TEE-BONE: Securing Smartphone Apps Using Hardware-Only Isolation Primitives

Thesis advisor: Professor James Mickens Kevin R. Loughlin

TEE-BONE: Securing Smartphone Apps Using Hardware-Only Isolation Primitives

Abstract

Modern device manufacturers often rely on a combination of hardware-assisted virtualization and privileged software to isolate a security-critical trusted execution environment (TEE) from a general-purpose rich execution environment (REE). Prior EE isolation technologies have required software support due to both the complexity of their models and a lack of fully-virtualizable phones (i.e., phones in which every physical resource can be virtualized by the hardware). Unfortunately, EE isolation models such as ARM TrustZone expect the privileged software to manage a complex set of inter-EE tasks, resulting in a large threat surface for attackers wishing to bypass EE isolation.

We propose that by removing unnecessary inter-EE functionality and expanding native hardware virtualization throughout the device, we can achieve EE isolation purely via hardware-based isolation mechanisms. We present Tee-Bone, the first smartphone EE isolation technology to implement all EE isolation mechanisms and policies in the device's hardware. At phone manufacture time, the manufacturer creates a static, immutable partitioning of the virtual resources belonging to each hardware component. Tee-Bone provides no mechanisms for inter-EE communication, prohibits simultaneous execution of EEs, and requires human-hardware interaction to switch between EEs. By placing these restrictions on EE usage, Tee-Bone can eliminate complex trusted software and its

associated threat surface. We argue that this approach will improve security while imposing minimal degradation of phone usability.

Contents

1 Introduction 1

2 Background 7

3 Problem Definition 12
- 3.1 Threat 1: Inter-EE Communication . . . 13
- 3.2 Threat 2: Simultaneous Execution of EEs . . . 14
- 3.3 Threat 3: Lack of Bidirectional EE Isolation . . . 15
- 3.4 Threat 4: Failure to Acquire Explicit User Consent . . . 16
- 3.5 Summary of Goals . . . 16

4 Design 18
- 4.1 New Hardware Components . . . 19
- 4.2 Virtualized Hardware Components . . . 21
- 4.3 Boot Process . . . 24
- 4.4 EECS Process . . . 24

5 Implementation 27
- 5.1 QEMU Setup . . . 28
- 5.2 Hardware Partitioning and Virtualization . . . 29
- 5.3 Board Initialization and Boot Process . . . 32
- 5.4 Emulating an EECS . . . 32
- 5.5 Remaining Work . . . 33

6 Related Work 35
- 6.1 Intel SGX . . . 35
- 6.2 AMD SVM/Intel TXT . . . 37
- 6.3 ARM Trusted Firmware . . . 38
- 6.4 GlobalPlatform TEE System Architecture specification . . . 38

7 Conclusions 40

References 43

Acknowledgments

There are a number of people I'd like to thank for helping me produce this senior thesis. I'll begin with my family – my Mom, Dad, and two brothers have provided me with a loving home, as well as all the opportunities I could ever ask for. In addition to caring for me as a son and brother, they have always supported and encouraged my academic pursuits, especially my interest in research.

I also extend my thanks to my friends, who never fail to put a smile on my face (even when I'm writing a thesis). Thanks for being there for me through the good and the bad. I especially thank Amy Kang and Ezra Zigmond for their helpful feedback on a draft of this thesis.

I'd like to thank Professor Eddie Kohler and Professor Jim Waldo for agreeing to read and critique this thesis. Their feedback will prove invaluable to the continued development of this project and to my growth as a researcher.

I'd also like to thank Professor Margo Seltzer (as well as Eddie) for inspiring my interest in systems. Even more importantly than being wonderful professors, they are extremely caring individuals who made every effort to assist me during some difficult personal times. I wish Margo the best in her new deanship at UBC, and I hope that Eddie doesn't get into too much trouble in her absence :-)

Finally, and most importantly, I thank Professor James Mickens for everything that he has done for me over the past year and a half. James has nurtured and developed my love for systems security research, and my experiences with him are what ultimately convinced me to pursue a PhD in Com-

puter Science next fall. In addition to being a fantastic researcher, mentor, and teacher, he is one of the funniest and kindest people I've had the pleasure of knowing. I am blessed to have him as my thesis advisor, but I am even more blessed to have him as my friend.

1

Introduction

Smartphones are the primary computational device for many individuals, and frequently execute programs that are trusted to manipulate sensitive data, such as banking apps. Unfortunately, smartphones may also execute a wide variety of potentially malicious programs. Thus, smartphones require isolation mechanisms that prevent untrusted programs from tampering with the code or data of trusted applications.

A popular way to provide isolation is to place the untrusted and trusted apps in separate execution environments (EEs), where an EE represents a logical collection of hardware and software components. The rich execution environment (REE) is a general-purpose EE, and is relatively large and complex in comparison to the trusted execution environment (TEE). The TEE is intentionally limited in its functionality, with a trusted computing base (TCB) that is designed to be small. The TEE's simplicity and security-driven design make it ideal for executing lightweight applications that manipulate sensitive user data.[11,24]

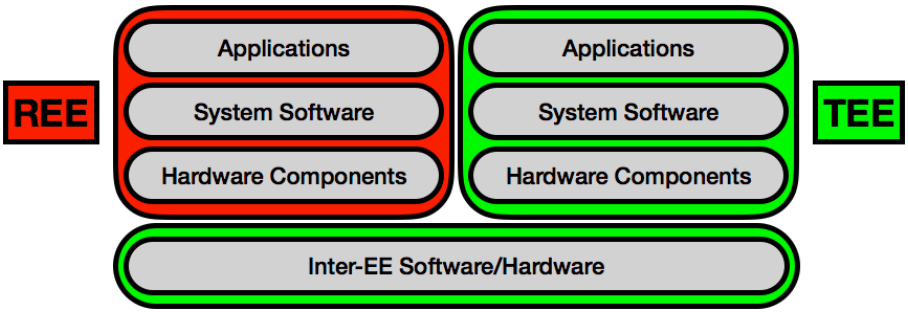

Figure 1.1: Two different execution environments (EEs) atop one physical machine. Note that the inter-EE software and hardware is also shown in green because it is implicitly trusted, given that the TEE depends on its reliability.

ARM chips are pervasive on smartphones, making ARM TrustZone the most common isolation technology for mobile devices.[16,20] TrustZone and other isolation models have traditionally relied on a small, simple set of hardware primitives, plus a privileged software layer.[13] For example, TrustZone adds an extra bit to memory addresses to divide the physical address space into REE and TEE partitions, and also provides separate registers for REE and TEE stack pointers. TrustZone relies on secure monitor software – in the form of privileged firmware – to handle inter-EE tasks such as

scheduling and communication.[3] Ultimately, the secure monitor software must multiplex the REE and TEE atop a single set of hardware components.

Unfortunately, multiplexing low-level hardware is complicated. Thus, the software-level monitor is large, providing a sizable threat surface for attackers wishing to bypass isolation. TrustZone's complexity (and therefore large attack surface) arises from two key facets:

1. TrustZone allows for the REE and TEE to send each other messages via dedicated machine instructions.[5] Accordingly, software must be augmented to appropriately sanitize these messages.
2. TrustZone permits REE and TEE code to execute simultaneously on multi-core systems.[3] As such, the secure monitor must include code to manage synchronized access of shared hardware resources such as disk.

We argue that supporting inter-EE communication and simultaneous execution is dangerous for maintaining trusted execution integrity, and is furthermore unnecessary in smartphone environments. For example, a banking customer is perfectly capable of flipping a switch to enter into the TEE, as well as subsequently launching and carrying out their transaction(s) before returning to the REE.

In cryptographic use cases such as digital rights management (DRM) – in which interaction between an EE and EE-external data has traditionally been necessary – one can alternatively support two worlds within an EE, isolated via the TrustZone model. While the isolation guarantees between these worlds are not as strong as those between EEs, support for TrustZone-based isolation within a single EE eliminates the need for inter-EE communication and simultaneous execution. Furthermore, such a setup provides support for legacy TrustZone code at no risk to inter-EE security.

Given a system that removes all communication channels between the REE and TEE prohibits simultaneous execution of the two EEs, the pieces of secure monitor code needed for these tasks can be eliminated. As a result, we can dramatically reduce the size and complexity of the TCB and therefore improve security.

Furthermore, under such a model, one can achieve EE isolation *without any software assistance*. In addition to their complexity, prior smartphone isolation models such as TrustZone have required software support due to a lack of fully-virtualizable phones (i.e., phones in which every physical resource can be virtualized by the hardware). However, we propose that it is feasible to construct a fully-virtualizable phone. Combined with our simplified isolation model, such a device allows the remaining set of isolation tasks to instead be implemented in the hardware, thereby completely eliminating the inter-EE software attack surface.

The recent Meltdown and Spectre attacks demonstrate the dangers of *simultaneously* hosting trusted and untrusted state atop a single set of complex hardware resources.[14,15] By eliminating multiplexing between EEs, we can eliminate entire classes of hardware-based exploits, such as inter-EE cache timing attacks. Admittedly, these security guarantees come at the cost of worse performance due to increased latencies when switching between EEs. However, we believe these performance costs to be modest in comparison to the security benefits provide by such a setup, especially as the capabilities and use of trusted applications continue to increase.

This paper describes Tee-Bone, a concrete design for a hardware-only isolation model. Tee-Bone uses a modified TrustZone architecture. Whereas TrustZone only uses hardware-based isolation in the processor, generic interrupt controller (GIC), and address space controller (ASC), Tee-

Bone adds native hardware virtualization support to RAM, the hard drive, the network interface card, and the screen, as pictured in Figure 1.2.

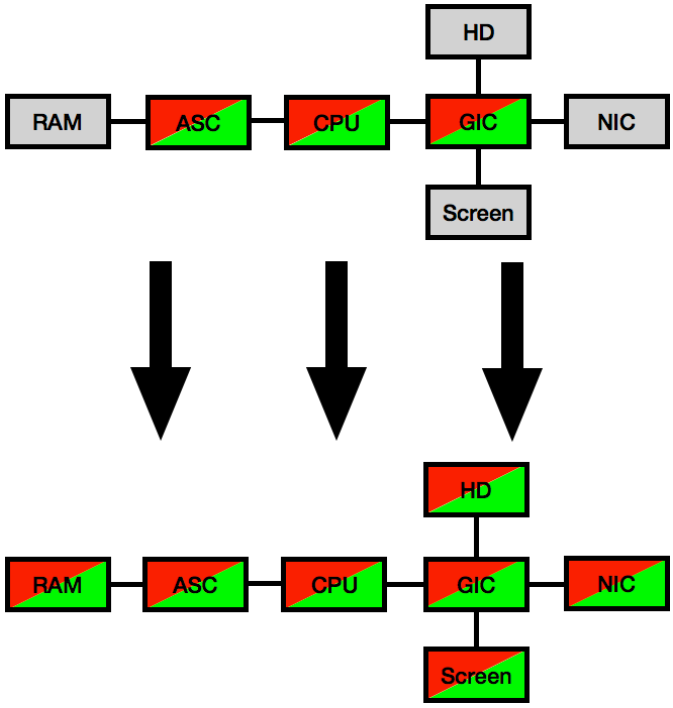

Figure 1.2: The expansion of native hardware virtualization from the TrustZone model (top) to the Tee-Bone model (bottom). Green represents a trusted virtual component, whereas red represents its untrusted counterpart. Note that native hardware virtualization is expanded to RAM, the HD, the NIC, and the screen in the Tee-Bone model.

At phone manufacture time, the manufacturer creates a static, immutable partitioning of the virtual resources belonging to each hardware device. The manufacturer can then install the desired operating systems on their respective partitions. The physical hardware in Tee-Bone prevents software in one EE from accessing resources in the other EE. Furthermore, only one EE can run at a time, with the active EE determined via an external button on the device. This external button is the sole

mechanism for changing between EEs; furthermore, it necessitates physical human interaction, ensuring that user consent is always obtained to swap EEs.

In summary, prior isolation mechanisms require a complex, software-level monitor that has a large threat surface. However, such a monitor is only necessary if one wishes to support inter-EE communication and concurrent EE execution. We argue that these features can be eliminated without significant drawbacks to the user experience, resulting in a dramatic decrease in TCB size. By additionally making a small set of changes to smartphone hardware, we can implement a simple EE isolation policy exclusively in hardware, completely eliminating the inter-EE software attack surface.

The remainder of this thesis is organized as follows. Section 2 provides necessary background information on TrustZone. Section 3 defines our problem, providing our threat model and design goals. Section 4 presents the technical details of our design, while Section 5 discusses the in-progress implementation of Tee-Bone atop Quick Emulator (QEMU). Section 6 offers related work, and Section 7 contains concluding remarks.

2

Background

ARM chips and the associated TrustZone isolation framework are widely deployed on modern smartphones. Thus, TrustZone is a natural extension point for Tee-Bone's design. In this section, we discuss the aspects of TrustZone that are necessary to understand Tee-Bone's design.

TrustZone is designed to prevent REE code from accessing TEE resources. To achieve this goal, TrustZone uses a combination of hardware mechanisms and software mechanisms. In TrustZone

parlance, EEs are referred to as "worlds," with the normal world corresponding to the REE and the secure world corresponding to the TEE.

From a hardware standpoint, each core on the processor contains a Non-secure (NS) bit to indicate its current security state. Each core can thus execute REE or TEE code in a time-sliced fashion, independent of the other cores' security states. The physical core presents a separate virtualized view of itself to each EE.[3]

The ARM architecture supports a higher-priority fast interrupt request (FIQ) line and a lower-priority interrupt request (IRQ) line, managed by the generic interrupt controller (GIC). ARM recommends using the FIQ line exclusively for secure world interrupts, such that the secure world cannot be disrupted by normal world IRQ interrupts. For example, secure world timers should be mapped to the FIQ line, whereas normal world timers should be mapped to the IRQ line. Additionally, normal world software can only program IRQ interrupt vectors, ensuring that normal world software cannot mask secure FIQ interrupts.[7]

The system interconnect includes the NS bit in all memory accesses to provide separate address spaces to the REE and TEE. The TrustZone ASC can be programmed to map these address spaces to distinct physical memory partitions based on the NS bit. Note that TEE resources can access both the TEE and REE address spaces, while REE resources are restricted to the REE address space. The REE address space is thus a proper subset of the TEE address space.* For peripherals that must make

*As ARM notes, TrustZone-based systems commonly allow the TEE to see all system state, while only permitting the REE to see REE state. In the case of memory management, it is theoretically possible for a system implementing TrustZone to have physically separate memory systems for the secure and normal worlds. However, most real systems instead share physical memory devices between the two worlds, using the NS bit as an isolation mechanism.[4]

direct memory accesses (DMAs), the system memory management unit (SMMU) can be configured similarly to the ASC to protect secure RAM from regular world devices.[19] Finally, processor cache lines are also tagged with the NS bit, eliminating the need for cache flushes during an EECS.

To complement the hardware-based isolation mechanisms of the processor, GIC, and ASC, TrustZone relies on privileged software to handle EECS requests. Prior to TrustZone, ARM processors supported 3 software privilege levels – EL0, EL1, and EL2 – with privileges increasing from EL0 to EL2. TrustZone's adds a fourth and highest software privilege level (EL3) for the management of EECS transitions. Figure 2.1 depicts the various pieces of trusted and untrusted software on an ARM system with TrustZone security extensions.

Each of the four privilege levels has its own set of stack pointers, stored in banked registers for which access is protected by privilege level. Levels EL1-EL3 also contain banked registers that store pointers to exception handlers, with access again protected by privilege level.[6] The least privileged level (EL0) is intended to be used for the execution of unprivileged application code, and can execute in either the REE or TEE. Such a setup allows for distinguishing between rich applications (RAs) and trusted applications (TAs). EL1 is designed for use by privileged OS code, and can likewise execute in either EE – therefore distinguishing between the Rich OS (ROS) and Trusted OS (TOS). EL2 code can only be executed in the REE (NS=1), and is intended to support a hypervisor that manages each ROS. Finally, EL3 code runs code for a secure monitor that implements EECS transitions. EL3 code can only be invoked by software that runs with NS set to 0, with a swap back to 1 occurring immediately upon return to the REE.

Transitions from the REE to the TEE occur in one of three ways. Firstly, the normal world can

receive an external abort signal. Secondly, EL3 code can map hardware interrupts to TEE FIQ exception handlers. Thirdly, privileged (EL1 or EL2) code in the REE can issue a secure monitor call (SMC) instruction. [20]

Similar to secure interrupts, the SMC forces the processor to trap to EL3 exception handling code. The instruction takes a desired TEE function ID as an argument, as well as the contents of up to six other 64-bit registers. It returns up to four 64-bit registers worth of data. [5]

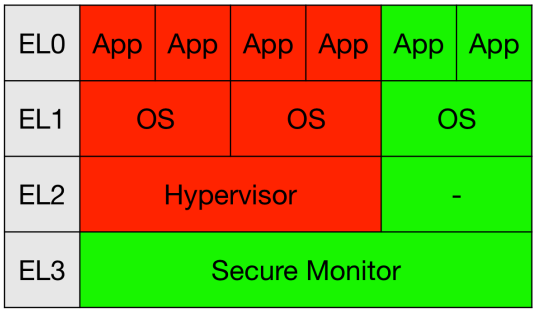

Figure 2.1: The ARM TrustZone EE isolation model. The normal world (REE) software stack is pictured in red in the left column, while the secure world (TEE) software stack is pictured in green on the right. Note that the trusted secure monitor software (EL3) is located in both the secure and normal world software stacks, providing a potential software attack surface for violating EE isolation.

3

Problem Definition

While TrustZone provides a well-defined isolation model, the complex set of tasks expected of the secure monitor software result in a large body of EL3 code. The presence of this EL3 code in both the TEE and REE software stacks provides a sizable threat surface for attackers wishing to bypass isolation.

Here we enumerate the threat surfaces exposed by the TrustZone model, proposing solutions

that we elaborate upon in section 4. Unless otherwise stated, our adversary models assume the attacker has complete access to the REE software stack, but not to the TEE software stack beyond the entry points provided by TrustZone. We additionally assume the attacker does not have physical access to the device. While physical attacks are a viable threat, their prevention is not a focus of our research.

3.1 Threat 1: Inter-EE Communication

In order to facilitate communication between the EEs, TrustZone gives EL3 secure monitor software access to both the TEE and REE software stacks (see Figure 2.1). As such, TrustZone inherently cannot provide 100% isolation of the REE and TEE software stacks.

An attacker could exploit bugs in any of the EL3 entry points – including SMCs and FIQs – from the normal world in order to subvert isolation. For instance, consider the parsing of SMC arguments by the secure monitor. These arguments are created by the untrusted REE. If the argument-parsing code in EL3 contained a buffer overflow vulnerability or failed to appropriately sanitize REE input, an attacker could subvert the trusted monitor.

Many current TrustZone use cases currently require interaction between an EE and EE-external code and data. For example, in the case of DRM, a content distributor can place a decryption key and code that checks whether the user has permission to view the distributor's material into the secure world. When the normal world media player wishes to play the encrypted file, it issues a SMC to this trusted DRM service, which can subsequently decrypt the content for REE playback.

However, we will show that a Tee-Bone EE can be divided into the secure and normal worlds of TrustZone. Therefore, the secure world can perform the standard TrustZone-style cryptographic services on behalf of the normal world within the same EE, thereby eliminating the need for inter-EE communication mechanisms. As such, we propose removing communication between the REE and TEE software stacks. In addition to eliminating the risk of the above attacks, such a change simplifies the isolation model and reduces the TCB, building towards the possibility of hardware-only isolation mechanisms.

3.2 Threat 2: Simultaneous Execution of EEs

Multi-core TrustZone-based systems allow for parallel execution of TEE and REE code. In particular, the NS bit is per-core, as opposed to being system-wide. Therefore, the secure monitor software must ensure synchronized access to shared hardware resources, such as the NIC.

However, synchronization is a complex task that augments the TCB, therefore increasing the likelihood of bugs in EL3 software. A single bug in the software's inter-EE synchronization logic could break isolation. In our view, the importance of maintaining trusted execution integrity – for example, in financial applications – outweighs the performance benefit of allowing trusted services to run simultaneously with untrusted apps.

Thus, we propose prohibiting simultaneous execution of EEs throughout the smartphone. By doing so, we eliminate all EE isolation vulnerabilities arising from synchronization issues. Additionally, we again reduce and simplify the TCB of the isolation model.

3.3 Threat 3: Lack of Bidirectional EE Isolation

TrustZone does not provide equal bidirectional isolation of the TEE and REE. Namely, TrustZone allows for TEE code to modify REE state. However, we view ensuring the bidirectional isolation to be of great importance.

While TEE code is trusted by definition, absolute trust can only come with complete formal verification of complex software, which is presently unrealistic. Furthermore, software in both EEs handles sensitive code and data that should not be available to external environments, including other EEs on the same system. For example, the TEE should not be able to overwrite the REE kernel's scheduling routine – at least not without explicit user consent. Ultimately, granting the TEE access to all REE code and data exposes an unnecessary attack surface.

Indeed, the Boomerang confused deputy attack takes advantage of this attack surface to perform a privilege escalation in the REE. At a high level, the attack involves unprivileged REE code using the TEE to modify privileged REE state. The authors demonstrate that the ROS often fails to properly sanitize arguments provided by an unprivileged RA. Because of this, the ROS can be tricked into issuing an SMC with malicious arguments. In turn, the TEE is fooled into (for example) reading and returning the list of REE usernames and passwords, thinking that the ROS has knowingly requested this behavior via its SMC.[16] To eliminate the risk of such an attack, our model will prohibit REE access to TEE resources, as well as TEE access to REE resources.

3.4 Threat 4: Failure to Acquire Explicit User Consent

Lastly, the TEE is invoked by the REE in the TrustZone model. In particular, privileged REE software is responsible for issuing an SMC to trigger an EECS. Unfortunately, such a setup means that a compromised ROS can simply refuse to issue SMCs, thereby denying the user access to the TEE. Alternatively, a compromised ROS could use the SMC to invoke an arbitrary TEE service with arbitrary arguments, against the user's will.

We can prevent these denial-of-service (DoS) and unauthorized usage attacks by requiring human-hardware interaction to induce an EECS. Namely, if a human must physically interact with hardware to invoke an EECS, then software mechanisms would be unable to force or prevent an EECS. We therefore propose replacing the software SMC mechanism for EECS with an external EECS button.

3.5 Summary of Goals

Our research aims to eliminate the possibility of software-based exploits of EE isolation on smartphones. In order to achieve this goal, we seek to simplify the TrustZone model, such that a minimally-sufficient subset of software-based isolation mechanisms can be shifted to the device's hardware.

By prohibiting simultaneous execution of the REE and TEE, removing communication mechanisms between EEs, and providing equally strong isolation in both directions, we greatly reduce and simplify the TCB – correspondingly, EE isolation becomes far less likely to be exploitable. By additionally requiring human-hardware interaction to perform an EECS, we can even eliminate inter-EE

denial-of-service (DoS) attacks.

Finally, by virtualizing all of a smartphone's devices, EE isolation can ultimately be achieved via hardware-only isolation mechanisms. In such a setup, the software attack surface would be *entirely eliminated*. We demonstrate the ability to achieve EE isolation via hardware-only isolation mechanisms in our design and implementation sections.

4

Design

Tee-Bone is a modified ARMv8-A architecture in which all EE isolation mechanisms and policies are implemented by hardware. At a high-level, our design modifies the CPU, GIC, RAM, HD, NIC, and screen to support *pervasive hardware-level virtualization* – in Tee-Bone, each EE runs atop an isolated, virtualized copy of the smartphone system.

Because Tee-Bone presents a fully-virtualized view of the smartphone system to each EE, back-

wards compatibility with TrustZone can be maintained for legacy EL3 code at no risk to Tee-Bone security. Thus, in the parlance of TrustZone, a single Tee-Bone EE can host a secure world and a normal world. Tee-Bone's isolation model is pictured in Figure 4.1.

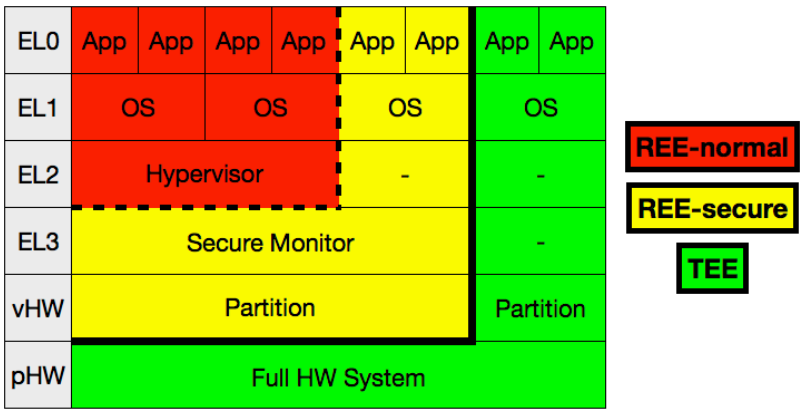

Figure 4.1: TEE-BONE isolation model, including TrustZone support in the REE. Isolation between the REE and TEE is enforced solely by TEE-BONE hardware, while REE-secure world and REE-normal world isolation occurs via legacy TrustZone mechanisms. Note that the pHW does not handle a trusted EE differently from a rich EE; the pHW merely enforces device virtualization boundaries, and manages EE context switches. Thus, although the figure shows both the pHW and the trusted EE in green, the pHW's correctness does not depend on the trusted EE; furthermore, the pHW does not enable new (virtual) hardware functionality when the trusted EE runs.

4.1 New Hardware Components

Virtualizing individual hardware devices is necessary but insufficient to achieve Tee-Bone's goals; we must additionally supply hardware to manage EECS transitions. We therefore provide a system-wide REE bit, an external EECS button, a dedicated interrupt line connecting the EECS button to the CPU, and a register scratch device (RSD) per-EE. Since these components manage EECS transitions, they are *not* virtualized, nor are they exposed as software-visible architectural state. An

Component	Role
REE bit	Indicates whether CPU has REE permissions (1) or TEE (0)
EECS interrupt port	Allows incoming EECS interrupt to bypass GIC and halt CPU
RSD per EE	Stores each EE's CPU and GIC register state while inactive

Table 4.1: Brief overview of new architectural components that support EE context switches

overview of the new hardware components that support EE context switches is provided in Table 4.1.

4.1.1 REE Bit, EECS Button, and Interrupt Line

We add a system-wide REE bit to the phone that can only be flipped by an EECS interrupt. Because there is no concurrent execution of code from different EEs in Tee-Bone, an EECS interrupt causes a complete suspension of EE execution on all cores. Execution state is then saved (described in the next subsection), after which execution state from the EE-to-resume is swapped onto the idle cores.

The EECS process is initiated exclusively by an external EECS button that we add to the device, which provides the only way to flip the REE bit. The button is connected directly to the CPU via a dedicated EECS interrupt line. By bypassing the GIC for EECS interrupts, we are able to treat the GIC as a separate logical unit from the EECS interrupt system.

EECS interrupts are granted the highest interrupt priority (above both IRQ and FIQ), and they cannot be masked. Furthermore, EECS interrupts and the ensuing context switches are handled fully in hardware. Thus, even a fully-compromised software stack in one EE cannot prevent an EECS to the other EE.

4.1.2 Register Scratch Devices

Because we seek to save execution state upon EECS, an EECS necessarily involves saving register state. We therefore supply a register scratch device (RSD) for each EE. For example, upon an EECS from the REE to the TEE, the CPU would save all register state to the REE RSD and subsequently load register state from the TEE RSD.

4.2 Virtualized Hardware Components

4.2.1 CPU and GIC

Upon receiving an EECS interrupt from the EECS button, the CPU halts execution of the current EE, masks all interrupts, and propagates the EECS signal to its attached peripherals. Manufacturers may choose to provide a short grace period to handle already pending interrupts on the IRQ and FIQ lines if desired. Once this optional period passes, the CPU saves all register state (including that of the GIC) to the relevant RSD as described above, and then flushes all caches in order to prevent cache side-channel attacks. Swapping off an EE is therefore somewhat similar to a traditional "hibernate to disk," in that pending interrupts need to be drained and per-core execution state must be saved.

The CPU then loads register state from the newly-active RSD, and subsequently waits for each peripheral to respond with a binary success or failure EECS signal. If any failure signals are received, the CPU resets and boots the EE that the user was attempting to load. If all signals are successes, the CPU flips the REE bit and begins executing the new EE according to the freshly-loaded register state.

Note that if the system contains a separate graphics processing unit (GPU) from the CPU, the CPU is responsible for propagating the EECS interrupt to the GPU and processing the GPU's success/failure response signal. The GPU must accordingly be modified to save its register state to the RSD, flush its caches, load EE-to-resume state, and send a success signal to the CPU.

4.2.2 Memory System

At phone manufacture time, the manufacturer burns partitions into the MMU such that the REE and TEE only see virtualized views of RAM and the HD – namely, their respective partitions. Because device caches are flushed on EECS, EE memory isolation is enforced at each level of the memory hierarchy.

Tee-Bone's design is agnostic to the fraction of RAM and disk that manufacturers allocate to each EE. However, in the intended use case, the REE will be allocated significantly more disk and memory than the TEE – TEEs are intentionally designed to have a small threat surface and use minimal resources.

In our setup, RAM and disk are fully-virtualized. Thus, the "physical" addresses that each EE sees are in fact intermediate physical addresses (IPAs). Ultimately, these IPAs are translated to physical addresses by a fixed "base and bounds" hardware policy, stemming from the manufacturer-defined allocations for TEE and REE memory.

Both RAM partitions maintain power while the physical CPU is awake, meaning they do not need to be written to persistent storage upon receiving an EECS signal from the CPU. Because conventional CPUs keep all of physical RAM powered while awake, this should not cause an increase in

power consumption.*

Finally, both RAM and the HD flip an internal REE bit upon receiving an EECS signal, indicating their newly active partition and allowing them to reject IO requests to the inactive partition. The devices then respond to the CPU with a success signal to complete their portion of the EECS process.

4.2.3 NIC

The NIC must ensure that both outgoing and incoming network traffic for a particular EE are not visible to the other EE. Such isolation is primarily achieved via an internal REE bit, as well as distinct MACs for the REE and TEE. The NIC's internal REE bit controls which MAC the NIC currently binds to. The NIC stamps all outgoing traffic with the MAC of the currently active EE, and only accepts incoming traffic if the packet's destination MAC matches that of the currently active EE. Since the inactive EE cannot receive new incoming packets and cannot generate new outgoing packets, the physical network link is only used by one EE at a time. To prevent inter-EE information leakage, all NIC caches are flushed on EECS.

4.2.4 Screen

The screen is perhaps the simplest of all architectural changes. We add a second framebuffer to the graphics system, therefore providing one framebuffer per EE. Only the active EE's frame buffer will

*To decrease power consumption, it is possible that the newly inactive RAM could be written to its relevant disk partition, either synchronously or asynchronously, and then put to sleep. However, we have chosen not to include this as part of design to increase EECS speed.

be physically-attached to the screen at any given time, ensuring that the inactive EE cannot influence screen content to deceive the user.

On EECS, the current screen state is saved in the active EE's buffer, and then the active buffer is detached. The screen is cleared of all content at this time, followed by the new buffer being attached to the screen and its content loaded. The screen finishes this process by sending a success signal back to the CPU.

4.3 Boot Process

Upon reset, the physical CPU performs Power-On Self-Tests (POSTs) and registers attached physical devices, per usual. At this time, the CPU takes note of the EECS button state, setting its internal REE bit accordingly. The CPU then boots the active EE. During the boot, EE software is only presented with its virtualized view of devices, thus remaining completely unaware of the presence of another EE.

4.4 EECS Process

With the relevant hardware components described, here we provide an integrated overview of the EECS process. As stated, the process is only initiated by the user pressing the EECS button, requiring explicit user consent to switch EEs. Trusted, read-only microcode is responsible for halting execution and propagating the EECS signal to its peripherals. Once all devices (including the CPU) have saved relevant state, flushed internal caches, and flipped their internal REE bits to reflect the new EE,

the CPU can begin execution of the new EE. The process is pictured in Figure 4.2

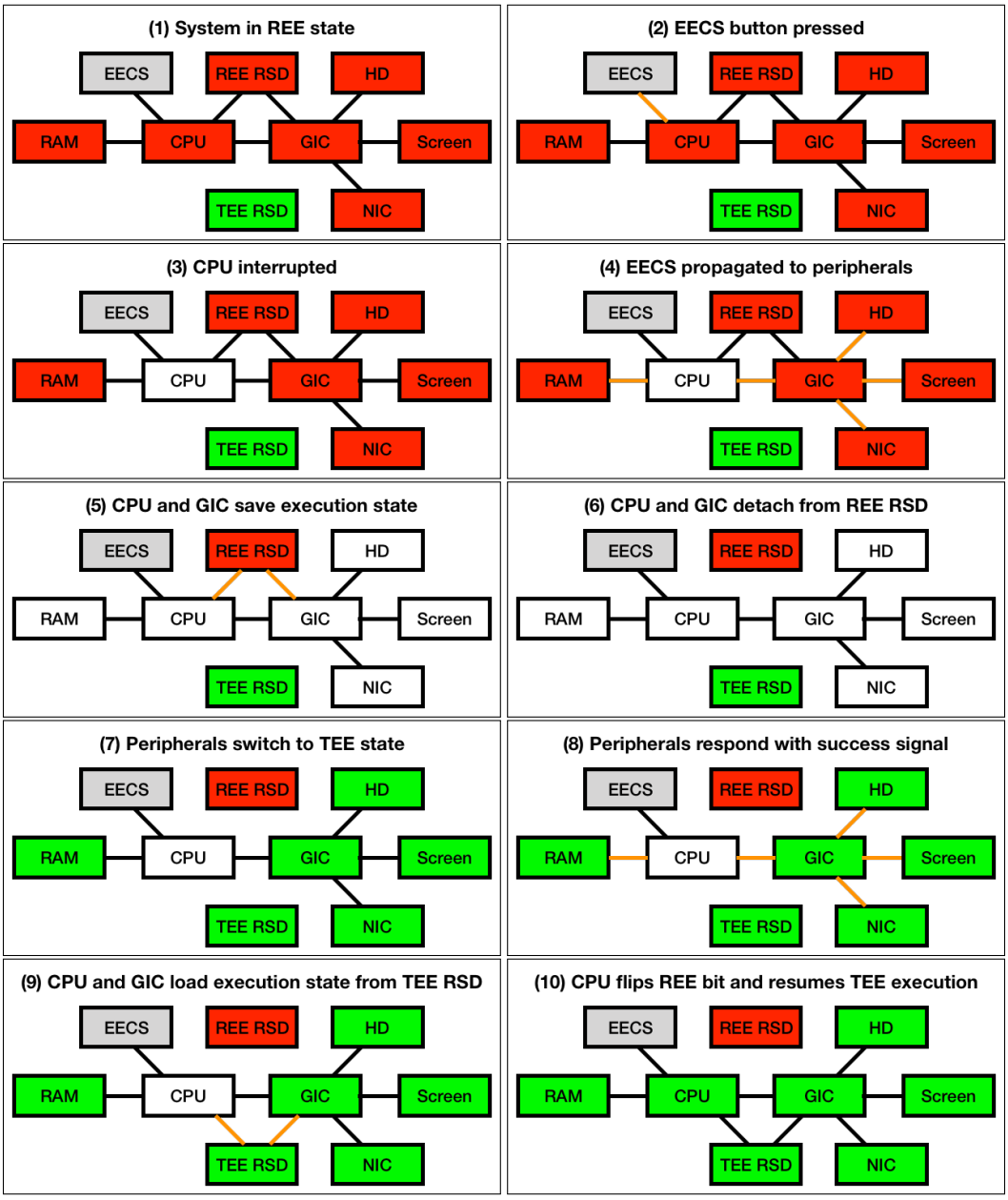

Figure 4.2: The steps taken in an EE context switch (EECS) from REE to TEE. While the REE is running, the user presses the EECS button to suspend the REE and awaken the TEE. Red is used to indicate REE state, green is used for TEE state, gray is used for the EECS button, and white is used for components in transition.

5

Implementation

As a proof-of-concept (PoC), Tee-Bone is currently being implemented atop of QEMU's AArch64 emulator. QEMU provides full emulation of ARM hardware boards and is open-sourced, allowing us to implement our proposed hardware modifications and simulate the behavior of each EE.[23] The system is expected to be completed in August 2018. Here we provide the details of our implementation thus far, and specify the remaining work to be done.

5.1 QEMU Setup

Tee-Bone runs on a modified version of QEMU's ARM "virt" machine. This machine is recommended by QEMU for running ARM Linux systems, includes support for 64-bit binaries, and provides the greatest hardware component flexibility of available QEMU boards.

Admittedly, the virt board does not support graphics out-of-the-box. However, we feel that the aforementioned benefits of the virt board outweigh this drawback, especially given that graphics virtualization support is not crucial to our PoC. Accordingly, our setup does not implement any aspects of our virtualized screen design, leaving this task to future development.

Our implementation adds QEMU command line arguments for specifying new Tee-Bone-specific fields. These fields include the TEE kernel, the optional TEE init root directory (initrd), TEE volatile and persistent memory sizes, and initial REE bit position.

With the new fields successfully parsed and loaded into QEMU, we began modifying the QEMU boot process to take these values into account. In particular, the REE bit determines which virtual hardware components are presented to the software stack during board initialization, as well as which of the ROS or TOS images is booted.

We are using a 64-bit Debian Linux image as our ROS, as we found ample guidance and support for running such an image on ARM QEMU.[22] For the TOS, we simply obtained a separate copy of the image. While Debian's relatively large and general-purpose design is inadvisable for production-quality TOS code, our PoC is focused on the ability to switch between two EEs at the hardware level. In other words, our PoC is ambivalent to the software running on top of our modified hardware, so

long as it is compatible.

In fact, due to the slow execution Debian in ARM QEMU – running atop of an x86 Ubuntu Linux VM on a host MacBook Pro – we switched to working with lightweight Buildroot AArch64-GNU-Linux kernels and initrds under certain circumstances. Namely, for testing in which persistent storage support is not needed, the lightweight software increased development efficiency.

Nonetheless, in the long run, we can build upon these basic software setups to better simulate a smartphone production environment. Ideally, we will use Android for the ROS and a small, security-oriented Linux flavor such as Alpine for the TOS.[1,2]

5.2 Hardware Partitioning and Virtualization

5.2.1 CPU and GIC

We chose to emulate a Cortex-A53 for our system's processor for 2 primary reasons. The first of these is that the A53 serves as the base design for a wide variety of smartphone processors, making its usage applicable to smartphone development. The second of our reasons is that unlike the slightly more modern A57, the A53 executes all instructions in-order and is thus not vulnerable to the recent Meltdown and Spectre attacks that rely on out-of-order execution.[14,15]

The A53 contains L1 instruction and data caches per core, a unified L2 cache shared amongst all cores (optional), a main TLB with 512 entries, and a micro TLB with 10 entries. Because we value the prevention of information leakage via cache side channels over performance, we flush each of these caches/buffers on EECS.

As for state that is saved upon EECS, QEMU's ARM CPU representation contains 3 structures of particular interest: `CPUState`, `ARMCPU`, and `CPUARMState`. `CPUState` is a QEMU-level representation of state common amongst all emulated CPUs, regardless of architecture. `ARMCPU` is a child of `CPUState` that adds the architectural features particular to ARM processors. One of these fields is for an environment of type `CPUARMState`, which defines the register configuration for ARM processors, as well as supplementary data such as boot info and GIC register state.

At a high level, the CPU metadata contained in `CPUState` and `ARMCPU` that controls emulated-CPU behavior – data such as the number of virtual cores and pointers to the address spaces for each EE – must be saved on EECS. With regards to real architectural details, all CPU and GIC registers can be used by either EE. Accordingly, we save the entire `CPUARMState` struct to the EE's RSD to preserve program state upon EECS. We then overwrite the CPU registers with the new EE's register state to prevent information leakage and maintain program correctness.

5.2.2 Memory System

The QEMU memory system does not correspond to a particular memory device, but rather is an internal, abstract representation of memory. In particular, memory in QEMU is a tree of memory containers and/or `MemoryRegion` objects, beginning with a root node. Specific memory behavior is emulated by programming the memory controller – for example, in its translation of virtual addresses to physical addresses.

As mentioned, our base and bounds IPA to physical address translation policy is fixed in the memory controller by the manufacturer. In our current simplistic setup, the machine uses 3 GB of

physical RAM. We present the first GB of physical memory to the TEE as IPAs `0x0` to `0x3FFFFFFF` via a dedicated `MemoryRegion` object. Thus, TEE IPA to physical address translation does not require a shift. On the other hand, we present the second and third GBs of physical memory to the REE as a separate `MemoryRegion` object, spanning IPAs `0x0` to `0x7FFFFFFF`. In this case, each REE IPA is translated to a physical memory address by adding `0x40000000` (1 GB) to it when processed by the memory controller. This is pictured in Figure 5.1.

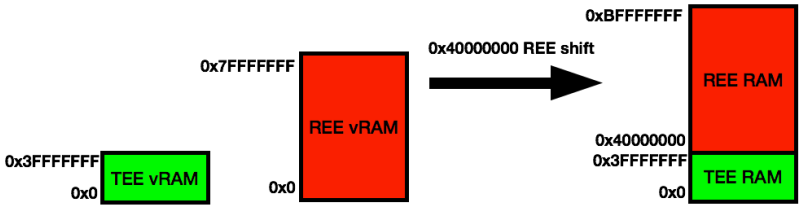

Figure 5.1: From left to right, the IPA address space for virtual TEE RAM, the IPA address space for virtual REE RAM, and the physical address space for all of physical RAM in our current implementation. Note that REE IPAs are shifted by 1 GB when translated to physical addresses.

Such a setup provides separate address spaces to the REE and TEE, allowing both EEs to map a virtual address to the same IPA, under the pretense that the underlying memory controller will shift the REE address to maintain EE memory isolation. If an EE attempts to use an address outside of its IPA range, the bus logic triggers a fault. We use a similar system to forward disk reads and writes to the correct partition.

5.3 Board Initialization and Boot Process

During board initialization, we allocate the REE and TEE virtual RAM regions and initialize them according to the policy laid forth in the previous section. Notably, we present separate flat device trees (FDTs) for the REE and TEE that expose a suggested platform hardware description to each kernel. While an exploited kernel could ignore the FDT, our hardware will nonetheless fault if unsupported behavior (such as a memory access outside of the active EE) is attempted. We additionally provide separate `arm_boot_info` structs to the REE and TEE, such that the REE and TEE can load distinct software stacks.

Once board initialization is complete, we save the initial register state of each EE to its RSD such that it can be loaded upon EECS. We then note the position of the REE bit and boot the corresponding kernel per QEMU's standard boot process. The other kernel can be booted on its own virtual hardware in a likewise fashion upon first EECS.

5.4 Emulating an EECS

The EECS button is emulated via the QEMU monitor, a wrapper around the emulated system that can send remote commands to the running machine instance. Thus, to emulate the "pressing" of the EECS button, the QEMU user issues the new command "qemu_system_eecs" via the QEMU monitor. Like the EECS button, this monitor command is the sole source of EECS initiation.

The EECS monitor command traps to an EECS event handler that interrupts the emulated CPU. In particular, a dedicated EECS signal triggers our hardware-defined EECS process, regardless of

IRQ and FIQ interrupt state. Such a setup simulates having a dedicated interrupt line for EECS as specified in our design, motivated by our desire to bypass the GIC in order to simplify its virtualization.

5.5 Remaining Work

Despite our progress, our implementation of Tee-Bone remains incomplete due to four key remaining tasks.

1. We have not finished full virtualization of our board's memory map; we only currently virtualize RAM and disk memory. In order to ensure complete virtualization of the memory system, we must virtualize each section of the physical memory map, including the regions listed in 5.1.
2. We have not yet implemented NIC virtualization.
3. We can improve upon the implementation of EECS by using existing ARM emulations to "halt" machine execution, as opposed to using QEMU infrastructure to artificially freeze machine state.
4. We will need to complement these tasks with a thorough evaluation of system behavior and resources. For example, we must measure the number of cycles required for EECS, and we must gauge the amount of space required for our new hardware devices. Most importantly, we have to test EECS corner cases (such as switching during a disk write or FIQ interrupt) to ensure that EE isolation is maintained and proper system functionality is preserved for both EEs.

Memory Region	Role
CPU Peripherals	Allocated for port-mapped CPU peripherals
Flash	Provides read-only initial boot code, which can differ by EE
GIC	Used to interact with the interrupt controller
GPIO	Enables communication via general-purpose IO ports
MMIO	Space for memory-mapped IO devices
PCIE	Enables communication with PCIE devices
Platform Bus	For sending and receiving data on the platform bus
Secure RAM	Used for secure world of TrustZone

Table 5.1: Remaining board memory regions to virtualize

6

Related Work

6.1 Intel SGX

Intel Software Guard Extensions (SGX) is another hardware-assisted EE isolation technology.[10] Unlike Tee-Bone and ARM TrustZone, SGX allows for an essentially arbitrary number of trusted software "enclaves" to be created within untrusted processes for secure computation.

Each enclave has a region of processor-reserved memory (PRM) for code and data in physical

RAM – only enclave code and data can access this PRM. The enclave treats all code outside of it as untrusted, including the OS. SGX enforces the integrity of enclave memory pages via MACs and encryption keys, which are isolated from the OS at the hardware-level. Nonetheless, because the enclave relies on the OS for privileged services such as I/O, the OS can launch denial-of-service attacks on the enclave.

SGX hardware uses cryptographic techniques to record a tamper-proof history of the enclave's initialization, meaning that remote clients can leverage this history to perform remote attestation with the enclave code. While such a defense prevents the execution of modified enclave code, it does not protect against DoS. Namely, an OS that is compromised before enclave creation could prevent the enclave's creation or interfere with the initialization of the enclave's code and data.

The modern x86 architecture is intrinsically complicated, and the fact that the architecture provides backwards compatibility for decades of older x86 features makes this complexity even worse. Ultimately, SGX is designed to improve the security of the x86 architecture without making major changes, but such a setup leaves the complexity of x86 hardware (and therefore, its large threat surface) in place.

Indeed, SGX has been shown to lack resistance to a wide variety of software side-channel attacks. For example, SGX does not disable hyper-threading, meaning enclave code that runs on a particular logical core may share functional units with untrusted code running on a different logical core. While each thread executes on its own logical core, the logical cores share functional units within the physical core. By combining micro-architectural knowledge of functional unit and scheduling behaviors with statistics garnered from the processor's performance counter, an attacker can use the

malicious thread to discern the instructions and memory access patterns being used by the enclave.

Relative to SGX, Tee-Bone does increase context-switch latencies between EEs, since entire EEs must be suspended and resumed. However, in Tee-Bone, trusted code and untrusted code are unable to concurrently share hardware resources. Therefore, side-channels attacks through architectural or micro-architectural features are impossible.

6.2 AMD SVM/Intel TXT

Both AMD Secure Virtual Machine (SVM) and Intel Trusted eXecution Technology (TXT) allow untrusted code to launch up to 64 KB of trusted code in a hardware-isolated environment by invoking a special instruction. The special instruction (SKINIT) takes the memory address of a trusted code snippet – known as the secure loader block (SLB) – as its argument.[17]

When an SKINIT is issued, the hardware prohibits DMAs to SLB memory pages until the SLB has finished execution. Interrupts and debugging access are also disabled to prevent outside REE processes from taking control. Finally, to prevent execution of modified code, the processor supports traditional PCR-extend-style attestation of the SLB by remote parties.

Tee-Bone offers a simpler isolation model that exposes a smaller threat surface than SVM/TXT. For example, inter-EE DMAs are impossible in Tee-Bone since each EE is unaware of the other's memory pages. Additionally, because the REE never executes simultaneously with the TEE, Tee-Bone does not need to disable interrupts during TEE execution. Finally, much like in the case of SGX enclave initialization, a malicious OS can disrupt the SLB initialization process. Tee-Bone, on

the other hand, does not rely on the REE to invoke the TEE, and therefore is not susceptible to such an attack.

6.3 ARM Trusted Firmware

Though still under development, the ARM Trusted Firmware project aims to provide an open-source reference implementation of TEE software, including an EL3 secure monitor. By providing a standardized software implementation of various ARM interfaces, Trusted Firmware has the potential to reduce the likelihood of bugs in inter-EE management across ARM-based platforms. For example, Trusted Firmware implements the SMC convention to standardize inter-EE communication, as well as the power state coordination interface (PSCI) and the system management control interface (SCMI) to standardize inter-EE power and performance management.[8,9]

We agree that a standardized and open-source reference implementation of EL3 software can limit software-based EE isolation bugs. However, we maintain our concerns about the complexity of TrustZone's current EE isolation model, and ultimately note that the Trusted Firmware project does not repair the systemic issues addressed in this paper – namely, those arising from software-based EE isolation management.

6.4 GlobalPlatform TEE System Architecture specification

GlobalPlatform is a not-for-profit association dedicated to developing specifications for the secure management of devices. Of particular interest to our research is GlobalPlatform's work in standard-

izing interfaces between the TEE and REE, such as those used for inter-EE communication and UI management.[12] Open-source TEE projects such as TrustZone-based OP-TEE and hardware-independent Open-TEE already conform to these standards.[18,21] Because standardization provides a well-defined set of TEE entry points and security requirements, we believe these efforts are well-founded. However, as we argue in this thesis, allowing *any* inter-EE interfaces is detrimental to security and unnecessary to support common usage patterns for mobile applications.

7

Conclusions

To our knowledge, Tee-Bone is the first smartphone EE isolation system to be based entirely in hardware mechanisms. Tee-Bone eliminates all software-level EE isolation vulnerabilities, including DoS, confused deputy, and hijacking attacks. Our system can be easily integrated into future ARM smartphones due its basis in and minimal changes to ARM TrustZone.

In August 2018, we will release our PoC implementation of Tee-Bone atop the emulated ARM

hardware provided by QEMU. While our PoC will establish the security of our EE isolation model, future research on Tee-Bone will need to be performed on physical boards in order to effectively evaluate our system's performance impacts. Indeed, at such a time, further performance optimizations must be explored and potentially introduced into Tee-Bone's architecture, so long as they do not compromise the security of the model.

Ultimately, prevention of inter-EE subversion at the software level improves the underlying security of sensitive applications. Should Tee-Bone be implemented on commodity ARM-based smartphones, device manufacturers would have the confidence to open their proprietary TEEs to third-party applications, spurring software development in high-security fields such as government and finance.

References

[1] Alpine-Dev-Team (2017). Alpine linux. https://alpinelinux.org/.

[2] Android-Dev-Team (2017). Android. https://www.android.com/.

[3] ARM (2009). Arm security technology: Building a secure system using trustzone technology.

[4] ARM (2015). Arm cortex-a series: Programmer's guide for armv8-a (version 1.0).

[5] ARM (2016). Smc calling convention: System software on arm platforms.

[6] ARM (2017a). Arm architecture reference manual: Armv8, for armv8-a architecture profile.

[7] ARM (2017b). Arm generic interrupt controller architecture specification.

[8] ARM (2017c). Arm power state coordination interface platform design document.

[9] ARM (2017d). Arm system control and management interface platform design document.

[10] Costan, V. & Devadas, S. (2016). Intel sgx explained. *Cryptology ePrint Archive*.

[11] Ekberg, J. (2014). The untapped potential of trusted execution environments on mobile devices. *IEEE Security and Privacy*.

[12] GlobalPlatform (2017). Globalplatform made simple guide: Trusted execution environment (tee) guide. https://www.globalplatform.org/mediaguidetee.asp.

[13] Jung, Y., Kim, H., & Kim, S. (2014). An architecture for virtualization-based trusted execution environment on mobile devices. *IEEE 11th Intl Conf on Autonomic and Trusted Computing*.

[14] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., & Yarom, Y. (2018). Spectre attacks: Exploiting speculative execution. https://spectreattack.com/spectre.pdf.

[15] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., & Hamburg, M. (2018). Meltdown. https://meltdownattack.com/meltdown.pdf.

[16] Machiry, A., Gustafson, E., Spensky, C., Salls, C., Stephens, N., Wang, R., Bianchi, A., Choe, Y., Kruegel, C., & Vigna, G. (2017). Boomerang: Exploiting the semantic gap in trusted execution environments. *Proceedings of the Network and Distributed System Security Symposium*.

[17] McCune, J., Parno, B., Perrig, A., Reiter, M., & Isozaki, H. (2008). Flicker: An execution infrastructure for tcb minimization. *ACM SIGOPS Operating Systems Review - EuroSys '08*.

[18] McGillion, B., Dettenborn, T., Nyman, T., & Asokan, N. (2015). Open-tee – an open virtual trusted execution environment. *IEEE Trustcom/BigDataSE/ISPA*.

[19] Mijat, R. & Nightingale, A. (2011). Virtualization is coming to a platform near you: The arm architecture virtualization extensions and the importance of system mmu for virtualized solutions and beyond.

[20] Ngabonziza, B., Martin, D., Bailey, A., Cho, H., & Martin, S. (2016). Trustzone explained: Architectural features and use cases. *IEEE 2nd International Conference on Collaboration and Internet Computing*.

[21] OP-TEE-Dev-Team (2017). Op-tee: Open platform trusted execution environment. https://www.op-tee.org/.

[22] pm215 (2017). Installing debian on qemu's 64-bit arm "virt" board. https://translatedcode.wordpress.com/2017/07/24/installing-debian-on-qemus-64-bit-arm-virt-board/.

[23] QEMU-Dev-Team (2017). Qemu: The fast processor emulator. https://www.qemu.org.

[24] Sabt, M., Achemlal, M., & Bouabdallah, A. (2015). Trusted execution environment: What it is, and what it is not. *14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*.