Πανεπιστήμιο Πειραιώς

University of Piraeus

M.Sc. Digital Systems Security

A Security Evaluation of TrustZone Based Trusted Execution Environments

SUPERVISOR: XENAKIS CHRISTOS
(xenakis@unipi.gr)
AUTHOR: KOUTROUMPOUCHOS NIKOLAOS
(nikoskoutr@ssl-unipi.gr)
UNIVERSITY OF PIRAEUS 06/05/2019

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Prof. Christos Xenakis who willingly gave time and effort in order to provide helpful input during the planning and development of this work.

Furthermore, I would also like to thank Prof. Christoforos Dadoyan for his help and advice and for being there when I needed him for any questions or problems I had during the research and the writing of this work.

Last but not least I would like to thank my patient and supporting significant other and my friends and family.

ABSTRACT

Trusted execution environments are marketed as a secure and robust solution that can greatly enhance the overall security of the system it is installed on. In this work we aim to analyze the architecture of trusted execution environment as a concept while also seeing in detail the implementation aspects of the TrustZone based trusted execution environments. Furthermore we aim to evaluate the actual security provided by this solution through an exploration of the characteristics of the various attacks and vulnerabilities that have been recorded in the bibliography.

TABLE OF CONTENTS

1 Introduction 6
2 TEE Background 7
2.1 GlobalPlatform 7
2.2 TEE High Level Security Requirements 8
2.3 TEE Hardware Architecture 8
2.4 TEE Software Architecture 13
3 ARM TrustZone 18
3.1 NS Bit Memory Separation 18
3.2 Processor Architecture 19
3.2.1 Securing the level one memory system 20
3.2.2 Secure Interrupts 22
3.3 Multiprocessor Support for TrustZone 23
3.4 ARM TrustZone Software Architecture 24
3.4.1 Software Architecture 25
3.5 Booting Process of a Secure System 27
3.5.1 Boot Sequence of a TrustZone Enabled System 27
3.5.2 Secure Boot 28
3.6 Monitor Mode 30
4 Vulnerabilities of TrustZone Based TEEs 31
4.1 Attacks 31
4.1.1 TrustZone Code Execution [11] 31
4.1.2 TrustZone Kernel Exploitation [12] 31

4.1.3 Extraction of Master keys from TrustZone [13] 32

4.1.4 BOOMERANG Vulnerability [7] 32

4.1.5 Downgrade Attack [14] 34

4.1.6 Cache Timing Attacks 35

4.1.7 CLKscrew [19] 36

4.1.8 BADFET [20] 37

4.2 Attack Comparison 38

5 Discussion 40

6 References 42

List of Figures

Figure 2-1: Potential TEE Hosting Components 9

Figure 2-2: Possible Architectures of TEE Enabled Systems 11

Figure 3-1: The Relationship Between the Normal World, the Secure World and the Monitor Mode 20

Figure 3-2: TrustZone Multiprocessor Support 24

Figure 3-3: Software Architecture of the Normal World OS and the Secure World OS ... 26

Figure 3-4: Boot Sequence of a TrustZone Enabled System 28

Figure 4-1: The Boomerang Attack 33

Figure 4-2: The CLKscrew Attack [19] 37

LIST OF ABBREVIATIONS

API - Application Programming Interface

CPU - Central Processing Unit

I/O - Input/Output

MITM - Man in the Middle

REE - Rich Execution Environment

SoC - System on Chip

TEE - Trusted Execution Environment

1 INTRODUCTION

As the quantity of internet connected devices increases, their security is becoming a difficult task. Trusted Execution Environments (TEEs) where introduced, in order to ease this difficulty and put another brick on the wall of device security. A TEE provides an isolated environment, that can be utilized by processes to offload sensitive tasks and store private data. The interesting thing about this solution, is that it provides the isolation through hardware enforcement, each processor core can be reserved for the execution of a TEE application when needed and released afterwards; the same applies for storage, a specific physical address space is reserved just for the TEE operating system, the system will allow nothing else to access this memory space.

TrustZone technology, is a family of hardware and software components that a TEE can be implemented upon. It is a hardware-based approach to security, that enables the separation of two worlds of execution, the trusted and the non-trusted worlds. The trusted world is where the TEE Operating System and its applications exist and store data, and the non-trusted world is where the normal operating system of the device exists. Entities that run on the non-trusted world do not have access to trusted world resources, but trusted applications, can access non-trusted entities. TrustZone promises perfect isolation and security, but research after research, keep finding holes and manage to break the world barrier. Is TrustZone truly secure after all?

The purpose of this thesis, is to find, review and assess all the published vulnerabilities of TrustZone implementations. A key question to be answered is: Can TrustZone be blamed alone, or the developers of the specific instance have some share of the fault too? In the next chapter, all the background knowledge of TEEs and TrustZone is going to be presented.

2 TEE BACKGROUND

A trusted execution environment on a computing device allow for the invocation of trusted applications without obstructing the normal computing environment. In addition, trusted applications running on this environment can securely interact with each other in such a way that other non-trusted entities cannot access the trusted applications and their resources. [1] Therefore, the trusted execution environment provides two basic services on a high level:

- Isolated execution of trusted applications with restricted interprocess communication.
- Unobstructed execution of normal applications within the same computing environment.

2.1 GLOBALPLATFORM

GlobalPlatform is a standardization body that took it upon itself to define the ideal hardware security guard that will be used on mobile phones. In order to do so it took into consideration some security requirements that where specified by the Open Mobile Terminal Platform which aimed to define a "trusted environment"; GlobalPlatform improved upon this specification and finally presented the final product that was named "trusted execution environment". [2] [3] This product immediately caught the attention of major organizations which backed the development of this technology and soon the trusted execution environment became a well-defined security component that exists in most mobile phone and computer processors today.

GlobalPlatform provides a wide range of specifications that define every aspect of the trusted execution environment, from hardware level to the final API and how it should be implemented. It also provides certifications for compliance with the specification and organizes conferences and workshops in order to spread the knowledge for trusted execution environments. The specification that GlobalPlatform created is widely accepted

as the de facto standard with most vendors implementing it in their devices with minor exceptions.

2.2 TEE High Level Security Requirements

The primary purpose of the TEE is to protect and isolate its resources from other environments present on the ecosystem that the TEE belongs. This isolation is enforced through hardware mechanisms that are not controllable by non-secure environments. Apart from the software protection, the TEE should also be protected against some physical attacks such as side channel attacks, but intrusive techniques are not in the scope of the TEE security. Within the context of hardware security, the ecosystem that hosts the TEE should remove or disable any debug hardware interfaces with direct access to the TEE. The final security requirement that the specification makes is that the TEE must be instantiated through a secure boot process, that produces a chain of trust and ensures that the devices is properly booted without any tampering. The secure boot process provides guarantees to the authenticity and the integrity of the device and all its components.

2.3 TEE Hardware Architecture

The trusted execution environment (from now on referenced as TEE) is a component of a computing system, and as such, it should be defined where it belongs. The TEE typically resides either within the SoC or as an external security processor that connects directly to the system bus. This is shown in **Error! Reference source not found.** where the blue entities represent these two possible points of TEE installment. When the TEE is an external entity, the division between the two worlds (trusted and untrusted) is obvious, as the TEE is an independent entity that interacts with the system. In the other case, where the TEE is part of the SoC, then the hardware separation is not so apparent. More specifically, the TEE can either share the SoC CPU and use it securely only when needed

or have separate CPU core(s) just for the trusted operations. [4] Some hardware implementations are shown in **Error! Reference source not found.**.

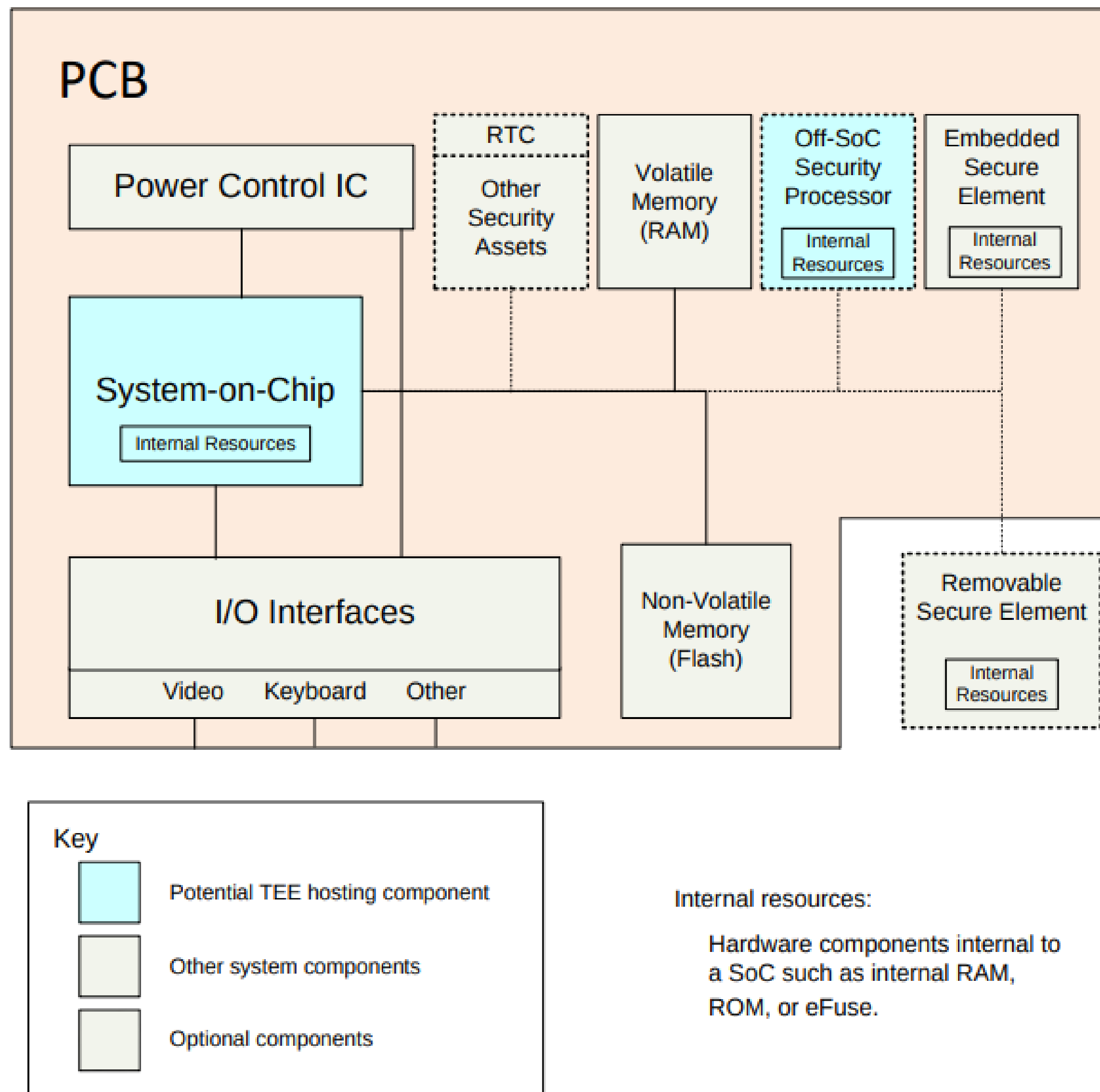

Figure 2-1: Potential TEE Hosting Components

The TEE manages three classes of resources [5]:

- **In-package resources:** These resources exist only within the TEE and are considered protected from any adversary. Communication between these

resources is also considered physically secure and as such, there is no need to encrypt it.

- **Off-package, cryptographically protected resources:** In order to extend the in-package resources, a TEE can utilize external resources that can be deemed trustworthy with the usage of cryptography. The protection of these memory areas is achieved through proven cryptographic methods, with the TEE being the only entity that holds the decryption capabilities. Although these resources are cryptographically secure, since they exist out of the package, the communication between the TEE and these resources can be vulnerable to man-in-the-middle (MITM) attacks. These off-package resources include the trusted replay-protected external non-volatile memory areas, and the trusted volatile memory areas
- **Exposed or partially exposed resources:** These resources are both off-package and not cryptographically protected. Some examples are: trusted DRAM-based buffers, trusted screen frame stores and input/output (I/O) devices. All of the aforementioned examples require the isolation provided by the TEE but do not require any encryption.

These resources can and will be shared between the TEE and the REE, so as to provide the intended functionality. As expected, any TEE resource can only be accessed by the TEE due to the direct (hardware) and indirect (off-package encryption) isolation that it provides. Some of these resources, can be accessed by the REE through API entry points or other services that the TEE exposes through the TEE Client API. [6] On the other hand, REE resources are always accessible by both the TEE and the REE, with the TEE having direct access to any memory location of the REE [7]. Depending on the implementations shown in **Error! Reference source not found.**, the resource sharing scheme is different; for example, the PCB A has complete isolation of the TEE and does not facilitate any resources from the REE.

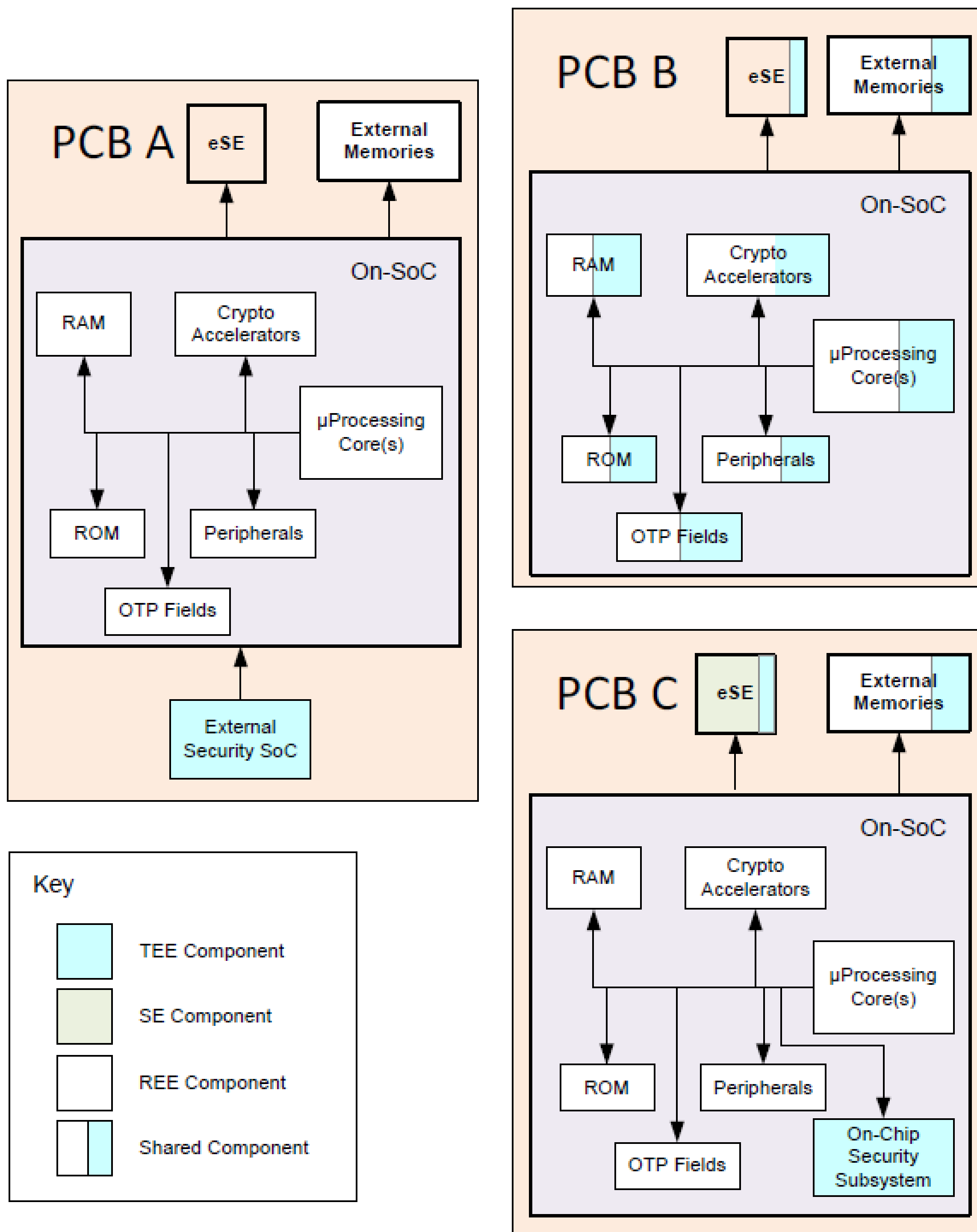

Figure 2-2: Possible Architectures of TEE Enabled Systems

From an abstract point of view, the TEE aims to have two worlds, the trusted and the untrusted world. When the untrusted world is in charge, then the system operations and

resources should be considered untrusted and accessible by anyone. The untrusted world hosts the REE (Rich Execution Environment), which provides an operational environment for the host operating system and applications to run on. The trusted world, on the other hand, exists with the purpose of hosting sensitive functionalities that need a high level of trust and authorization. Only specific components have access to the hooks provided by the trusted world and they need to be authenticated first before the operation. The trusted world is not just a fence that protects its components, it provides a usable and programmable environment for developers to deploy their sensitive applications. It has its own libraries with the aim of reducing the secure code base and assure that the TEE is used only when absolutely needed.

The world change takes place with the help of an underlying call manager that is responsible for handling requests for transitions between the worlds. This manager is either a supervisor, or in the case of ARM TrustZone, the secure monitor. When an untrusted application needs to run sensitive functionalities that exist within the TEE, it issues a system call to the manager. If the information provided by the application is correct (authentication data, authorization data, ID of the called trusted application etc.), then the manager proceeds with the world change and gives control to the trusted world. On the other side, the trusted world validates the command it received and executes the appropriate commands. When the specified functionality is finished, then it issues a new system call to the underlying manager with all the relevant information (process ID of calling application, results etc.) and the manager returns control to the untrusted process along with said data.

The low-level communication between the REE and the TEE is handled by hardware drivers that are responsible for properly handling the messages to be exchanged, assessing their authenticity, validity, feasibility etc. As these drivers are only accessible by privileged users, communication services are installed that provide a low-level API for applications to use. The TEE Client API [6], the specification of this service, provides functions to accomplish a rich communication between the two entities. Furthermore,

there is a definition for the TEE Protocol Specification, that adds another layer of abstraction to the TEE Client API, this way, developers can provide a higher-level API for applications to utilize the TEE within. According to the specification of GlobalPlatform, the TEE TA Debug API and the TEE Management Framework Specification are both using the TEE Protocol Specification layer.

2.4 TEE SOFTWARE ARCHITECTURE

Secure Boot

As the core component of a TEE is trust, it firstly must be verified for its integrity. Only trusted and verified images should be run on the device to maintain the desired security level. The secure boot scheme uses cryptographic checks in order to assert the integrity of all the secure world images to be loaded. It is based on a concept that is called “chain of trust”, which is a waterfall model that aims to propagate trust through multiple layers of software. More specifically, the root of trust in this chain, is a hardware-bound cryptographic key that is used to check the integrity and authenticity of the first piece of software to be executed. After the first software is authenticated it then can be trusted to authenticate the next piece of software and so on, creating a chain of trusted software images, each authenticated by the previous trusted software. This chain leads up to the TEE images, which are always authenticated before they are trusted to perform any sensitive operation.

Secure OS

From the software perspective, the TEE requires some basic building blocks which are: the secure operating system, trusted applications, the rich operation system, normal applications, the message broker (hypervisor/ARM trusted firmware) and the management framework. More specifically, the secure operating system runs on the secure world and it is responsible for providing a basic set of operating system functionalities while also preserving a high level of security so as to conform with the

GlobalPlatform specification. The secure OS practically is the implementation of the secure part of the TEE specification, as it should employ any available technology (hardware and/or software) to protect its assets from the normal world, while also isolate trusted applications from each other. Some implementations of secure operating systems include Google's Trusty, Qualcomm's QSEE, Trustonic's Kinibi and Linaro's optee_os, with the most widely deployed and used being QSEE and Kinibi. All of the aforementioned secure operating systems are based on the ARM TrustZone technology, there are other operating systems that are based on other technologies like the Intel SGX and AMD SME/SEV.

Trusted Applications

The second component of the TEE software architecture that we are going explore, are the trusted applications (TA) or trustlet as they are called for convenience. Trustlets are pieces of software that run as applications within the trusted operating system and are developed either by first party or third-party developers. Each trustlet is run isolated from each other, with their resources protected in a manner that they cannot be attacked neither from the normal world nor from another trustlet. The primitives used to protect trustlets from each other are software/cryptography based, whereas the cross-world protection also utilizes hardware-based isolation technologies like TrustZone. The only entity that can possibly access another trustlets resources is the secure OS kernel in a manner that is identical to the normal operating system kernel accessing its user space applications.

The trustlets are powerful applications that are compiled from memory sensitive languages (C) and are also capable of accessing hardware memory addresses in some implementations. Moreover, they are software that is considered as trusted to do sensitive functions for the normal and the secure world. Because of this, most TEE implementations have chosen to secure the confidentiality, integrity and authenticity of the trustlets in multiple ways. The authenticity and the integrity of the trustlets is

protected by a signature on the hash of the trustlet binary that can be verified using a certificate chain that is rooted on a hardware bound public key. This way, an attacker cannot load a trustlet that is not verified by the hardware vendor. The confidentiality and integrity of the trustlet, or more correctly, the confidentiality and the integrity of the trustlet data is protected with a cryptographically secure storage that also has TEE binding functionalities which assert that the data cannot be cloned to other TEEs.

Secure Monitor

The Secure Monitor mode is responsible for the switch from one world to the other. It acts as a middleman between the secure world and the normal world and handles all requests between them. In most use cases, the functionality is similar to the context switch of any operating system, that is, it should correctly and safely store the initial context state, give control to the target context and finally restore the initially stored state. There are strict constraints when it comes to transitions from the normal world to the secure world, whereas the other way there is more flexibility. That is because the secure world holds sensitive information and has high privileges, so it is only logical that userspace normal world application access attempts are put under the microscope. This does not happen the other way around, something that may initially sound normal, but as we will see later, it rises a class of vulnerabilities that leverage the trusted world to attack the normal world.

As evident, the security of the secure monitor is of crucial importance, as it acts as a gatekeeper to the highly privileged and sensitive secure world. It is advised that the execution of secure monitor code is always run with interrupts disabled to minimize any malicious attempt to manipulate the execution flow of this sensitive piece of code. Moreover, the secure monitor is always run on the secure world, it only provides some external interfaces for the normal world to use, but it completely runs within the context of the secure world so as to provide the maximum possible level of security.

Normal World

The normal world is the normal operating system that is controlled by the end user. There are a few minor changes so that the normal world will incorporate the secure world in its design. As the focus of this work is the trusted world, we are going to focus only on the components of the normal world that are relevant. There are three common implementations for the normal world to access the secure world; by directly accessing the driver and using a middleware library, with the most common method being the latter.

TEE Driver

The drivers are kernel modules that are responsible for providing hooks for the userspace applications to execute privileged functions. Controlling any peripheral, requires kernel privileges that a simple application does not have, in order to enable usage of these peripherals, the kernel developer installs a set of drivers that aim to enable the usage of installed peripherals by normal users. This is the case for the secure monitor. Because the system calls to initialize a world switch, there always is a secure driver in the system so that the normal world can easily communicate with the secure world.

The driver is a simple block device, that can be written to and read from. Its purpose is to handle any trusted world access intent and either propagate it to the secure monitor or block it. Furthermore, it manages any allocated shared memory between the two worlds so as to provide unobstructed communication. Finally, and more importantly, the driver populates the system calls that are sent to the secure monitor to initialize a context switch. These calls are very important as they provide the largest attack surface to the trusted world as we will see.

TEE Library

The purpose of a library is to provide readily available functionality for other programs to use without the complexity of implementing said functionality. A core component of the TEE is the libraries and APIs it encloses within it that are divided across the two worlds. The secure world library provides the **trusted core framework API** that defines basic data

structures while also managing trustlet instances, inter-trustlet communication and memory. The **trusted storage API** has functions that store securely persistent or transient data objects in the trusted storage of the TEE. There is also, the **cryptographic operations API** that as its name implies, has hooks that map to all the supported cryptographic operations which include: symmetric and asymmetric cryptographic functions, hashing functions, MAC functions, authenticated encryption functions and key derivation functions. Finally, there is the **time API** that provides time related functions, the **arithmetic API** that manages arithmetic operations between TEE specific data types and the **peripherals API** which handles any external trusted peripherals that might exist.

On the other side, the normal world library aims to ease the process of calling trustlets and the communication between the two worlds. This library is much simpler that its trusted counterpart, as it does not implement any software logic, one can think of it as the window from the normal world to the trusted world. It has functions for context management, session management and command invocation which should be called whenever a normal program wishes to delegate a sensitive operation to a trustlet. Furthermore, it provides operations for the allocations and registration of shared memory spaces that will be used by both normal and trusted applications to exchange data. Finally, it defines common data structures and constants in order to provide a common language that the two worlds can use to communicate.

3 ARM TrustZone

ARM TrustZone [8] [9] [10], according to the official terminology is a system-wide approach to security for a wide array of client and server computing platforms, including handsets, tablets, wearable devices and enterprise systems." [4] As a feature, TrustZone can been seen as a special kind of hardware supported virtualization of CPU state, memory, interrupt signals and I/O data, with the purpose of isolation. This virtualization technology enables each physical CPU core to provide an abstraction of two virtual ones, orthogonally dividing the process state in two logical realms, or worlds as they are named: The normal world of the REE and the secure world of TEE. TrustZone can be seen as a technology that enables any part of the system to be made secure, while also sharing it with the normal world.

3.1 NS Bit Memory Separation

To achieve the world separation, the TrustZone technology employs some unique techniques with the most distinctive one being the NS bit. This bit is defined in the AMBA3 AXI bus protocol specification and it is an additional 33d bit to the 32-bit address space that is provided to both the secure and non-secure world and it is appended to all read and write bus messages of the system. If the bit is set to high, then the message is non secure while a low NS bit signifies that the message is to be accessed only by the secure world.

This extra bit works fine within the boundaries of the ARM made components, but when a peripheral that does not "speak" ARM receives such a message then it will probably not work properly. To this end, the AMBA3 specification defines a separate bridge from the AXI bus that has the NS bit to the APB bus that can work properly with any peripheral. This bridge acts as a gatekeeper that will not accept non secure messages when the NS bit is set to low. This way, the peripherals are separated from the normal world when

sensitive operations are executed (e.g. when a user types a password, the keyboard is bound to the secure world).

As with any address space, including those without TrustZone technology, care must be taken to ensure that the 33-bit address space is used in such a way that data remains coherent in all of the locations that it is stored, otherwise data corruption may result. If we consider the case where a Secure world master wants to access a non-secure slave that is cached. A design may implement either of the following choices:

- The master makes a Non-secure access to the slave.
- The master makes a Secure access to the slave and the Non-secure slave accepts the Secure transaction. The slave treats these accesses as Non-secure.

In the second design the hardware must support address space aliasing. In this aliased memory system, the same memory location appears as two distinct locations in the address map, one Secure and one Non-secure. As a result, it is possible to have multiple values representing the same data present in the cache simultaneously. For modifiable data this aliasing causes coherency problems; if one copy of the data is modified while the other exists in the cache you will have versions of the data but both will be different. System designers must be aware of potential data coherency problems, and must take steps to avoid them.

3.2 Processor Architecture

The current ARM processors that support the ARM TrustZone technology belong mainly in the Cortex-A family of ARM and a lightweight version of TrustZone can also be found in ARMv8-M, an indicative list of these processors is:

- ARM1176JZ(F)-S™ processor
- Cortex™-A8 processor
- Cortex-A9 processor

- Cortex-A9 MPCore™ processor
- Cortex-M23 processor
- Cortex-M33 processor

Each of the physical processor cores in these designs provides two virtual cores, one considered Non-secure and the other Secure, and a mechanism to robustly context switch between them, known as monitor mode. The value of the NS bit that is added to all bus messages is derived automatically from the virtual core that made the call. This way there is a seamless integration of the TrustZone technology in the system where non-secure cores can access only non-secure data and secure calls can see everything.

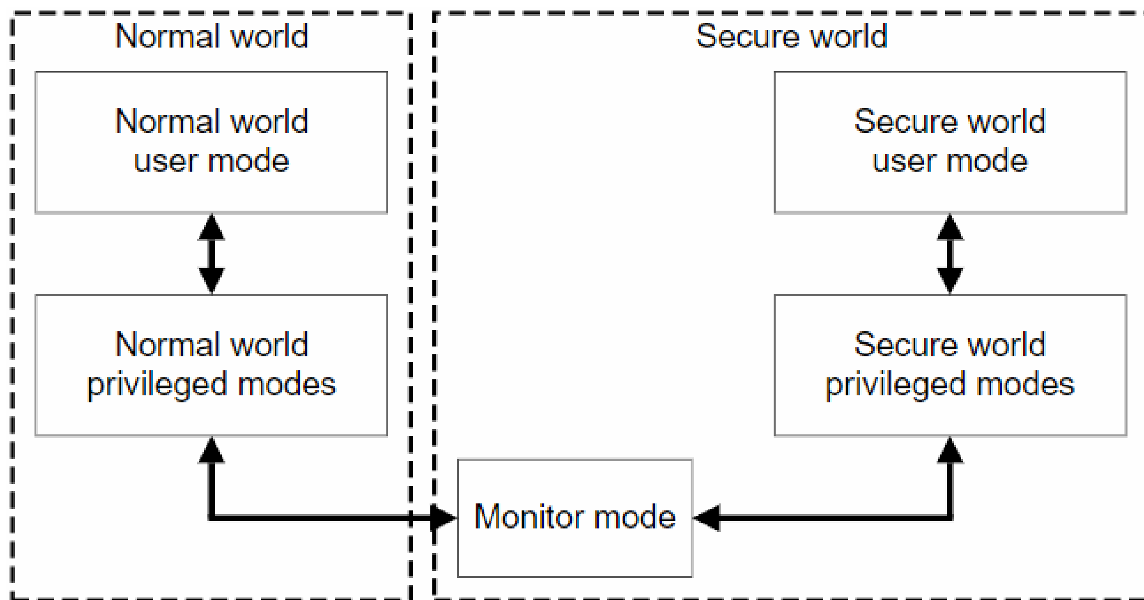

Figure 3-1: The Relationship Between the Normal World, the Secure World and the Monitor Mode

3.2.1 Securing the level one memory system

The memory infrastructure outside of the core separates the system into two worlds, and a similar partitioning needs to be applied within the core to separate the data used and stored within the components of the level one (L1) memory system.

The basic component of the level one memory system is the memory management unit (MMU) which translates the virtual addresses that are seen by the software to actual

physical addresses that are bound to the hardware. This address translation is managed by a software-based solution that dictates the exact mapping that should take place, the cacheability of the addresses and access permissions. Within a TrustZone processor the hardware provides two virtual MMUs, one for each virtual processor. This enables each world to have a local set of translation tables, giving them independent control over their virtual address to physical address mappings and total isolation between the two memory spaces.

The ARMv6 and ARMv7 L1 translation table descriptor MMU software includes an NS field which is used by the secure-world to determine the value of the NS-bit to use when accessing the physical memory locations associated with the secure-world MMY. The non-secure world hardware completely ignores this field, and the memory access is always made with the NS-bit enabled. This design enables the Secure virtual processor to access either Secure or Non-secure memory and denies the non-secure virtual processor any possibility of accessing secure world memory.

Furthermore, ARM processors tag entries in a table named “Translation Lookaside Buffer” (TLB), which caches the results of address translations, with the identity of the context that made this translation. This allows entries from both contexts (secure and non-secure) to coexist within the TLB and enhance the performance of memory translations by reducing the stress of the MMU.

Another important entity of the level one memory system is the cache. It is a desirable feature of any high-performance design to support data of both security states in the caches. This removes the need for a cache flush when switching between worlds and enables high performance software to communicate over the world boundary. To enable this the level one memory system, and where applicable level two and beyond, processor caches have been extended with an additional tag bit which records the security state of the transaction that accessed the memory. This tag will mandate who can access the corresponding entry and what he can do to it. Any non-locked down cache line can be

evicted to make space for new data, regardless of its security state. It is possible for a secure line load to evict a non-secure line, and for a non-secure line load to evict a secure line.

3.2.2 Secure Interrupts

ARM processors provide an interrupt model that is composed of two kinds of interrupts, FIQ and IRQ. FIQ interrupts provide a method of performing a fast interrupt in a digital data processor having the capability of handling more than one interrupt is provided. When a fast interrupt request (FIQ) is received a flag is set and the program counter and condition code registers are stored on a stack. At the end of the interrupt servicing routine the return from interrupt instructions retrieves the condition code register which contains the status of the digital data processor and checks to see whether the flag has been set or not. If the flag is set it indicates that a fast interrupt was serviced and therefore only the program counter is unstacked. On the other hand, regular interrupt requests (IRQ) are handled in the priority they come and are always put on hold when a FIQ arrives.

The ability to trap IRQ and FIQ directly to the monitor, without intervention of code in either world, allows for the creation of a flexible interrupt model for secure interrupt sources. This way the monitor can route these interrupts to the correct world. Together with a security-aware interrupt controller, it allows for a design that can provide secure interrupt sources which cannot be manipulated by the normal world software. The recommendation from ARM is to use IRQ as non-secure world interrupt source and FIQ for the secure world. IRQs are the most common interrupts in common operating systems, this way, applying FIQs in the secure world will allow for TrustZone integration with minimal changes. If the processor is running the correct virtual core when an interrupt occurs there is no switch to the monitor and the interrupt is handled locally in the current world. If the core is in the other world when an interrupt occurs the hardware traps to the monitor, the monitor software causes a context switch and jumps to the restored world, at which point the interrupt is taken.

To provide the aforementioned interrupt functionality there should exist three interrupts table in the system. One for each entity, the normal world, the secure world and the monitor.

3.3 MULTIPROCESSOR SUPPORT FOR TRUSTZONE

The ARM architecture allows for configurations with up to four processors in a cluster. These processors can be set-up in either in Symmetric Multi-Processing (SMP) mode, or in Asymmetric Multi-Processing (AMP) mode.

When a processor is executing in SMP mode the cluster's Snoop Control Unit (SCU) will transparently keep data which is shared across the SMP processors coherent in the L1 data cache. When a processor is executing in AMP mode the executing software must manually maintain memory coherency if it is needed.

These multiprocessor systems may implement the ARM Security Extensions, giving each processor in the cluster a separate TrustZone implementation to work with. The ARM processor which currently implements both the multiprocessor features and the security features is the Cortex-A9 MPCore processor. With each of the processors within the multiprocessor cluster having a separate TrustZone implementation, they will have a normal world and a secure world per processor. This gives a four-processor cluster a total of eight virtual processors and twenty-four translation tables, each with independent control over their MMU configuration.

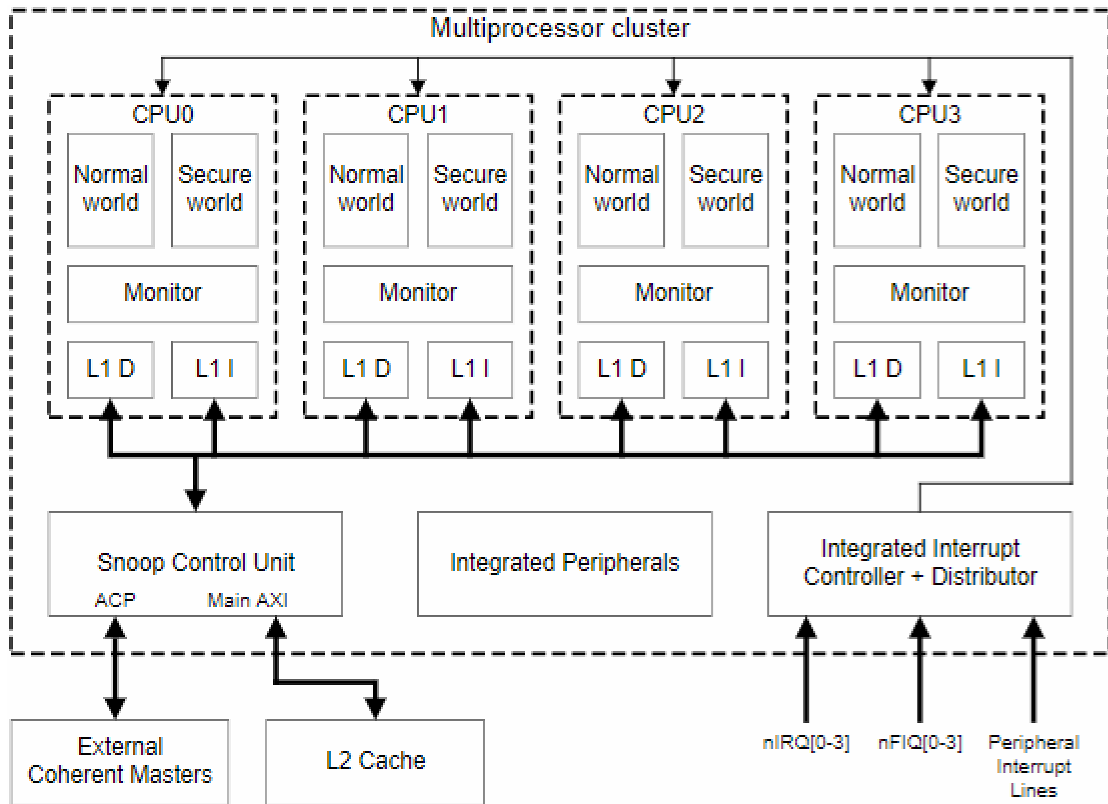

Figure 3-2: TrustZone Multiprocessor Support

3.4 ARM TrustZone Software Architecture

Until now we have seen all the hardware components that are provided by the TrustZone technology. But hardware alone, cannot provide enough functionality for the deployment of a TEE as we have defined it in the previous chapters. That is why secure software TrustZone aware software should be installed in a system that wishes to use the TrustZone technology to create the secure and non-secure world isolation of a TEE. The ARM architecture specifies the open component of security extensions which can be used to create a custom secure world software environment to meet their requirements. This section presents some of the possibilities that a software architect might want to consider when designing a secure world software stack.

The overall structure of the software architecture will be heavily influenced by the nature of the available Secure world processing resource. A system may provide a TrustZone-

enabled core, such as the ARM1176JZ(F)-S processor or may provide a dedicated processor for the Secure world, such as a Cortex-R4 processor.

The design that consist of the two separate physical processors is the simplest as each processor has a self-contained operating system with a minimal overall impact on the system design. Although, the most common and cost-efficient solution is incorporating the TrustZone technology within a single SoC together with the normal world dye.

3.4.1 Software Architecture

There are many possible choices when designing the software architecture of the secure and the non-secure world. The most complicated one is having a full-fledged OS just for the secure world, on the other hand the simplest solution is having just the libraries that provide the required functionality installed in the secure portion of the system, with many possible choices between these options.

Secure Operating System: Having a dedicated OS just for the secure world is the most secure option as it allows for having a flexible design within the secure world. This enables the developers to run concurrently multiple secure applications, dynamically installing new secure applications while having complete independence from the untrusted world. The most extreme option for this design resembles a design of separate CPUs for the both worlds in an Asymmetric Multi-Processing (AMP) configuration. The next option is to have the virtual processors in a Symmetric Multi-Processing configuration that will allow for a closer relationship between the two worlds. For example, the secure world might inherit the priority of the normal world that will enable better response times for media applications. An example of a separate secure world OS can be seen in Figure X.

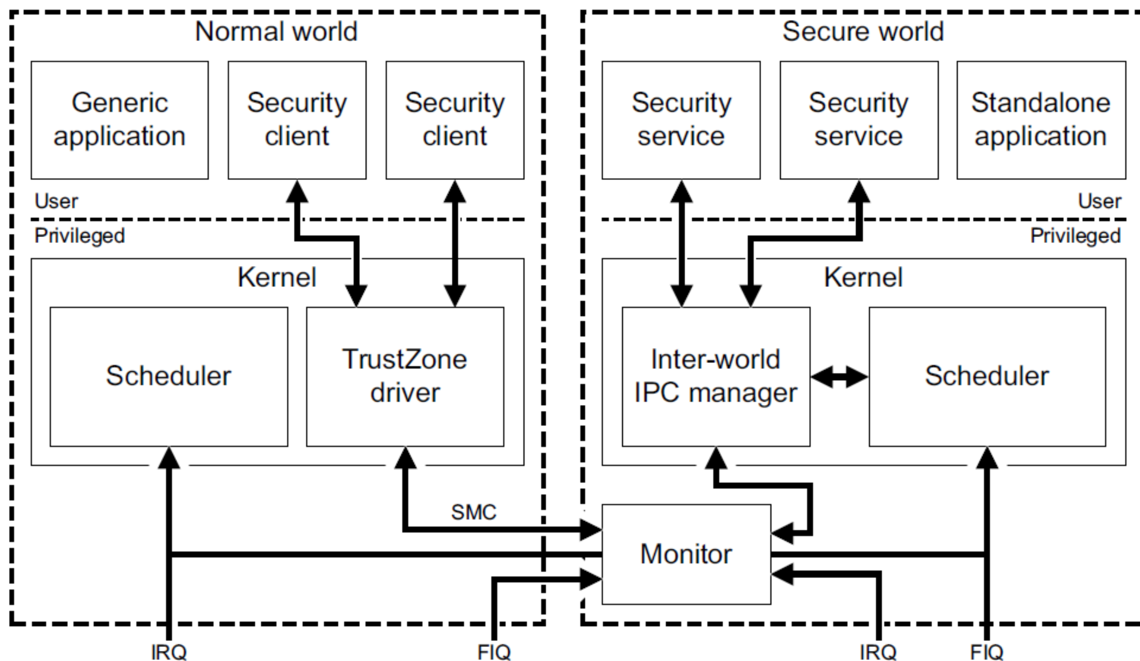

Figure 3-3: Software Architecture of the Normal World OS and the Secure World OS

One of the advantages of having this design is that the MMU of each processor can be utilized to sandbox each secure application in the secure world by providing separate virtual address spaces for each one and effectively isolating them from each other. This way independent secure applications can be run concurrently in the secure world without the need of trusting their peer applications. The kernel design can enforce the logical isolation of secure tasks from each other, preventing one secure task from tampering with the memory space of another.

Synchronous Library: Many use cases do not need a separate operating system just for the secure world, especially in low powered devices that do not have the capacity of running two operating systems at the same time. The option of having just a library that utilizes the TrustZone technology installed are enough to handle one job at a time with a scheduling architecture that roots in the normal world. In this case, the secure world is the slave and the normal world the master, in a scheme that binds the secure world to the normal and deeming it incapable of running independently from it.

Intermediate Options: There are many possible options in-between these two extreme designs. For example, there might be a design where the secure world OS does not implement an interrupt system of its own and utilizes the system found within the normal world OS as a virtual interrupt. This design might be vulnerable to a denial of service attack if the normal world OS was compromised and was not able to provide this virtual interrupt service. Alternatively, the MMU could be used to statically separate different components of an otherwise synchronous Secure world library.

3.5 Booting Process of a Secure System

A core component and a common attack target for a secure system is its boot process. For example, while the device is powered off, an attacker might try to replace the secure world image with a tampered one that opens certain attack paths. If the system boots without first checking the authenticity of the binary blobs that run within it, then the system is vulnerable and easily exploitable.

3.5.1 Boot Sequence of a TrustZone Enabled System

The first concept that needs to be understood for the boot process of a TrustZone enabled system is what exactly happens during boot up. Any processor that has the TrustZone extension installed boots up directly into the trusted world, this allows for checking the authenticity of the normal world.

The very first boot process is the ROM SoC bootloader which is responsible for initializing crucial peripherals such as memory controllers before switching to the second level boot of the device which is contained within non-volatile memory such as a flash memory. Afterwards, the boot process will fully initialize the secure world before booting and initializing the normal world. By then, the system is considered to be in a running state. Figure X depicts this process.

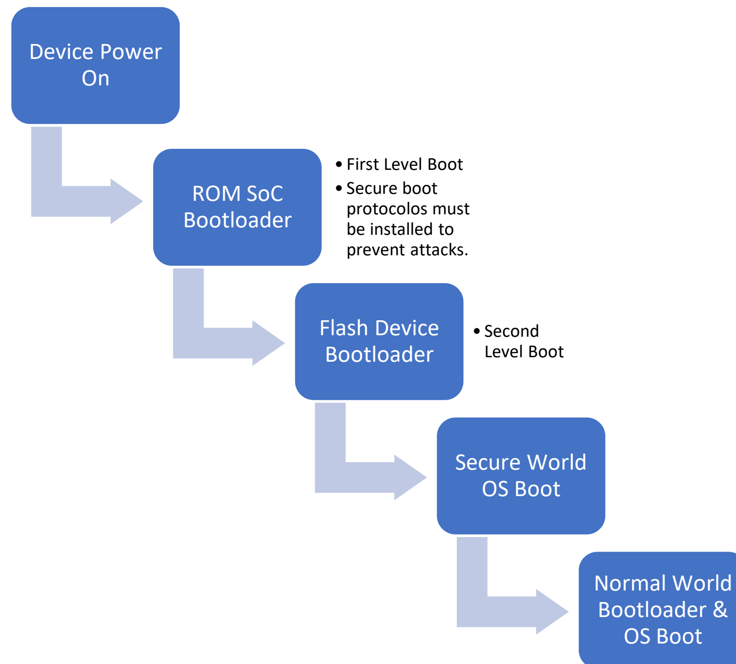

Figure 3-4: Boot Sequence of a TrustZone Enabled System

Systems that want an additional level of protection can use a signal input into the processor core to lock-down some of the critical Secure world configuration options in CP15. Asserting the **CP15SDISABLE** processor input signal will cause some of the Secure world CP15 settings to become unmodifiable, even if the modification is attempted by Secure world privileged software.

3.5.2 Secure Boot

The secure boot process adds cryptographic verification steps on each stage of the boot process. The target of this process is to check the integrity of each binary that is loaded in the system before it has an opportunity to run. The cryptographic signature protocol used is based on public-key cryptography such as RSA-PSS where a trusted vendor uses their private key to generate a valid signature on the code he wants to publish and each device comes with the corresponding public key preinstalled with the ability to verify each received software for its integrity and authenticity. The public-key does not need to be

kept confidential, but it does need to be stored within the device in a manner which means it cannot be replaced by a public-key that belongs to an attacker.

The secure boot process utilizes a concept called chain of trust. Starting with an inherently trusted component (such as the hardware) every consequent component can be verified before it is allowed to execute. The root of trust can change on each stage, for example, initially the bootloader can be verified using the OEM public key as described above, but then the bootloader might contain a separate public key that will be used to verify the next stage boot component. Since the bootloader image is verified and trusted then the new public key contained within it is also trusted to be used for verification.

Storing the first stage public key is a puzzling problem. If the key is stored as-is in the device hardware, then the system suffers from a class-break attack that will deem all the devices vulnerable if the private key was to be stolen or reverse engineered. One-time programmable hardware on the SoC such us poly-silicon fuses, can be used to store unique values in each SoC during device manufacture. This will allow for a different number of public keys to be stored on different devices that will partially mitigate this class break attack.

The simplest defense against hardware attacks is to keep sensitive code within on-SoC memory locations. If the code is never exposed outside the SoC then it is hard for an attacker to probe components within an integrated SoC design to mount hardware attacks due to the complexity of installing the needed physical probes within the SoC. The secure boot is responsible for loading code on the SoC memory, that is why care must be taken to properly verify any piece of code that is loaded and runs in the SoC so as to prevent any attack windows. Assuming the running code and required cryptographic hashes are already in safe on-SoC memory, the binary or public-key being verified should be copied to a secure location before being authenticated using cryptographic methods. A design that authenticates an image, and then copies it into the safe memory location

risks attack. The attacker can modify the image in the short window between the check completing and the copy taking place.

3.6 Monitor Mode

As described above, the monitor mode is the management software stack that handles the context switching between the normal and the secure world while it also handles the communication of the two worlds. The monitor mode acts as a gatekeeper from the less privileged normal world to the high privileged secure world. Normal world entry to monitor mode is tightly controlled. It is only possible via the following exceptions: an interrupt, an external abort, or an explicit call via an SMC instruction. The secure world entry to the monitor mode is a little more flexible, and can be achieved by directly writing to CPSR, in addition to the exception mechanisms available to the normal world. The monitor mode is a security critical component as it provides the only interface between the two worlds.

4 VULNERABILITIES OF TRUSTZONE BASED TEES

4.1 ATTACKS

4.1.1 TrustZone Code Execution [11]

This is the first of a set of three attacks against a TrustZone based TEE implementation in MSM8974 SoCs found in commodity mobile devices. The first attack achieves code execution in the context of a trusted application in the TEE. In this attack the author first of all exploited the MediaServer process to gain arbitrary code execution within its context. The MediaServer process is one of the entities that is able to communicate with the underlying TEE and that is why it is a high-value initial target.

With MediaServer privileges, the author then proceeded to probe the QSEECom driver. This driver is responsible for managing the TEE device while it also runs in normal-world kernel context. It was found that this driver contained a bug that allowed code escalation to kernel privileges of the normal-world. This way the author escalated to a higher-privileged position that allowed him to directly speak to the TEE device without going through the QSEECom driver.

With this position, the author probed the secure world with direct SMCs that allowed him to discover a set of vulnerabilities in the corresponding MediaServer trusted application that ultimately let him run arbitrary code within the context of this application. This way the author managed to escalate from normal user privileges to trusted application privileges that let him access any assets that this application had access to.

4.1.2 TrustZone Kernel Exploitation [12]

Continuing on the previous attack, the author was not able to access any information in the secure world due to the fact that each trusted application is isolated from each other and from the secure world kernel. This time though he went through a different path,

instead of gaining normal-world kernel privileges he directly found a way of exploiting the trusted application by using the QSEECom driver directly.

Once again, the author had trusted application code execution privileges and targeted the kernel of the secure world. By bypassing a set of lockdown security features and hijacking the system call architecture of the secure world, he was able to write shellcode binaries which were run in the context of the TrustZone kernel. This way he was finally able to run arbitrary secure-world kernel-privileged code, the highest level of privileges in the system as it can access any resource from any other entity in the system.

4.1.3 Extraction of Master keys from TrustZone [13]

With the highest privileges any resource could be accessed except for hardware bound keys that are concealed through the hardware. Although this master key is unattainable through the software, all the other keys that are derived from this key are stored in memory accessible by the software. The Android full-disk encryption (FDE) is one of those keys that the author of the blog targeted to demonstrate the power of the privileges he obtained. The FDE key is responsible for encrypting and decrypting the android memory so that only legit users can access it. It is a crucial part of the Android security system as with it anybody can read anything from the device memory.

More specifically, the author reverse engineered the keymaster trusted application which is responsible for managing all the cryptographic keys used in the Android ecosystem and discovered that the FDE key is not directly protected by any hardware-bound keys but by a software key which resided in the global buffer of the keymaster trusted application. By using a chain of exploits available to the author through the trusted world kernel privileges he was able to gain access to the FDE key, thus nullifying the disk encryption system of the Android ecosystem.

4.1.4 BOOMERANG Vulnerability [7]

BOOMERANG is a class of vulnerabilities that arises due to this semantic separation between the TEE and the untrusted environment. These vulnerabilities permit untrusted

user-level applications to read and write any memory location in the untrusted environment, including security-sensitive kernel memory, by leveraging the TEE's privileged position to perform the operations on its behalf. BOOMERANG can be used to steal sensitive data from other applications, bypass security checks, or even gain full control of the untrusted OS.

This exploitation is possible due to the fact that the two worlds have not well-defined means of communication in an ecosystem with ubiquitous implementations of trusted execution environments in many consumer devices. The problematic behavior roots in the different memory access control systems that are installed in the two worlds that are not well equipped to inherit the restrictions set by them. The target of this attack is to send restricted normal world addresses to the high-privileged secure world and convince it to write or read from these addresses since it has access to them. So, the attack targets the normal world and begins from the normal world, but it utilizes the secure world as a middle man to execute the actual attack.

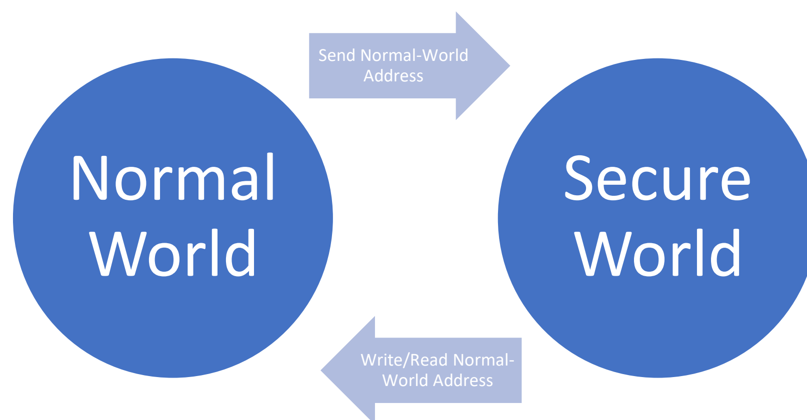

Figure 4-1: The Boomerang Attack

More specifically, the pointer sanitization that controls what addresses are sent to be used by the secure world, only checks in specific regions of the share memory between the two worlds. The sanitization process checks if the addresses checked are within an allowed range that the user has access to, if these addresses are out of this range then

the execution is halted. The boomerang attack requires that the attacker hides these addresses in a part of the share memory where the memory sanitizer does not check in order to bypass this security measure. When the secure world receives this address, it has no way of checking the province of this address, it assumes that it has been checked and it will act blindly upon it. With this behavior in hand, several trusted applications were identified that could be used as primitives for writing arbitrary binary values in any specified memory address of the normal world.

4.1.5 Downgrade Attack [14]

The downgrade attack is a form of attack that can be performed on the current implementations of the widely deployed ARM TrustZone technology. The attack exploits the fact that the trustlet (TA) or TrustZone OS loading verification procedure may use the same verification key and may lack proper rollback prevention across versions. If an exploit works on an out- of-date version, but the vulnerability is patched on the latest version, an attacker can still use the same exploit to compromise the latest system by downgrading the software to an older and exploitable version. Experiments were made on popular devices on the market including those from Google, Samsung and Huawei, and found that all of them have the risk of being attacked.

Research has shown that most TEE implementations, include systems for version controlling the binaries installed within the system for rollback attack prevention, it was found that all the vendors checked do not use this feature. So almost all devices in the market are vulnerable to the downgrade attack. This attack is very easy to mount as the string that specifies the trusted application binary location path can be easily manipulated and without many privileges an older version of the trusted application could be loaded from the SD card of the system.

4.1.6 Cache Timing Attacks

There is a set of attacks that exploit the side channel of timing the cache usage of the secure world.

Cache Timing Attack on AES in Virtualization Environments [15]**:** This attack does not target any specific TrustZone implementation but aims to show that the isolation characteristic of system virtualization can be bypassed by the use of a cache timing attack. Using Bernstein's correlation in this attack, an adversary is able to extract sensitive keying material from an isolated trusted execution domain. The authors of the research have demonstrated the attack on an embedded ARM-based platform running an L4 microkernel as virtualization layer by extracting an AES key that was used in a virtualized environment. This attack is mounted against an isolated virtualized environment running on the same processor as the "normal world" environment, proving that cross-world side-channel cache-timing attacks are possible, something that can be applied in TrustZone TEEs.

Prime+Count Attack [16]**:** The researchers have demonstrated a side channel that does not use the classic fine-grained methods (prime+probe, flush+reload etc.) but instead use the prime+count method that is a coarser-grained approach that significantly reduces the noise introduced by the pseudo-random replacement policy and world switching. This is not an attack per se but more of a demonstration for the methods that could be used to "smuggle" data through unmonitored side channels of the system.

ARMageddon Attack [17]**:** The ARMageddon attack provides a novel method for performing cross-core cache timing attacks against ARM based CPUs that were not possible before. More specifically, the researchers have shown how to solve key challenges to perform the most powerful cross-core cache attacks Prime+Probe, Flush+Reload, Evict+Reload, and Flush+Flush on non-rooted ARM-based devices without any privileges. According to the research, this attack outperforms most cache-timing attacks at the time of the writing while it provides proof of concept demonstrators that

sniff gestures of the user in secure inputs and steal cryptographic keys from a Java AES implementation. Although the attacks are not made against a TrustZone TEE, the research shows evidence that it can be escalated to affect even the secure world of the system.

TruSpy Attack [18]**:** This attack exploits the cache contention between normal world and secure world to recover secret information from secure world. Two attacks are proposed in TruSpy, namely, the normal world OS attack and the normal world Android app attack. In the OS-based attack, the attacker is able to access virtual-to-physical address translation and high precision timers. In the Android app-based attack, these tools are unavailable to the attacker, so the researchers devise a novel method that uses the expected channel statistics to allocate memory for cache probing. They also show how an attacker might use the less accurate performance event interface as a timer.

4.1.7 CLKscrew [19]

The need for power- and energy-efficient computing has resulted in aggressive cooperative hardware-software energy management mechanisms on modern commodity devices. Most systems today, for example, allow software to control the frequency and voltage of the underlying hardware at a very fine granularity to extend battery life. Despite their benefits, these software-exposed energy management mechanisms pose grave security implications that have not been studied before.

The CLKscrew attack exploits the software power management system to set the operational frequency and voltage in a combinational setting that will induce faults during sensitive operations of the system. Due to the software nature of this method, this is one of the few hardware attacks that can be mounted remotely.

More specifically, the attacker investigates possible combinations of operational frequency and voltage to find the exact settings needed so that the CPU begins to introduce faults in its computations. The attack then can begin by first clearing the cache of the victim core to reduce the random noise that could be introduced by it (step 1). In the next step, the attacker profiles the execution of the targeted code to find the exact

time that the code to be faulted is executed (step 2). Based on this profiling, the attacking process sets a timing anchor (step 3) after which a specific number of no-op operations are performed until the targeted piece of code is executed. Then the attack takes place by setting the fault-inducing settings on the processor for the time that the targeted operation executes and afterwards it is set to the normal operational settings. (steps 5,6).

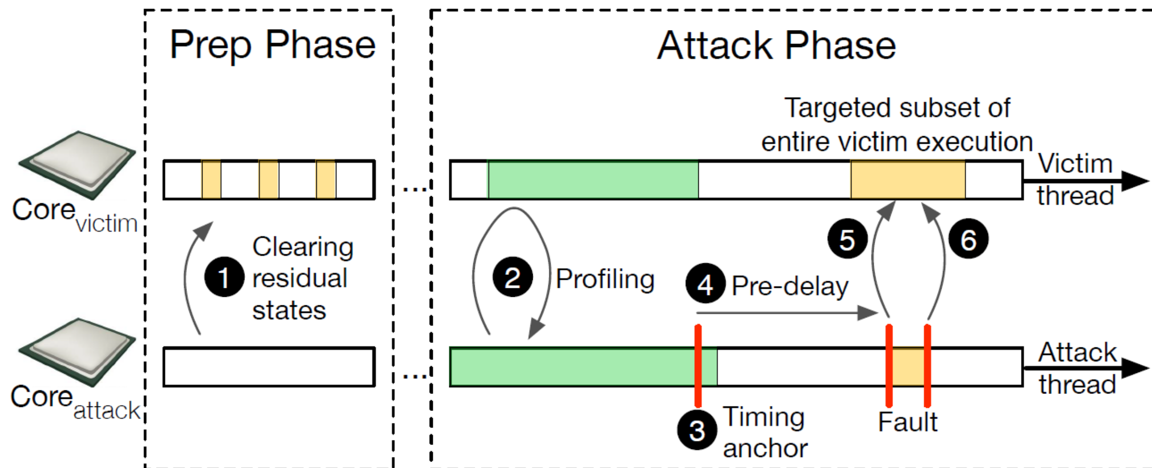

Figure 4-2: The CLKscrew Attack [19]

4.1.8 BADFET [20]

The researchers that demonstrated the BADFET attack proposed a novel solution that reinvented the electromagnetic fault injection attacks. The main contribution is that instead of targeting the processor during its operation, they targeted the peripherals that store sensitive data. So, the CPU does not directly introduce the faults during its operation but intermediate data stored outside of the CPU are faulted so that the CPU operates with faulty data.

To perform this second-order electromagnetic attack the researchers utilized a computer-controlled targeting system which was equipped with a high-power precision laser that will induce the faults in the peripherals. In the research an attack was demonstrated against the TrustZone equipped Cisco 8861 IP phone. The researchers first used their attack setup to interrupt the boot process which in turn forced the device to give a u-boot

console. It is through the u-boot console that the attackers managed to find a way to access the TEE of the device and gain a terminal with the privileges of the trusted world.

4.2 ATTACK COMPARISON

The attacks that have been presented this far achieve TrustZone based TEE exploitation by either gaining execution within the secure world context or uncovering secrets hidden in the secure world. Depending on how and what the attack achieves, each attack is analyzed in the following table (Table 4-2).

The characteristics of each attack include the target of each attack which can be either the secure world, the normal world or both of them. A strong attack targets the secure world or both of them, without diminishing the potential of just normal world attacks which can provide the attacker with normal world kernel privileges. Furthermore, there is also the characteristic of whether the attack requires normal world root privileges, something that defines the applicability of each attack due to the fact that this privileges might not be available on all devices.

Depending on what the attack can achieve, there are also the characteristics of whether an attack uncovers secrets from the secure world, gains code execution in the secure world and gains kernel code execution in the secure world. Once again, the more the attack achieves the stronger it is as it has a larger field of effect in the system.

I have marked with X the attacks that achieve the characteristic while with a question mark (?) the attacks that have the potential of achieving the corresponding characteristic.

	Targets Secure World	Targets Normal World	Requires Root Privileges	Uncovers TrustZone Secret	Gains TrustZone Execution	Gains TrustZone Kernel Execution
TrustZone Code Execution	X		X		X	

TrustZone Kernel Exploitation	X				X	X
Extraction of Master keys from TrustZone	X			X	X	X
BOOMERANG Vulnerability		X	X			
Downgrade Attack	X			?	?	?
Cache Timing Attack on AES in Virtualization Environments	X		X	X		
Prime+Count Attack	X	X	?	X		
ARMageddon Attack	X	X	?	X		
TruSpy Attack	X	?	?	X		
CLKscrew	X		X	?	?	?
BADFET	X			X	X	

Table 4-2: List of Characteristics for Each Attack.

5 DISCUSSION

The main problems identified in the TrustZone implementations can be categorized in the following categories:

- Closed Source Design
- Coding Bugs
- Shared Architecture
- Unused Features
- Unstandardized Communications

The closed source of the code that implements the TEE of commodity devices cripples the ability of the community to identify and fix possible vulnerabilities. Choosing to open source the core components of each TEE implementation will benefit the vendors by letting other experts check their code and improve their security. After all the security of the system must be consolidated by openly reviewed protocols and not the obscurity of closed source proprietary solutions.

Most of the implementations that were presented by the researches in this thesis suffered from serious coding bugs. In many occasions, these bugs lead to vulnerabilities that ultimately let the researchers to completely exploit the TEE installed on the device. This situation could be improved if proper coding techniques were utilized such as static code analysis and dynamic probing of the running software. Vendors should make a greater effort to develop secure and bug-free code because as the results show, their implementations have problems that could have been easily avoided.

The shared architecture of the TEE sacrifices the complete hardware isolation in order to reduce the manufacturing cost. More specifically, depending on the implementation, the secure and the non-secure world both share CPU cores, cache memory and non-volatile memory. Although the TrustZone technology provide techniques to properly isolate one world from each other, it is possible for attackers to find side channels that will ultimately

allow them to gain sensitive information from the secure world. This problem can be either solved by adding separate hardware for the secure world, but this will increase the manufacturing cost or properly check their software code in order to avoid side channels, but this path might not lead to a guaranteed solution to the problem.

Furthermore, the developers of most TEE implementations do not use the basic feature of version controlling the trusted application binaries in the secure world. This has led to the major vulnerability of the downgrade attack that allowed an adversary to sideload an older and possibly vulnerable trusted application binary and exploit this binary to gain access to the secure world. Missing this feature is a major issue because the functionality was already provided and not using it is wasteful.

Finally, the lack of standardization for the communication between the two worlds has led to vulnerabilities that were unforeseen. More specifically since the secure world has very high privileges, it can act upon any memory that belongs to the secure world even on addresses that contain kernel data and code. That is why in the boomerang vulnerability the researchers were able to use this miscommunication to push the secure world into exploiting the normal world kernel.

Closing, the trusted execution environment is marketed as a security solution that can solve many trust problems that rooted in the fact that security sensitive operations were run in the same environment as the normal operating system. Although sometimes it is claimed that it is one of the highest security solutions, the TEE can have many security problems that can totally break the security guarantees it should have provided. That is why care must be taken to fix these issues or the TEE technology could be deprecated and replaced by other dedicated solutions such us TPMs.

6 REFERENCES

[1] J. Seibel, K. LaFlamme, F. Koschara, R. Schumak and J. Debate, "Trusted execution environment". US Patent US20170214530A1, 27 01 2016.

[2] "ADVANCED TRUSTED ENVIRONMENT: OMTP TR1," OMTP Limited, 2009.

[3] V. Galindo, "Poulpita," 18 2 2014. [Online]. Available: https://poulpita.com/2014/02/18/trusted-execution-environment-do-you-have-yours/. [Accessed 22 5 2018].

[4] G. Arfaoui, S. Gharout and J. Traore, "Trusted Execution Environments: A Look under the Hood," in *2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, Oxford, 2014.

[5] "TEE System Architecture v1.2," GlobalPlatform, 2018.

[6] "TEE Client API Specification," GlobalPlatform, 2010.

[7] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel and G. Vigna, "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, 2017.

[8] ARM, "ARM Security Technology Building a Secure System using TrustZone® Technology," 2004. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0301h/Chdebaee.html.

[9] ARM, "ARM Security Technology Building a Secure System using TrustZone® Technology," 2005. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[10] Arm, *Layered Security for Your Next SoC.*

[11] G. Beniamini, *Full TrustZone exploit for MSM8974,* 2015.

[12] G. Beniamini, *QSEE privilege escalation vulnerability and exploit (CVE-2015-6639),* 2016.

[13] G. Beniamini, *TrustZone Kernel Privilege Escalation (CVE-2016-2431),* 2016.

[14] Y. Chen, Y. Zhang, Z. Wang and T. Wei, "Downgrade attack on trustzone," *arXiv preprint arXiv:1707.05082,* 2017.

[15] M. Weiss, B. Heinz and F. Stumpf, "A cache timing attack on AES in virtualization environments," in *International Conference on Financial Cryptography and Data Security*, 2012.

[16] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupe and G.-J. Ahn, "Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone," in *Proceedings of the 34th Annual Computer Security Applications Conference*, New York, ACM, 2018, pp. 441-452.

[17] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016.

[18] N. Zhang, K. Sun, D. Shands, W. Lou and Y. T. Hou, "TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices.," *IACR Cryptology ePrint Archive,* vol. 2016, p. 980, 2016.

[19] A. Tang, S. Sethumadhavan and S. Stolfo, "{CLKSCREW}: Exposing the Perils of Security-Oblivious Energy Management," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017.

[20] A. Cui and R. Housley, "{BADFET}: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection," in *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.