl/status_screen.txt.

[61] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[62] zdnet.com. Security flaw lets attackers recover private keys from Qualcomm chips. https://www.zdnet.com/article/security-flaw-lets-attackers-recover-private-keys-from-qualcomm-chips/, Visited Aug 2019.

[63] H. Zhang, D. She, and Z. Qian. Android root and its providers: A double-edged sword. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[64] H. Zhang, D. She, and Z. Qian. Android ion hazard: The curse of customizable memory management system. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

A Appendix

A.1 Selecting Components to Emulate

Table 7 shows whether we choose to emulate or reuse each component that the TZOS depends on, based on the criteria in Table 1.

A.2 SLOC for Emulated Components

Table 8 quantifies our implementation effort using source lines of code for core PARTEMU and each emulated component.

TZOS	Component *C*	*C* – TZOS Coupling	*C* – Other Coupling	Source Avail?	Encrypted?	Decision
QSEE	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Tight	Loose	Closed	No	Reuse
	TEE Driver	Loose	Loose	Open	No	Emulate[note b]
	TEE Userspace	N/A	N/A	Closed	No	Exclude
Kinibi	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Loose	Tight	Partial	No	Emulate
	TEE Driver	Tight	Loose	Open	No	Reuse
	TEE Userspace	Loose	Tight	Open	No	Emulate
TEEGRIS	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Loose	Tight	Partial	Yes	Emulate
	TEE Driver	Tight	Loose	Open	No	Reuse
	TEE Userspace	Loose	Tight	Closed	No	Emulate
OP-TEE	Bootloader	Loose	Tight	Partial	No	Emulate
	Secure Monitor	Tight	Loose	Closed	No	Reuse
	TEE Driver	Loose	Loose	Open	No	Emulate[note b]
	TEE Userspace	N/A	N/A	Closed	No	Exclude
TAs[note a]	-	-	-	-	Reuse	
TZOSes[note a]	-	-	-	-	Reuse	
Hardware	-	-	-	-	Emulate	

Table 7: Showing components chosen for emulation or reuse for QSEE, Kinibi, TEEGRIS, and OP-TEE.
[note a]Since the TZOS and TAs are the target components we want to analyze, we have to reuse the original binaries.
[note b]We believe both reusing or emulating the TEE drivers in these cases are practically feasible.

Category	Component	New or Modification to Existing Code?	SLOC Added or Modified
PartEmu	QEMU (PartEmu run management API)	New	1060
	PartEmu AFL plugin	New	846
	PartEmu LLVM run plugin	New	147
QSEE	QEMU (hardware emulation)	New	4642
	Bootloader	New	1636
	TEE driver+AFL driver	New	1379
Kinibi	QEMU (hardware emulation)	New	551
	Secure Monitor	Existing [46]	781
	TEE userspace	Existing [53]	49
	AFL driver	New	656
TEEGRIS	QEMU (hardware emulation)	New	551
	Secure Monitor	Existing [53]	677
	TEE userspace	New	435
	AFL driver	New	542
OP-TEE	QEMU (hardware emulation)	New	310
	Bootloader	New	2[note a]
	AFL driver	New	266
Total	-	-	**14530**

Table 8: Table with lines of code added or modified for each emulated component. SLOC was calculated using `sloccount`.
[note a]OP-TEE only required the bootloader to set up two registers, which we did using two assembly instructions.