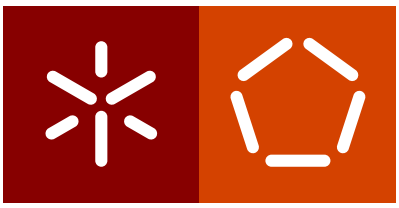
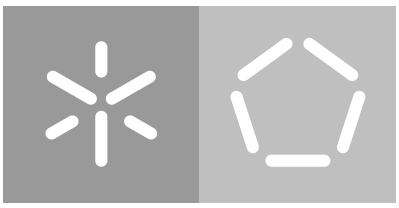


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Miguel Miranda Quaresma

TrustZone based Attestation in Secure Runtime Verification for Embedded Systems

July 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Miguel Miranda Quaresma

TrustZone based Attestation in Secure Runtime Verification for Embedded Systems

Master dissertation
Integrated Master Degree in Informatics Engineering

Dissertation supervised by
José Carlos Bacelar Almeida

July 2020

ACKNOWLEDGEMENTS

I would like to thank my supervisor José Carlos Bacelar for his continued support and guidance which made this experience an extremely gratifying one that I will fondly remember. Additionally I'd like to thank my friends and colleagues Eduardo Correia, Etienne Costa, João Vieira and José Carlos Martins for their support and guidance over the last five years. I would also like to thank my father, mother and sister for all the sacrifices they made in favor of my work and their support throught this process. Lastly I would like to extend these words of gratitude to my closest family and friends whose support was crucial.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

ARM TrustZone é um "Ambiente de Execução Confiável" disponibilizado em processadores da ARM, que equipam grande parte dos sistemas embebidos. Este mecanismo permite assegurar que componentes críticos de uma aplicação executem num ambiente que garante a confidencialidade dos dados e integridade do código, mesmo que componentes maliciosos estejam instalados no mesmo dispositivo. Neste projecto pretende-se tirar partido do TrustZone no contexto de uma *framework* segura de monitorização em tempo real de sistemas embebidos. Especificamente, pretende-se recorrer a components como o ARM Trusted Firmware, responsável pelo processo de *secure boot* em sistemas ARM, para desenvolver um mecanismo de atestação que providencie garantias de computação segura a entidades remotas.

Palavras chave: Atestação, Sistemas Embedidos, TEE, TrustZone

ABSTRACT

ARM TrustZone is a security extension present on ARM processors that enables the development of hardware based Trusted Execution Environments (TEEs). This mechanism allows the critical components of an application to execute in an environment that guarantees data confidentiality and code integrity, even when a malicious agent is installed on the device. This projects aims to harness TrustZone in the context of a secure runtime verification framework for embedded devices. Specifically, it aims to harness existing components, namely ARM Trusted Firmware, responsible for the secure boot process of ARM devices, to implement an attestation mechanism that provides proof of secure computation to remote parties.

Keywords: Attestation, Embedded systems, TEE, TrustZone

CONTENTS

1 INTRODUCTION 1
1.1 Motivation 2
1.2 Thesis Structure 3
2 STATE OF THE ART 4
2.1 Threat model for embedded systems 4
2.2 Trusted Execution Environments 5
2.2.1 TEE Hardware Architecture 6
2.2.2 TEE Software Architecture 7
2.3 ARM TrustZone 9
2.3.1 TrustZone Overview 9
2.3.2 Switching worlds 10
2.4 Secure Boot 11
2.4.1 Trusted Board Boot - Secure Boot in ARM-TZ 11
2.5 OP-TEE 15
2.5.1 Architecture 15
2.5.2 Exceptions 16
2.5.3 TEE driver 16
2.5.4 Trusted Application Types in OP-TEE 18
2.5.5 Trusted Storage in OP-TEE 20
2.6 TEE Alternatives 21
2.7 Summary 21
3 THE PROBLEM AND ITS CHALLENGES 23
3.1 Attested Computation 24
3.1.1 Specification 25
3.1.2 System Architecture 27
3.2 Third Party Trusted Applications 28
3.2.1 Specification 28
3.2.2 Practical Considerations 29
4 DEVELOPMENT 30
4.1 Decisions 30
4.2 Implementation 31

4.2.1 Initialization 32
4.2.2 Attestation 38
4.2.3 Tools 41
4.2.4 Third Party TA Support 41
4.2.5 Attested Key Exchange 52
4.3 Summary 64
5 CASE STUDIES / EXPERIMENTS 66
5.1 Experiment setup 66
5.1.1 Board Setup 67
5.1.2 QEMU Setup 71
5.2 Use Cases 72
5.2.1 Trusted Peripherals 73
5.2.2 IoT Aggregation 75
5.2.3 Qualified Digital Signatures 75
5.2.4 Secure DDS 79
5.3 Summary 84
6 CONCLUSION 85
6.1 Future Work 86

LIST OF FIGURES

Figure 1	Secure Storage Overview [46]	7
Figure 2	TrustZone security model [24]	10
Figure 3	Secure boot overview	12
Figure 4	Chain of Trust	13
Figure 5	BL-2 Verification Flow	14
Figure 6	Normal World OP-TEE Architecture	17
Figure 7	Bootstrap TA binary structure	19
Figure 8	Encrypted TA header	20
Figure 9	High Level Architecture of the System	27
Figure 10	Chain of Trust for TAs	28
Figure 11	TA load flow	43
Figure 12	Third party TA binary format	45
Figure 13	Pine A64+	67
Figure 14	ARMv8 Boot flow	68
Figure 15	Board setup	71
Figure 16	QEMU setup	72
Figure 17	Proposed VSC Architecture	77
Figure 18	LibDDSSec Architecture [2]	81
Figure 19	LibDDSSec 3-Way Handshake	82

ACRONYMS

AES Advanced Encrytption Standard 9, 21, 36, 52, 60, 62, 63, 83
APB Advanced Peripheral Bus 9
API Application Programming Interface 6–8, 11, 15, 18, 26, 59, 64
ASN.1 Abstract Syntax Notation One 29, 40
ATF-A ARM Trusted Firmware-A 1, 11, 26, 31–33, 66, 69, 70
AXI Advanced eXtensible Interface 9

BSP Board Support Package 67

CA
1) **CA** Client Application 6–8, 15–18, 27, 30, 31, 53, 54, 77, 81, 85, 86
2) **CA** Certificate Authority 57, 82, 83
CAN Controller Area Network 9, 73, 74
CBC Cipher Block Chaining 63
CCM Counter with CBC-MAC 52, 63
CoT Chain of Trust 1, 12, 13, 28, 68, 69
CSR Certificate Signing Request 77–79, 82
CTR Counter 36

DDS Data Distribution Service 79–81
DER Distinguished Encoding Rules 41
DH Diffie-Hellman 53, 82, 83
DOS Denial of Service 7, 26
DRAM Dynamic Random Access Memory 69
DRM Digital Rights Management 5
DT Device Tree 70

E2EE End-to-End Encryption 5
ECC Elliptic Curve Cryptography 20, 40
ECDH Elliptic-curve Diffie–Hellman 53, 55–57
ECDSA Elliptic Curve Digital Signature Algorithm 13, 39, 40, 45
eIDAS Eletronic Identification, Authentication and Trust Services 75, 76, 79
EL Exception Level 10, 15, 17, 18, 27, 30, 31, 33, 34, 53, 68, 69

ELF Executable and Linkable Format 18, 19, 25, 41, 50
eMMC embedded MultiMedia Card 21

FEK File Encryption Key 21
FINeID Finnish Eletronic Identity 76
FIQ Fast Interrupt Request 11, 16

GCM Gallois/Counter Mode 21, 52, 62, 63, 83
GIC Generic Interrupt Controler 16
GMAC Gallois Message Authentication Code 63
GP Global Platform 6, 7, 21, 76

HUK Hardware Unique Key 37, 41

IMA Integrity Measurement Architecture 3
IoT Internet-of-Things 2, 4, 75, 86
IR Infrared 73
IRQ Interrupt Request 11, 16
ISA Instruction Set Architecture 10, 66, 67
IV Initialization Vector 19, 62, 63

MAC Message Authentication Code 63, 80, 82, 83
MiTM Man-in-the-Middle 53
MMU Memory Management Unit 33, 35, 73

NIST National Institute of Standards and Technology 63
NS Non-Secure 9, 11, 33, 73
NV Non-Volatile 12, 13

OEM Original Equipment Manufacturer 5
OS Operating System 2, 3, 5, 8–11, 15, 16, 18, 20, 26, 29–31, 67–69, 81
OTP One-Time Programmable 15

PTA Pseudo Trusted Application 18, 38, 53

QoS Quality of Service 79

RAM Random Access Memory 36, 37
RBG Random Bit Generator 63, 64
REE Rich Execution Environment 5–9, 15, 16, 18, 21, 43, 46, 48
REEFS Rich Execution Environment File System 21

RNG Random Number Generator 64
ROM Read Only Memory 12
ROTPK Root of Trust Public Key 12, 14, 32
RPC Remote Procedure Call 15–17, 20, 43, 46, 47, 49
RPMB Replay Protected Memory Block 19, 21, 46
RSA Rivest-Shamir-Adleman 13, 18, 20, 28, 30, 39, 41, 42, 44–46, 49, 51
RV Runtime Verification 2, 22, 23

SCR Secure Configuration Register 11
SE Secure Element 21
SEV Secure Encrypted Virtualization 2
SGX Software Guard eXtensions 2, 23, 24, 41, 65, 85
SHDR Signed Header 44, 49
SMC Secure Monitor Call 11, 16, 31, 69
SOC Secure Outsourced Computation 24–26, 30, 52
SoC System on Chip 10, 11, 23, 31, 32, 66–68, 85
SPD Secure-EL1 Payload Dispatcher 31–33, 69, 70
SPL Secondary Program Loader 69, 70
SPSR Saved Program Status Register 10
SRAM Static Random Access Memory 68, 69
STS Station-to-Station 52

TA Trusted Application 5–8, 10, 15–21, 23–31, 35–38, 41–57, 60–62, 64, 65, 72, 74, 76–79, 81, 83, 85, 86
TBB Trusted Board Boot 11, 13, 23, 24, 68
TCB Trusted Computing Base 1
TCP Transmission Control Protocol 17
TEE Trusted Execution Environment 1–3, 5–9, 11, 15, 16, 18, 20–25, 29, 30, 41, 53, 64, 75–77, 83, 85, 86
TLS Transport Layer Security 53
TPM Trusted Platform Module 21, 23
TSK Trusted Application Storage Key 21
TZASC TrustZone Address Space Controller 9
TZPC TrustZone Protection Controller 9, 73

UART Universal Asynchronous Receiver/Transmitter 67
UC Universal Composability 86
UDP User Datagram Protocol 17

UUID Universally Unique Identifier 8, 17, 19, 38, 48, 60, 62, 74

VM Virtual Machine 15

VSC Virtual Smart Card 76

1

INTRODUCTION

Embedded devices are used across a wide range of sectors, from simple home appliances to automated driving systems, handling large amounts of data that must be kept safe both at rest, in use or in transit. The safety-critical nature of some of these systems introduces a requirement for security driven development, where the reliability of the system is a combination of several mechanisms at different layers (hardware, firmware, software).

Trusted Execution Environments (TEEs) are one such mechanism, combining firmware and hardware to provide a secure area in the processor where the confidentiality and integrity of (critical) code and data is ensured. In order to achieve this, the code running in this secure area, called **Secure World**, is separated from the one running in the non-secure area, called **Normal World**, via different isolation mechanisms. ARM TrustZone, a security extension present in ARM processors, is one such "mechanism" that provides support for the implementation of hardware backed TEEs where the Secure World is isolated from the Normal one using both hardware and firmware mechanisms, which in combination with a trusted firmware layer, such as ARM Trusted Firmware-A, is able to provide a system that implements security from the CPU to the system level. This mechanism should be paired with a root of trust which justifies the trust placed on the system, as such it's common for these systems to resort to a Chain of Trust during the boot process that prevents untrusted code to be executed at startup. This process becomes part of the Trusted Computing Base that can be used in conjunction with other features to attest, via a cryptographic signature, the integrity of the system to a remote party. This capability is fundamental for detecting possible breaches in the security of a device that performs a critical function or has privileged access to a larger infrastructure.

In use cases that require a high degree of trust to be put on the system, additional mechanisms need to be employed in order to justify that trust, namely ones that assert the correctness of the implementations both at the firmware and software level. Asserting the correctness of software is usually achieved through one of two approaches:

- **Static analysis** which ensures the correctness of a program before it's execution by analysing the source code (or object file in some cases).
- **Dynamic analysis** which works by examining the execution of a single system.

Both of these approaches have their advantages and disadvantages. Static analysis techniques such as formal verification, which provide mathematical guarantees of a program's correctness, are limited in the amount of properties that can be expressed (**e.g.** memory leaks). Furthermore, techniques such as model checking suffer from exponential complexity growth as the system being checked grows, commonly referred to as the state explosion problem. Dynamic analysis on the other hand is limited in its scope, with techniques such as fuzzing requiring a large amount of data to cover an acceptable amount of code. Runtime Verification (RV) is a dynamic analysis method that works by checking whether the target system verifies a given set of properties. In Runtime Verification the properties the system is expected to exhibit generate monitors which are responsible for verifying its status during the execution of the system. This allows the monitors to provide the system with feedback when certain properties are violated so that corrective measures can be taken.

1.1 MOTIVATION

The adoption of trusted hardware in recent years has been motivated by the need for stronger security guarantees, in an era where the ever increasing complexity of software systems results in an increased attack surface. The implementation of security mechanisms that contemplate the whole stack, from hardware to software, is therefore crucial. Some of these mechanisms might involve running common services in isolated environments that ensure the confidentiality and integrity of both code and data.

Hardware-based solutions such as Intel SGX, ARM TrustZone or AMD SEV aim to provide this isolated environments, referred to as Trusted Execution Environments, through a combination of hardware and software mechanisms, with SGX providing remote attestation capabilities. These capabilities can be used to design and implement a secure outsourced computation protocol, as shown in [4], which, amongst other things, can be used by cloud service providers to give users integrity and confidentiality guarantees. ARM TrustZone doesn't natively support remote attestation, as it was designed for running applications locally, inside the Trusted Execution Environment, however, with the emergence of IoT [13, 18] devices acting as trusted sensors or the integration of a Runtime Verification framework as a monitoring infrastructure present novel use cases for attested computation and remote provisioning of applications. A remote attestation mechanism would allow remote parties to confirm the source of sensor information as a trusted one or to outsource computational workloads, with high security requirements, to ARM-based servers and verify its execution inside the TEE. On the other hand, the use of attested computation to aid in the task of a Runtime Verification framework acting as a monitoring infrastructure protects the results produced by the monitors against active adversaries that could corrupt them and, as a result, prevent the detection of violations. The combination of these mechanisms with a secure boot process is paramount in preventing code from being injected at boot time, which if successfull could lead to rootkit vulnerabilities in the Normal World while also rendering the use of TEEs meaningless as there would be no way of attesting the integrity of the "Trusted" OS. While there have been several attempts at controlling the code that can run in computer systems,

either by resorting to a hash table of the memory pages from the kernel memory region, such as IMA, or via sandboxes [17], all these mechanisms only work after the system has booted, leaving it vulnerable to malicious firmware. Thus, resorting to techniques such as a trusted firmware layer that is responsible for secure boot functionality and acts as a root of trust, represents an additional step in hardening the system. Thus, the present work proposes the implementation of an attested computation scheme that relies on trusted hardware features such as ARM TrustZone and secure boot, to reinforce the trust placed on the results produced by the code that runs inside this devices.

1.2 THESIS STRUCTURE

ARM TrustZone is the fundamental technology underlying this project, thus in the following section we will present an in-depth description of all the parts involved in harnessing this technology to produce a Trusted Execution Environment such as OP-TEE. Namely, we will examine topics such as the hardware mechanisms used by TrustZone to enforce isolation, the architecture used by OP-TEE, a specific implementation of a TEE, to provide secure services to applications executing in the Normal World, in this case running Linux. Additionally we will describe the secure boot mechanism that functions as the root of trust for ARM TrustZone and present some alternatives to TrustZone itself, briefly discussing the advantages and disadvantages of each one.

The third chapter will present an overview of the problem being tackled by the system we propose along with a formalization of this system that describes the functionality expected out its components. This formalization entails the design of a scheme, based on the one proposed in [4], which specifies a secure outsourced computation scheme as the composition of three different schemes. Furthermore we also describe some of the pratical considerations that should be taken into account when developing systems such as this, that deal with sensitive information.

The fourth chapter deals with the development process of the system. We discuss some of the challenges encountered and provide information regarding some of the most important design choices that were made, referring factors such as performance and security. This discussion includes the examination of the inner workings of the Secure OS and the firmware layer to provide the reader with an overall view of where the system fits.

The applicability of the whole system is discussed in the fifth chapter, where we walkthrough the installation and configuration process of the software and firmware stack in real hardware **i.e.** a development board. This is followed by the description of four different use cases for the system, describing the context and applicability of each use case by itself and how it can harness the system to improve its security mechanisms.

The last chapter discusses the key takeaways from the work that was developed and provides some guidance regarding future work pertaining to the TEE ecosystem.

2

STATE OF THE ART

Embedded systems, as opposed to general purpose computer systems, have been designed to perform a clearly defined task, usually as part of a more complex system. As a consequence, they're commonly used in safety-critical systems, such as flight control or automotive systems, that present a high security requirements which, if not met, can have serious consequences. This, in addition to their widespread use as Internet-of-Things (IoT) devices, has made them one of the primary targets for attackers. The nature of the systems they integrate, as well as their lack of resources, calls for both performance and security to be taken into account, leading to a complex design process that aims to balance these two aspects. In an effort to address these requirements, several technologies have emerged in an attempt to provide secure environments for information to be stored and programs to be executed without introducing overhead.

2.1 THREAT MODEL FOR EMBEDDED SYSTEMS

The process of securing systems, whether software or hardware ones, involves elaborating a threat model which describes the type of threats that the system is subject to as well as the assets being protected. The mechanisms used to secure the system will not only depend on the threats themselves, but also the following security principles, usually referred to as the CIA triangle:

- **Confidentiality**: information should only be available to its owners
- **Integrity**: information shouldn't be modifiable by external agents
- **Availability**: legitimate clients should have access to information at any given moment

Most embedded systems deal with sensitive data, such as cryptographic keys, whose confidentiality must be assured. Integrity is also a key factor in some of these systems, for instance, embedded systems used in cars, either for the auto-pilot or the entertainment system, must ensure that the code executed in the IoT nodes responsible for these tasks isn't modified as this could lead to arbitrary code execution (**CVE-2017-9983**) [31, 30]. Finally, the availability of systems such as the ones used in nuclear facilities to monitor the reactors, is paramount to ensure the correct functioning of these.

Another important factor to consider when securing a system is its attack surface, which is directly related to the complexity of the system. Hence, favoring simpler systems that perform clearly defined functions and limit outside interaction is a way of decreasing the attack surface, this being one of the main principles of TEEs, as explained in section 2.2.

The type of attacks that can target this "surface" can be classified in three different categories:

- **Software attacks**: attacks against the software level that don't require physical access
- **Simple hardware attacks**: passive attacks with physical/hardware access to the device
- **Laboratory hardware attacks**: sophisticated hardware attacks that resort to techniques such as power analysis and laboratory equipment

ARM TrustZone, described in detail in section 2.3, aims to protect against the first two types of attacks, and so that is the threat model considered in this work.

2.2 TRUSTED EXECUTION ENVIRONMENTS

Before understanding what Trusted Execution Environments are, it's useful to understand the need for such environments and what concrete problems they address. Computing devices provide users with an environment in which to run a large set of applications, expanding their use and making them flexible to different use cases. This environment, usually referred to as the Rich Execution Environment (REE) which is comprised of a Rich OS such as Android, although flexible also leads to a larger attack surface and gives up control over what code (applications) can be executed on the system, as well as the data it handles, both in terms of access control and integrity assurance. While most applications don't require security measures beyond the ones provided by the OS, the ones dealing with sensitive data such as banking applications, E2EE messaging applications, biometric data or even DRM frameworks benefit from the use of a separate component for storing cryptographic keys and other sensitive data, making the REE unsuitable for applications that require a high degree of security.

TEEs, as enviroments providing isolated execution, were proposed to address this problems by running as secure areas of the main processor that are separated (via hardware and/or software) from the main operating system **i.e** the REE. This approach aims to provide guarantees in terms of integrity and confidentiality of both code and data via mechanisms such as physical memory isolation, secure storage, etc. Applications running in these environments, called Trusted Applications (TAs), are also isolated from eachother as well as from the REE, to ensure that even if a malicious agent installs itself in the REE, it won't compromise the applications running in the TEE.

The trust in TEEs is based on the assumption that all the code executed in these environments has been signed by a trusted entity (**e.g.** OEM) and verified before being executing. Thus, the need for a secure boot

process is implicit in order to ensure that every instruction executed from the instant the system boots, including the ones used in verifying the trusted applications, is verified. As will be made evident in 2.4, the importance of this mechanism lies in the fact that, if compromised, it removes all security guarantees provided by TEEs and can lead to arbitrary code execution that is persistent across reboots [47, 36, 51].

In an effort to standardize TEE implementations, Global Platform (GP) has devised a standard [16] to which new implementations should adhere to in order to ensure portability and ease of use. By adhering to this standard, developers can easily port Trusted Applications between different TEEs.

2.2.1 *TEE Hardware Architecture*

To ensure the isolation of systems running in the TEE, GP expects resources to be managed in a way that guarantees the security of Trusted Applications. Thus, when a resource is controlled by the TEE, the REE should not be able to interact with that resource outside of the API entry points exposed by the TEE. However, the contrary is not always true, as due to the nature of the REE **i.e.** no isolation guarantees, it can be possible for the TEE to interact with resources controlled by the REE.

Shared Memory

Although the TEE should guarantee the isolation of main memory, there should be a way for TAs and CAs to share data when interacting with eachother. According to GP, this feature can be implemented by a component of the TEE called Communication Agent, which is responsible for providing an efficient way for both parties to share large amounts of data, either by copying or directly sharing the data. However, due to the fact that the REE has unrestricted access to shared data, TAs shouldn't blindly trust the data they read from this memory. On the other hand, and since shared memory can also be used for TAs to share data amongst themselves, data stored in these memory can be trusted by the TA if the trustworthy indicator can assure the TA that the data hasn't been exposed outside the TEE.

Trusted Storage

Digital data can be at any of three diferent states at any given instant: at rest, in transit or in use, all of which require mechanisms to be set in place to ensure that it's kept safe. In TEEs, data in transit or in use is isolated in secure memory (RAM or registers) by the mechanisms put in place by the TEE. However, data at rest, such as cryptographic keys or other confidential data, must also be protected. Trusted Storage is a mechanism that aims to address those issues by guaranteeing confidentiality and integrity of general purpose data and key material as well as atomicity of the operations performed on that data. An overview of a proposed architecture can be seen below:

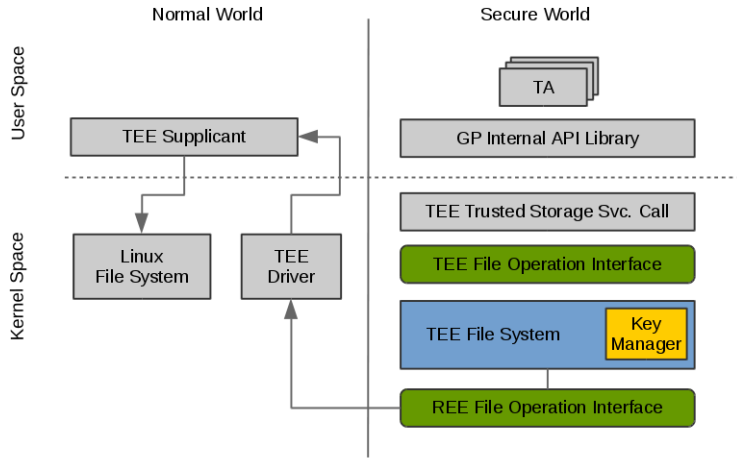


Figure 1: Secure Storage Overview [46]

According to Global Platform, trusted storage can be backed by non-secure resources if adequate cryptographic protection is applied. Additionaly, it must be bound to a particular device and prevent unauthorized access as well as rollback prevention mechanisms. One interesting takeway from these requriments is that, due to the fact that trusted storage can be backed by non-secure resources, to which the REE has access to, it is vulnerable to Denial of Service (DOS) attacks through intentional data corruption by a malicious application running in the REE.

2.2.2 *TEE Software Architecture*

Building on the previous mechanisms, a TEE must combine hardware and software mechanisms to provide a secure runtime environment that meets the expected security requirements. One common way to limit the amount of interaction between systems, thus reducing the attack surface, is the use of APIs that define the set of features made available by one system. APIs exposed by TEEs should allow REE applications to use the services provided by the TEE and TAs to use core system services such as cryptographic and timing operations. In addition to these pre-defined services, the implementation of new TAs allows for new functionality to be added, albeit in a controlled manner.

TEE Client API

The **TEE Client API** [15] specifies how Client Applications (CAs) executing in the REE can establish sessions to interact with TAs via commands. In the context of the TEE Client API, sessions are sequences of logically linked commands (messages) issued by the CA, where each command corresponds is associated with an identifier for that represents the operation made available by the target TA and a set of parameters it expects,

that be used for either input or output. The messaging between the CA and the TA is handled by the REE Communication Agent and the TEE Communication Agent.

Trusted Applications

Trusted Applications are programs that run in the TEE, isolated from the REE and from eachother, which can be used to implement code that benefits from running a secure environment. For a TA to be installed on a device, it must first be signed by the developer using a valid/trusted key, which ensures that all the code executing in the TEE has been verified, further strengthening the trust in this environment. This step is paramount as it prevents malicious actors from being able to run code inside the TEE which would weaken the trust that could be put into it. Each TA is identified by a Universally Unique Identifier (UUID), which is used by CAs when they wish to invoke their services. After installation, when a CA establishes a session with a TA, it connects to an instance of the TA, which can be seen as process with it's own physical memory. Previously, we've seen that the memory of a device where a TEE is installed is also partitioned to ensure isolation, this remains true for TA instances running inside a TEE, whose address space is also isolated from other instances, even if they belong to the same TA. Trusted Applications can also act as Client Applications in the sense that they have the ability to invoke other TAs.

TEE Internal Core API

Apart from the TEE Client API, the TEE also exposes an internal API that allows TAs to access system level functionality. The intent with standardizing such an API is to allow for TAs to be portable across diferent TEEs without much, if any, code refactoring. The **TEE Internal Core API** provides TAs with a set of core services made available by the Trusted OS such as OS functionality, cryptographic operations, timing or even arithmetic operations. Each of this services are, in fact, exposed through a dedicated API, namely:

- Trusted Core Framework API
- Trusted Storage API for Data and Keys
- Cryptographic Operations API
- Time API
- TEE Arithmetical API
- Peripheral and Event API

It's the Trusted Core Framework API that allows TAs to act as clients of another TA. One key advantage of providing a constant interface for services such as cryptographic operations is the ability to swap out the

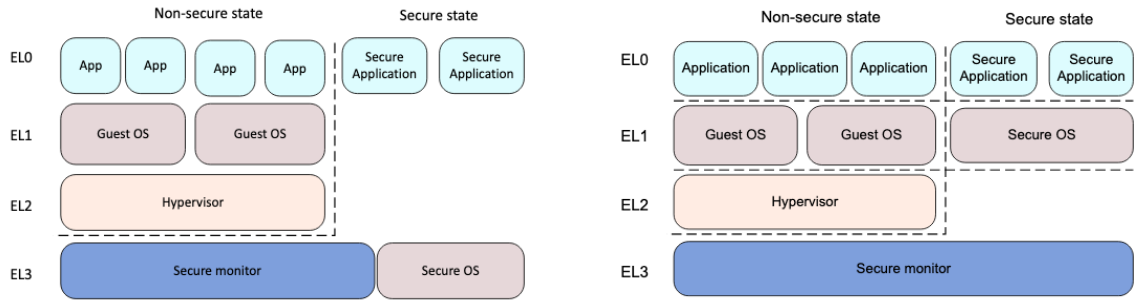
underlying implementation without "breaking" the interface. A use case where this is important is when cryptographic operations such as AES encryption can be executed by a single hardware instruction on devices that support it, instead of using software implementations, which greatly reduces the execution time as well as the size of the OS image.

2.3 ARM TRUSTZONE

ARM TrustZone is a security extension present in ARM processors that provides hardware and firmware support for the implementation of TEEs. TrustZone provides system-wide isolation by partitioning each physical core in two virtual cores, one executing in the Secure World (**i.e.** TEE) and the other in the Non-Secure World(**i.e** REE). In addition to the CPU, memory (**i.e** RAM and registers) is also divided by using an additional bit along with addresses of memory or peripherals. Memory accesses made through the Advanced eXtensible Interface (AXI) bus include this bit, called the Non-Secure (NS) bit, to indicate whether the access is made from the secure or non-secure world, which is used by the TrustZone Address Space Controller (TZASC) to limit the (physical) memory regions accessible from that world. Accesses to cached data also include the NS bit, which can trigger a cache miss when the normal world tries to access a cache line marked as secure. Peripherals, which are connected to the CPU via the Advanced Peripheral Bus (APB), which in turn connects to the AXI via an AXI-to-APB bridge, are configured as either secure or non-secure. This configuration is set in the bridge to ensure backwards compatibility and can be either statically set or dynamically changed at runtime. The state of a peripheral can be changed, by the secure world, via the TrustZone Protection Controller (TZPC), which alters the NS bit used as input to the bridge. This granularity is extremely useful when dealing with peripherals such as a CAN bus, as it can give exclusive access to ARM TrustZone, preventing adversaries from modifying the data on this buses.

2.3.1 *TrustZone Overview*

A diagram depicting TrustZone's security model can be seen below:



(a) AArch32 (ARMv7)

(b) AArch64

Figure 2: TrustZone security model [24]

The ARMv8 ISA introduces the concept of Exception Levels (ELs), EL0 to EL3 which, similarly to rings, determine the privilege level of the current execution state, with EL3 corresponding to the highest privilege level. The difference between the AArch32 and AArch64 security models lies in the exception level that the Secure OS runs at: EL3 in the former (compatible with ARMv7) vs. EL1 in the latter. This difference ensures full compatibility with ARMv7, as when EL3 is using AArch32, all the privileged modes in the Secure state are executed in EL3. Additionally, by moving components such as the Secure OS to a lower privilege level, the attack surface of higher privilege levels such as EL3 is effectively reduced [24].

Looking at the diagram it's possible to see that, in AArch64, only the Secure Monitor, alongside the SoC firmware, runs at the highest privilege level EL3, while the trusted world kernel runs at S-EL1 and the TAs at S-EL0. Switching between exception levels are only possible when the processor takes or returns from exceptions, either by increasing/maintaining the exception level in the former case or by decreasing/maintaining the exception level in the latter. In ARM, any condition or event that requires handling by privileged software, a Secure Monitor for instance, is considered an exception.

The exception vector table is used to map exceptions to their respective handler based on the current exception level. Therefore, when an exception is taken, execution is handed over to the corresponding handler, after saving the current execution state in the Saved Program Status Register (SPSR) of the current EL level (SPSR_ELn). As will be made evident in the next section, this is the underlying mechanism of switching between security worlds.

It's important to note that, as a direct consequence of this architecture/layout, the Secure Monitor is a fundamental part of TrustZone while, on the other hand, the secure world can take many forms, varying in terms of complexity and functionality. Some of these include a dedicated OS running (trusted) applications, such as OP-TEE, a language runtime for .NET applications [40] or even a single library.

2.3.2 *Switching worlds*

Switching security worlds is directly related to changing exception levels via two types of exceptions:

- Secure Monitor Calls (SMCs)
- FIQ and IRQ interrupts (which can be configured to be considered Secure interrupts)

When one such exception is triggered, the CPU hands execution over to the Secure Monitor running in EL3, which will save the current execution context (registers and return address) and invoke either the normal or the secure world. The world in which the processor is currently running, as indicated by the NS bit in the Secure Configuration Register (SCR), coupled with the type of the exception directly influences how it (the exception) is handled. Secure world exceptions triggered in the normal world, such as the invocation of a Secure Monitor Call (SMC) as part of a TEE Client API function, are handled by the Secure Monitor. On the other hand, if the exception is triggered in the secure world, it is handled by the secure world itself. Conversely, when a normal world exception is triggered in the secure world, the Secure Monitor transfers execution to the normal world. As expected, and similarly to what happens with the secure world exceptions, normal world exceptions are handled by the normal world itself.

2.4 SECURE BOOT

The security guarantees provided by the previously described mechanisms all work under the assumption that a root of trust, considered to be secure by default, has verified their integrity before executing them. In ARM TrustZone this trust anchor is the secure boot mechanism, which ensures the integrity of the instructions executed on the SoC from the instant the system boots. This means that only valid firmware and Trusted OS images will, after successful authentication, be loaded into the SoC. One key feature of this process is the ability for the Trusted OS to be loaded before the normal world kernel, which allows it to perform any necessary operations before any untrusted code is executed.

In ARM processors, the component responsible for implementing secure boot along with runtime services for the TEE being booted, is the (Trusted) firmware. Due to the lack of standardization on secure boot in ARMv7 processors, where the implementation was device dependent, ARMv8 introduced a reference implementation called Trusted Board Boot (TBB) [3], as part of ARM Trusted Firmware-A, also responsible for SMC calling convention and power management.

2.4.1 *Trusted Board Boot - Secure Boot in ARM-TZ*

ARM processors using Trusted Board Boot rely on digital signatures to verify each stage of the boot process. Each stage loads a cryptographically signed firmware image BL-X, with X being the stage number, and checks if its hash matches the one stored in the BL-X Content Certificate, which in turn is verfied using the BL-X Key Certificate, also using its hash. If all verifications succeed, the image is loaded, else the system fails to boot. The execution flow of this process is represented by the following diagram:

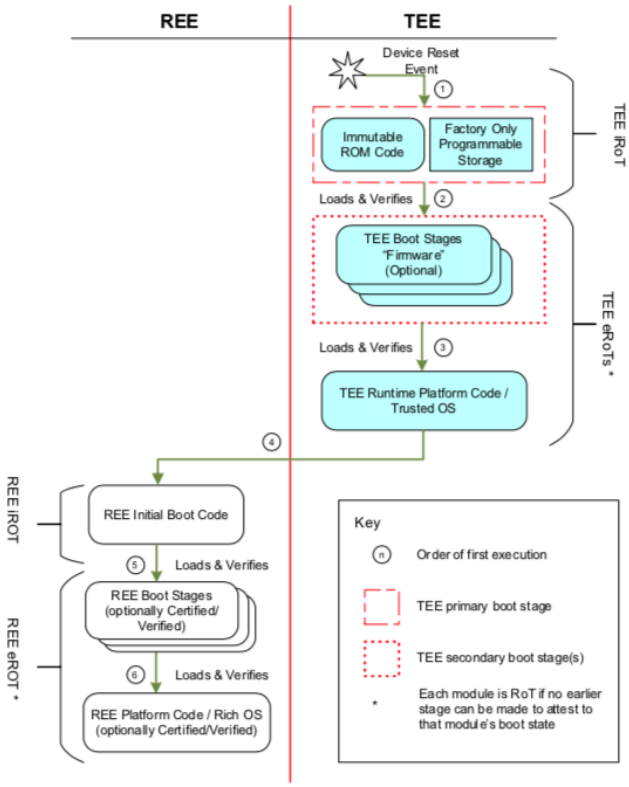


Figure 3: Secure boot overview

The different verification phases of the boot process form a Chain of Trust (CoT) where each link corresponds to a stage that relies on the previous ones to ensure the integrity of the boot. One caveat of this approach is the need for bootstrapping trust in the first stage of secure boot, in ARM this initial trust is established via the following set of implicitly trusted components that make up the **Root of Trust**:

- SHA-256 hash of the Root of Trust Public Key (ROTPK), stored in the trusted root-key storage registers
- **BL-1 firmware image**, called ROM boot code as it resides in the ROM and cannot be tampered with, signed by the manufacturer using the private part of the ROTPK

The image authentication process involves two types of X.509 v3 certificates, a Key Certificate used to authenticate the public keys whose private part was used to sign the Content Certificate that contains the SHA-256 hash of the firmware image being loaded at that stage. The authentication process loads an image and calculates its SHA-256 hash, which it then compares to the one stored in the Content Certificate, that has been previously authenticated. In addition to the image's hash, the Content Certificate also contains a Non-Volatile (NV) counter that is compared to the one stored in hardware. This mechanism prevents rollback attacks, where an adversary could force the system to load an outdated firmware image with known vulnerabilities. To prevent this, each time a patch is applied, the NV counter stored in hardware is

incremented and only images whose Content Certificate has a NV counter value greater or equal to this one are loaded. Due to the resource constrained nature of embedded systems, the size of the keys as well as the choice of cryptographic algorithms used for each cryptographic primitive ought to be considered carefully as to not sacrifice security for performance and vice-versa. In particular, the digital signature algorithm used is an important factor to take into account, as it can negatively impact performance or, in some cases, make implementation unfeasible, depending on the key size. According to the TBB specification, both ECDSA and RSA algorithms are supported, with ECDSA being more efficient due to the reduced key size, in comparison with RSA, for the same security level.

The trust relation between the different components that integrate secure boot forms the Chain of Trust, which is depicted in the following diagram:

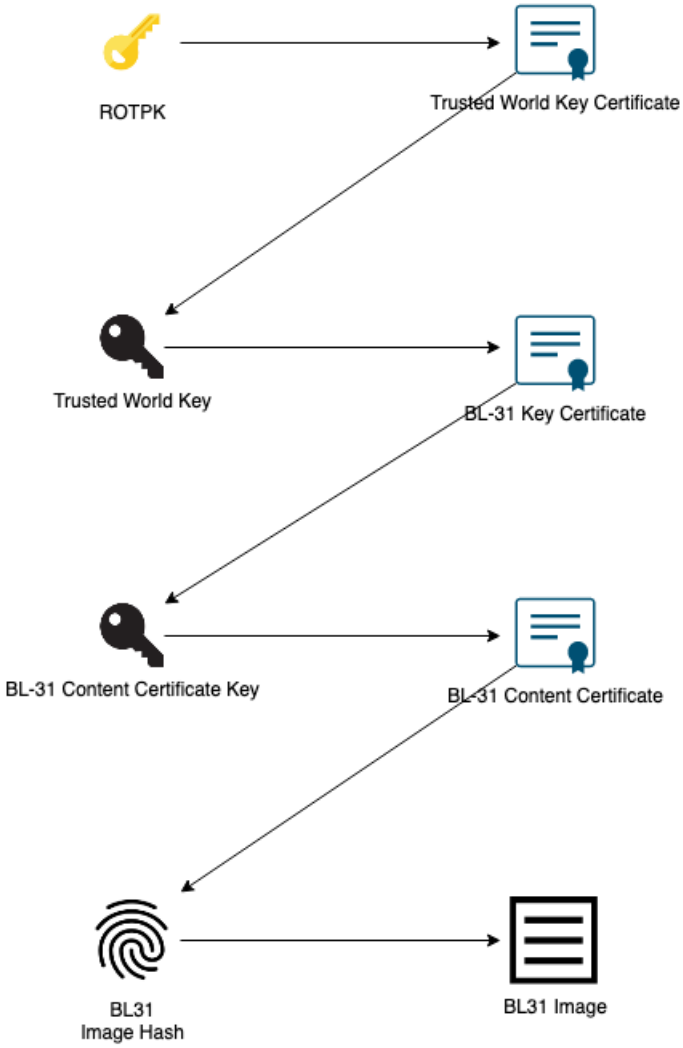


Figure 4: Chain of Trust

Each of the certificates, whether a Key or a Content certificate, is signed by a private key whose public part must also be certified. After the initial BL-2 image is loaded, whose content certificate is signed by the Root of Trust Key(private part of the ROTPK), the remaining images (BL-3X) require that both the Key Certificate and the Content Certificates be verified. The Key certificate, which contains the public part of the BL-3X key whose private is used to sign the Content certificate, is signed by the Trusted (or Non-Trusted) World Key. The Trusted World Key, along with the Non-Trusted World Key, are certified by the Trusted Key Certificate which is signed by the Root of Trust Key.

It's important to note that each firmware image is responsible for verifying the next image, by loading the Content and Key certificates and checking the digital signatures. For instance, before loading BL-2, BL-1 compares the hash it has stored of the ROTPK with the one in the BL-2 Content Certificate and, if they match, it then calculates the hash of the stored BL-2 image and compares the output against the one in the BL-2 Content Certificate, loading BL-2 in case they match:

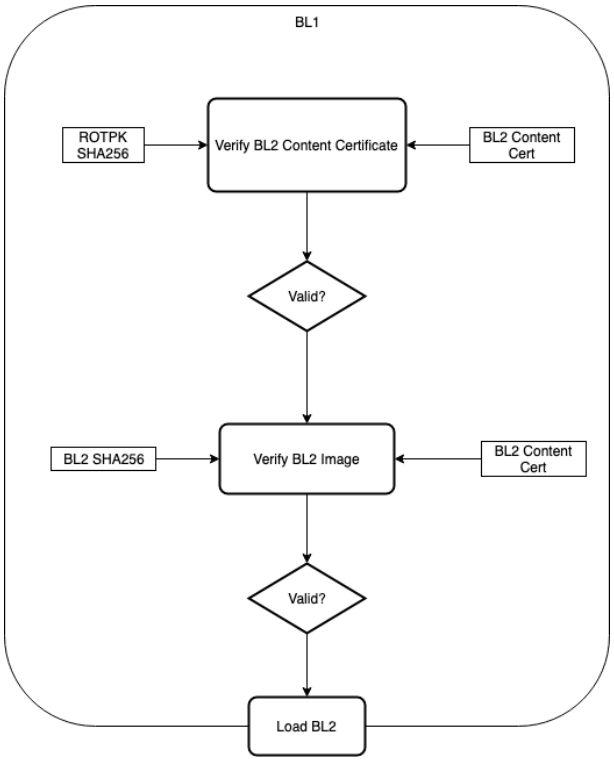


Figure 5: BL-2 Verification Flow

Even though the public key for the root of trust, used in the verification of firmware images, doesn't need to be confidential, its authenticity must be protected to prevent it from being replaced by keys that belong to attackers, which might allow untrusted firmware to be loaded during the boot process. The storage of this key should be carefully considered as resorting to hardware components such as SoC-ROM would lead all to devices to use the same public key, making them vulnerable to class-break attacks. On the other hand,

on-SoC One-Time Programmable (OTP) hardware allows unique values to be set during manufacturing, making them more suitable for storing such information [23].

2.5 OP-TEE

OP-TEE is a TEE implementation, compliant with the GlobalPlatform Standard [16], designed to run in parallel with a non-secure OS which takes advantage of ARM-TrustZone hardware isolation mechanisms. In addition to running in TrustZone enabled ARM processors, it's designed in such a way as to take advantage of other forms of isolation such as VMs or even separate processors (**e.g.** secure element). This design has three main goals in mind:

- **Isolation**: it should provide isolation from the non-secure OS and isolate the TAs from eachother using hardware mechanisms
- **Small footprint**: it's size should be such that it fits in the on-chip memory of ARM processors
- **Portability**: it should support several architectures, isolation mechanisms and setups

Since the code running the TEE is isolated from the main operating system, OP-TEE provides a common interface for applications in the REE (**i.e.** Client Applications) to interact with the services provided by the TAs. This interaction is achieved via Remote Procedure Calls (RPCs), where each call invokes a command made available by each TA. The commands supported are entirely defined by each Trusted Application, the only requirement being the implementation of a common interface defined by OP-TEE.

2.5.1 *Architecture*

Being compliant with the GlobalPlatform Standard [16] means OP-TEE's architecture is similar to the one described in section 2.2 however, it's still useful to examine some of the implementation dependent aspects. One of this aspects is the TEE itself, which can have varying degrees of complexity as well as the Normal World components used for Trusted Storage and interaction with Client Applications. In OP-TEE, these components are:

- a secure privileged layer running at EL-1 (Trusted OS)
- secure userspace libraries to be used by Trusted Applications (Trusted Framework)
- Linux kernel TEE framework and driver
- Linux userspace library designed upon the TEE Client API

- Linux userspace supplicant daemon (tee-supplicant) which provides services to the Trusted OS

Establishing the security boundaries on this architecture is fundamental to develop safe systems, enabling the separation between trusted and untrusted code. In this context, any code running in the normal world, independently of the developer, is considered untrusted, which means no sensitive information should be shared with this code, all inputs should be validated and no assumptions should be made about its behaviour (integrity). On the other hand, all code running in the secure world is trusted, which is made possible by the hardware features of TrustZone as well as the usage of cryptographic primitives such as digital signatures to ensure the authenticity of Trusted Applications and firmware images. As a direct consequence of this separation, components such as the TEE driver or the tee-supplicant daemon aren't trusted by OP-TEE, as these are vulnerable to attackers.

2.5.2 *Exceptions*

Exceptions are the main actor when switching between security worlds, with the exception type used to decide at which level the exception should be handled. Specifically, FIQ/IRQ interrupts can be configured via the ARM Generic Interrupt Controler (GIC) as secure interrupts to be handled in the secure world, leading to two distinct types of interrupts in OP-TEE: **native interrutps** handled by the OP-TEE OS and **foreign interrupts** routed to the normal world via the Secure Monitor.

The execution model of OP-TEE, whereby Client Applications issue Remote Procedure Calls to interact with TAs, means that thread scheduling doesn't need to be implemented by the OP-TEE OS as threads in the secure world, called trusted threads, are scheduled on-demand. This way, when a thread in the normal world invokes a Secure Monitor Call, the Linux kernel TEE driver maps it to a trusted thread in the secure world. Conversely, when the OP-TEE OS receives a foreign interrupt, the execution context of the trusted thread is saved by the Secure Monitor, and a normal world thread is scheduled to handle the interrupt. Due to the fact that the trusted thread only resumes execution when the normal world thread invokes an SMC again, trusted threads are, to a certain degree, scheduled by the Linux kernel running on the REE.

2.5.3 *TEE driver*

The TEE driver plays a fundamental role in managing the interaction between the normal and the secure world. On the normal world side, it handles the requests from Client Applications to the secure world, namely command invocations or shared memory allocation requests. On the secure world side it is responsible for handling the Remote Procedure Calls issued by OP-TEE, which are stored in a queue that is processed (consumed) by tee-supplicant threads:

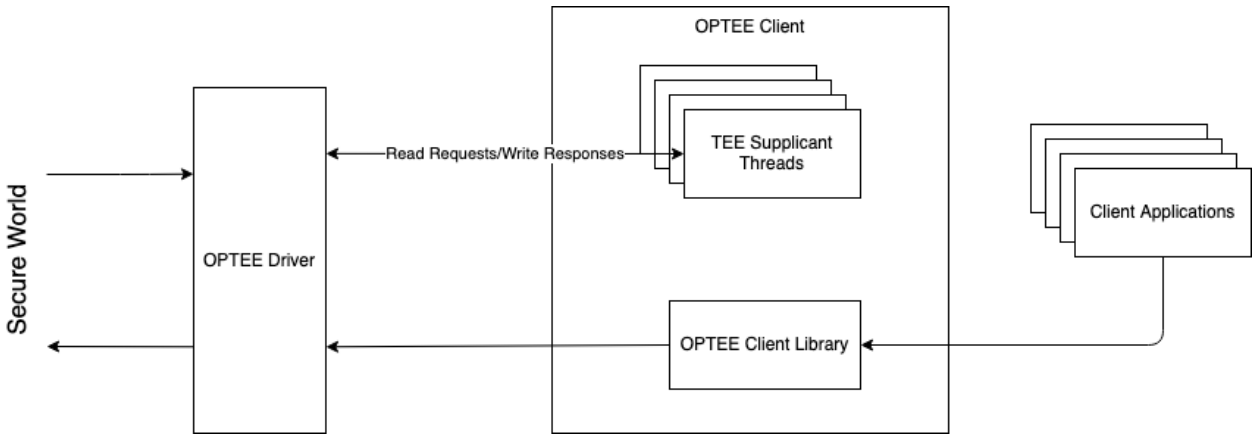


Figure 6: Normal World OP-TEE Architecture

The results of these calls are returned to OP-TEE via the driver as well. Each of these Remote Procedure Calls has a command identifier, which indicates the requested functionality, and a set of parameters that are used as input or output of the command. Some of the commands tee-supplicant exposes are:

- Secure storage access
- Shared memory allocation
- Network sockets (TCP and UDP)

To invoke a Remote Procedure Call, OP-TEE must save the context of the current thread and trigger an Exception Level change, that is handled by the EL3 runtime firmware **i.e.** the Secure Monitor, which invokes the OP-TEE driver running in the normal world. Upon receiving the RPC, the driver will place it on a queue, which is consumed by tee-supplicant threads, responsible for performing the requested functionality.

In addition to the tee-supplicant and the OP-TEE driver, the Normal World side of OP-TEE also includes a library that is used by Client Applications to interact with the Secure World **i.e.** initiate sessions and invoke commands.

An interesting detail that stems from the trust boundaries established by OP-TEE, whereby all the code running in the Normal World is untrusted, including the TEE library and driver responsible for translating the requests from Client Applications to the Secure World, is the inability to guarantee that these components (the driver and the library) don't modify the parameters of the requests made by the client. As such, the driver or library might modify the TA UUID or the values of the input parameters set by the Client Application. As we'll see in later sections, this problem can be solved using some of the mechanisms developed as part of this work.

2.5.4 *Trusted Application Types in OP-TEE*

In OP-TEE there are two major types of Trusted Applications which differ, amongst other things, on the Exception Level they run at:

- Pseudo Trusted Applications
- User Mode Trusted Applications
  - Early TAs
  - REE filesystem TAs
    - Bootstrap TAs
    - Encrypted TAs

Pseudo Trusted Applications allow developers to expand the set of features provided by the Trusted OS itself, as these are compiled at the same time as the OS, integrate the same binary and run at the same Exception Level. User mode TAs on the other hand, are similar to the normal world applications, running at a lower privilege level and being given access to the TEE Internal Core API as well as the ability to make use of other TAs, both Pseudo and User mode ones. In addition to these two main types, there are three (sub)types of Usermode TAs: Early TAs are TAs that are executed as part of the boot process and come pre-installed; Bootstrap TAs are executed on-demand, when an application, either a Client Application or a Trusted Application, invokes a command; Encrypted TAs, which are also executed on demand but whose binary is encrypted while stored.

Bootstrap and Encrypted TAs are both stored in the REE filesystem, and are loaded into secure memory when a Client Application or a TA issues a command. The tee-supplicant component is responsible for fetching the TA binary from the REE filesystem, which loads it into the shared memory region between the Normal World and OP-TEE, which copies the binary to secure memory before performing the necessary verifications. All Trusted Applications are compiled into ELF binaries and signed with an RSA key, whose public part is used by OP-TEE to verify the signature of the binary application upon loading it. This verification requires additional metadata which wraps the ELF binary, resulting in a new binary format with the extension **.ta** and the following structure:

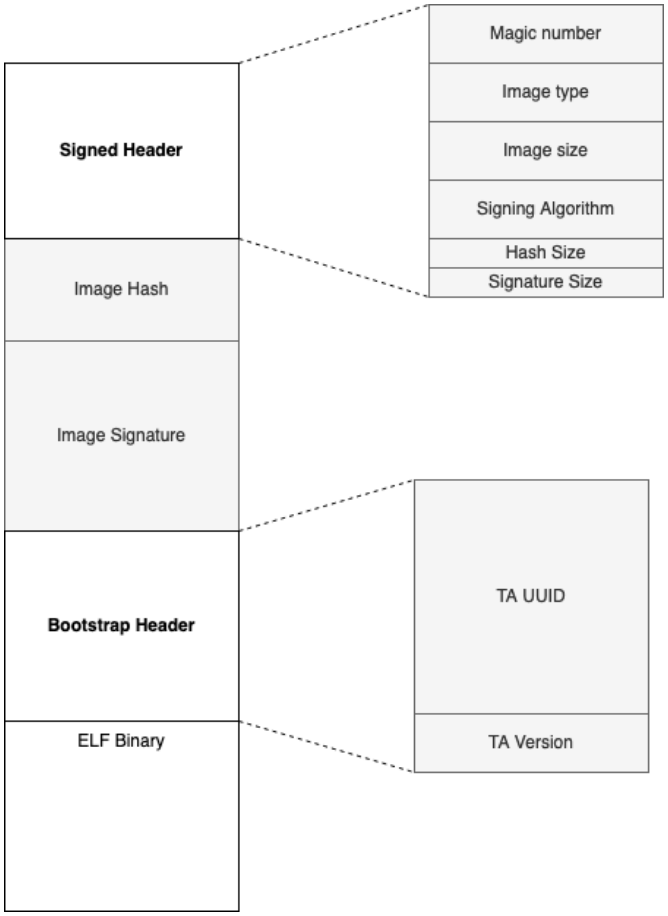


Figure 7: Bootstrap TA binary structure

The **Signed Header** is the first section of the binary, and it stores metadata used in the signature verification operations, namely the signature algorithm, hash and signature size as well as the image size. The image hash, which is computed over the Signed Header, the Bootstrap Header and the ELF binary, is stored immediatly after the signed header, and is followed by the image signature, performed over this hash. The **Bootstrap Header** holds information about the TA itself, namely the UUID used to identify it, as well as the TA version, used to prevent rollback attacks, when the RPMB filesystem is enabled. While both Bootstrap TAs and Encrypted TAs share this metadata, the latter have an additional header region, which contains the encryption algorithm used, the IV length and value, as well as the tag generated by the authenticated encryption algorithm:

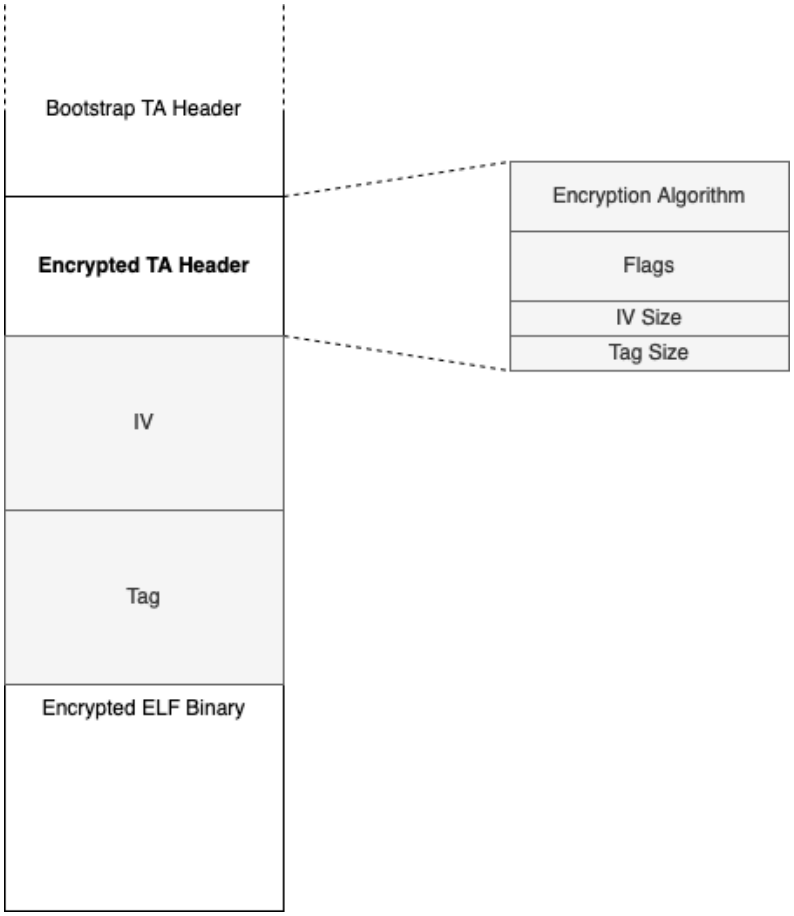


Figure 8: Encrypted TA header

In this case, the image hash and signature are also computed over these values, ensuring the authenticity of the full binary file.

One caveat of the TA signing process is that only RSA keys are supported, which ensures cross-compatiblity between different versions of OP-TEE, as some of these are compiled without support for Elliptic Curve Cryptography.

One final key consideration regarding Trusted Applications is that, due to the high privilege with which Pseudo TAs run, and considering the principle of least privilege, Usermode TAs should be the default type for most use cases, as they don't increase the attack of the underlying (Trusted) OS. Pseudo TAs should only be used in cases that are best handled inside the TEE core, for instance, issuing RPCs to the Normal World in order to load Trusted Applications.

2.5.5 *Trusted Storage in OP-TEE*

OP-TEE's implementation of Trusted Storage, called Secure Storage, is used to store general data and key material while assuring its confidentiality and integrity as well as the atomicity of the operations performed

on it. In compliance with Global Platform's standard, secure storage provides each TA with its own private storage, not accessible to other TAs nor the REE. As a way ensure that each TA has exclusive access to its own secure storage TEE files, called secure objects, are encrypted using a File Encryption Key (FEK) that is encrypted with the Trusted Application Storage Key (TSK), that only the TA knows.

Secure storage in OP-TEE can be of two different types which although GP compliant, differ in the degree of security they provide:

- Rich Execution Environment File System (REEFS)
- Replay Protected Memory Block (RPMB): a partition on an embedded MultiMedia Card (eMMC)

The main difference between the two is the rollback protection offered by the second filesystem using a counter as part of the secure object's metadata, which is encrypted using AES-GCM. Although OP-TEE supports both types to be used simultaneously, using their respective identifiers **TEE_STORAGE_PRIVATE_REE** and **TEE_STORAGE_PRIVATE_RPMB**, either storage type must be enabled at compile time. The **TEE_STORAGE_PRIVATE** identifier can be used to refer to either implementations but defaults to the first one when enabled.

2.6 TEE ALTERNATIVES

The problems addressed by TEEs such as ARM TrustZone, namely handling cryptographic data and operations in a secure environment, has been the focal point of several alternative approaches. One example are Secure Elements (SEs), which consist of a (separate) dedicated chip, on the same device, that can be used to store cryptographic data and perform cryptographic operations, commonly used in smart cards. Features such as tamper resistance and hardware separation make SEs a more secure alternative to TEEs, albeit a more complex one due to the additional hardware involved. Trusted Platform Modules (TPMs) are another alternative which enable trust in the computer hardware by providing, amongst other things, remote attestation capabilities to attest the software and hardware configuration at a given instant, secure storage of cryptographic keys used in full disk encryption [27] or early malware detection [28]. However, in comparison to a TEE such as ARM TrustZone, both of these solutions trade usability/felixibilty for security, effectively reducing the scope of usage. In addition, some of the functionalities provided by TPMs have successfully been implemented using ARM TrustZone [35], even showing improvements in the performance of cryptographic operations.

2.7 SUMMARY

Trusted Execution Environments are a combination of hardware and software isolation mechanisms aimed at creating a secure execution environment where code and data integrity as well as confidentiality are ensured.

This makes them ideal for running security critical applications while also providing services to normal world applications dealing with sensitive data. However, all the security properties of TEEs depend on a single root of trust, the secure boot process, to ensure that only trusted code is loaded into the secure world. This leads to a single point of failure which, if successfully exploited, removes any of the security benefits of TEEs. It's this single point of failure that makes a Runtime Verification framework for bootstrapping the security of the boot process a key element in ensuring the reliability of mission critical embedded systems. Furthermore, the combination of additional security mechanisms with TrustZone-based TEEs is made even more important by the discovery of vulnerabilities, with demonstrable exploits, that allow the extraction keys and other confidential information stored in the secure world by taking advantage of side channels such as cache timing [20, 38, 43]

3

THE PROBLEM AND ITS CHALLENGES

The guarantees provided by TEEs make them ideal for running security critical applications that benefit from running code in an isolated environment. However, unlike similar trusted hardware technologies such as Intel SGX or TPMs, ARM TrustZone doesn't provide cryptographic proof that the execution took place inside the TEE. This is especially important when the applications being executed perform system critical functions, as is the case for RV frameworks or embedded systems working as physical sensors. In these cases, the absence of attestation capabilities, makes these applications vulnerable to adversaries capable of simulating the execution inside a TEE, even if it takes place in an untrusted environment, potentially leaking confidential information or modifying recorded data. Attested computation is the process by which trusted hardware provides cryptographic proof that a computation or result was generated in a secure environment. This means that, when the environment in which the execution took place is trusted, a cryptographic proof for the execution of an application inside that environment could be used to verify that the properties the trusted hardware provides were enforced. The implementation of an attested computation scheme in ARM TrustZone would yield several advantages, from reinforcing the trust put on sensor informations [22] to ensuring the authenticity of results produced by monitors as part of an RV framework. However, the trust in the attestation mechanism still relies on the existence of a root of trust that ensures that the mechanism itself hasn't been compromised. On ARM System on Chips the secure boot mechanism, **i.e.** Trusted Board Boot, ensures the integrity of all the components executing in the secure world, namely the Trusted Firmware and Secure OS components, with the attestation mechanism being integrated in the latter.

On the other hand, implementing an attestation mechanism that only works for a fixed set of (trusted) applications greatly limits its usefulness, as only authorized developers, **i.e.** ones holding the private part of the key used to sign the TAs, have the ability to implement applications that will run inside the Trusted Execution Environment. To fix this, a solution must be found that gives a wider range of users the oportunity to use their own keys to sign Trusted Applications all the while making sure that non-trusted parties or ill-intentioned adversaries can't execute arbitrary code inside the TEE. In the context of a Runtime Verification framework, such capability would provide for a dynamic environment where monitors could be added on-demand, based on properties the system should uphold, as long as they had been validily signed.

Integrating these two mechanisms (attestation and support for third party TAs) with a key exchange protocol would allow Trusted Applications, such as monitors, to be remotely installed on TrustZone enabled devices while ensuring the confidentiality and authenticity of their outputs and the monitors themselves. This is similar to the remote attestation features provided by Intel SGX, and can be used to design and implement a Secure Outsourced Computation protocol, as shown in [4], in ARM TrustZone, where there is no native support for such feature. Thus, the focus of this work will be on the desgin and implementation of such procotol and it's applicability to the embedded system context. Attestation mechanisms using ARM TrustZone are not a novelty, works such as fTPM 2.0 [35], proposed and developed TPM like functionalities using ARM TrustZone as the trusted hardware base. Furthermore, the attestation mechanism proposed has a broad range of applicability, from the implementation of trusted sensors [22] with hardware support to its integration in a secure outsourced computation scheme [4]. Similar theoretical work has been developed showing the applicability of trusted hardware, namely in the development of a Universaly Composable [8] secure multiparty computation scheme using tamper proof hardware [19], removing the need for setup assumptions involving trusted third-parties.

3.1 ATTESTED COMPUTATION

In broad terms, an attested computation scheme combined with a key exchange used to establish a secure communcation channel should provide the following guarantees:

- **Attestation**: execution takes place inside the TEE
- **Honest Execution**: ensure the integrity of the programs to be executed inside the TEE
- **Client Authentication:** only programs signed with valid (client) keys [1] are executed
- **Confidentiality**: the I/O behaviour of the program being executed is known only to the TEE and the client

The attestation guarantee is a direct consequence of the trust placed in ARM TrustZone which ensures that data signed with the attestation key was computed inside the Secure World. This attestation mechanism also serves the purpose of authenticating the device, which will allow the implementation of an attested key exchange protocol to establish a secure communcation channel that ensures the confidentiality and authenticity of the I/O behaviour of the Trusted Applications invoked remotely by the Client. Honest execution follows as a consequence of establishing an authenticated (and encrypted) communication channel coupled with the attestation of the outputs generated by the TA. One important detail is that the guarantees and the protocols hereby described are secure under the assumption that ARM TrustZone is trusted and, when combined with Trusted Board Boot, provides isolation, confidentiality and integrity guarantees.

1 A valid key corresponds to a key generated by the `NewClient` algorithm, which is bound to an existing strong digital identity

3.1.1 *Specification*

Following the approach in [4], the SOC scheme can be seen as the composition of two (secure) protocols: a secure key exchange protocol and an attested computation scheme which, when combined, result in a SOC scheme.

Attested Key Exchange

An attested key exchange protocol for establishing a secure communication channel with the Secure World is defined by the following algorithm:

- **NewSessionKey(): Session Key** - initiate a key exchange with a remote instance of ARM TrustZone.

This key exchange can be seen as a simplified version of the one described in [4], where users are given the ability to tweak parameters used in the key exchange. While the ability to tweak these parameters is useful, it is not essential in providing the security properties and as such, it will be left out for simplicity. The fact that this protocol resorts to the attestation capabilities enunciated earlier gives it the ability to provide cryptographic proof that the key generation took place in a secure environment and that the storage of the shared secret is protected by the hardware and software mechanisms made available by ARM TrustZone.

Attested Computation

The proposed attested computation scheme, using ARM TrustZone, is defined by the following algorithms:

1. **NewClient(pub_k): Third Party Key Certificate** - key certification algorithm. Given an existing public key, it generates a public key certificate which will be used by the VerifyProg algorithm. The private part of the keypair allows the client to sign applications to be executed inside the TEE
2. **Compile(P, priv_k): TA binary** - program compilation and signing algorithm. Given a program P [2] and a private key (whose public key has been signed by the NewClient algorithm), outputs a compiled program P* signed by the private key
3. **Load(P*, pub_k, c)** - program loading algorithm. Given a program P*, produced by the Compile algorithm, a public key and the corresponding a public key certificate (generated by the NewClient algorithm)
4. **VerifyProg(P*, c): Boolean** - program verification algorithm. Given a program P*, produced by the Compile algorithm, and a public key certificate generated by the NewClient algorithm, verifies the chain of trust of c and the authenticity of P*

2 A program P in the context of OP-TEE corresponds to the ELF portion of a Trusted Application, while the compiled program P* is the Trusted Application itself.

5. **Attest(io): Signature** - attestation algorithm. Given a sequence of input/ouput pairs, produces a signature over these (io,)

6. **Verify(o*): Boolean** - verification algorithm. Given an attested output o*=(io,), produced by the Attest algorithm, verifies the signature

This set of algorithms can be seen as a high level API for interacting with the Secure Outsourced Computation protocol. The advantage of taking a modular approach, making the distinction between each phase of the protocol, is the simplification of the process of proving the security of such scheme, out of the scope of this work, along with the simplification of the development/implementation process. In addition, by proving and implementing each protocol individually, said protocols can be integrated in larger system with relative ease.

Pratical Considerations

As previously mentioned, an attested computation scheme aims to provide, via a cryptographic proof, guarantees that a computation took place in an isolated environment, where properties such as integrity and confidentiality are ensured via some of the already described mechanisms. There are several factors that should be taken into account when implementing this mechanism. Firstly, the integration of the attestation mechanism with existing TrustZone services, namely a Secure OS or the Secure Monitor. In the proposed architecture, the attestation itself will be performed by a Pseudo TA running as part of OP-TEE, which will provide an interface that allows TAs to request the attestation of a block of data or request a copy of the device certificate, in order to verify the attested data. The other important factor to consider regards the handling of the attestation material, namely the device certificate and the attestation key. The importance of correctly handling the latter is obvious, whose confidentiality and authenticity must be assured, as a leak of the key would allow an attacker to generate valid attestation signatures for computations that didn't take place in an isolated environment. The authenticity of the key, although not as obvious, should also be taken into account to prevent adversaries from replacing or modifying the attestation key. Thus, the key should be encrypted, using an authenticated encryption scheme, before being stored and should only be decrypted inside the secure environment using a device specific key accessible only to Secure World itself. The handling of the certificate, although not as paramount as the key, must also be considered due to its role in attestation. Many of these devices don't have a constant connection to the internet, which makes it hard to fetch a device certificate for use in verification, hence this should be stored locally so that it could be accessed on demand. However, the certificate should be stored in such a way to prevent malicious access, as compromising the authenticity could result in a Denial of Service where the verification of attested data would fail due to a corrupt certificate. To allow for a more flexible setup of the Secure World, namely allowing both libraries or OSs to provide attestation services, the handling of the attestation material should be implemented by the firmware running in EL3, which is standardized in ARMv8 through ATF-A. In this

scenario, when the device boots, the firmware will load the device certificate and the encrypted key and pass them to the Secure World. Another constraint that must be contemplated when expanding the set of features of OP-TEE are the security boundaries, which classify Client Applications as untrusted code, making them unsuitable for handling sensitive information. This is an important detail especially when deciding on where to integrate the attested key exchange protocol, as any application (code) running in the Normal World is vulnerable to adversaries and is out of the scope of the properties provided by ARM TrustZone. Furthermore, as a direct consequence of this boundary, the information that is stored, even if temporarily, in shared memory is susceptible to modification and it's confidentiality can't be assured.

3.1.2 *System Architecture*

The following diagram depicts the architecture of the system that provides the previously listed properties:

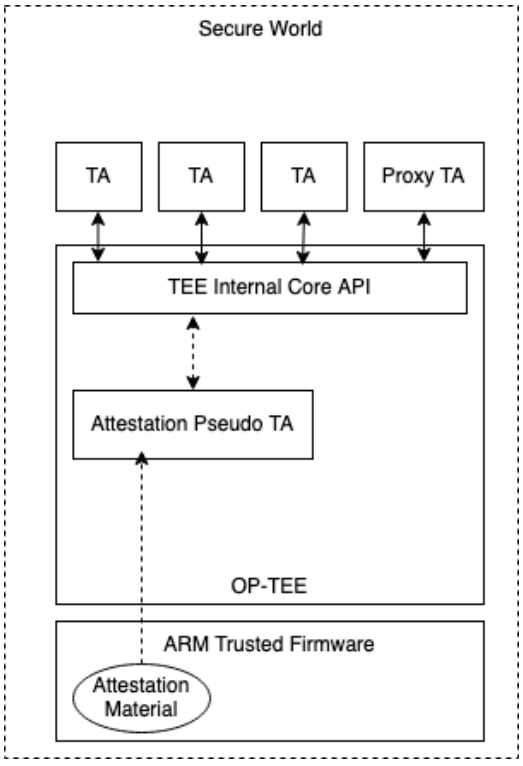


Figure 9: High Level Architecture of the System

In this architecture, the Trusted Firmware running at EL-3 holds the attestation material, which consists of the encrypted attestation key and the device certificate, and loads it into memory at boot. This data is passed to the attestation Pseudo TA, part of the OP-TEE kernel and executing at EL-1, which will use it to provide the attestation services to Trusted Applications running at EL-0 in the Secure World. In addition, the

Proxy TA which will be described in the following section, will perform the attested key exchange protocol to allow Clients to communicate via a secure channel with the TAs running the Secure World.

3.2 THIRD PARTY TRUSTED APPLICATIONS

The implementation of the previous scheme is not possible with the default feature set provided by OP-TEE as it only supports loading Trusted Applications signed by a "master" RSA key, whose public key is part of the OP-TEE binary and resides in main memory. This fixes the key used for verifying Trusted Applications after OP-TEE is compiled, greatly reducing the ability of legitimate users to load their own applications. The proposed solution involves the implementation of a mechanism that adds support for third party Trusted Applications, which are TAs signed with third party keys that have been signed by the "master" key, which results in the following "Chain of Trust":

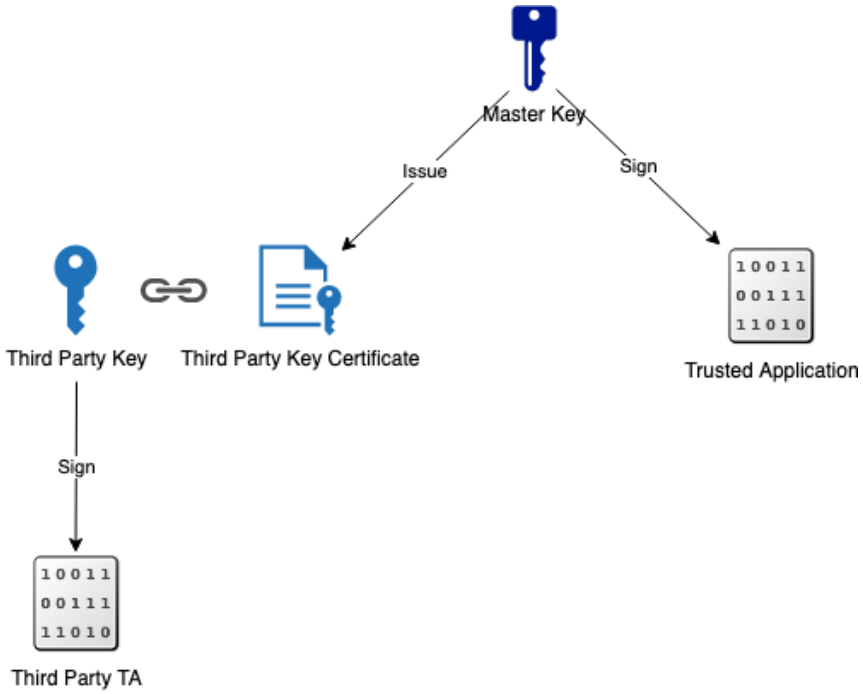


Figure 10: Chain of Trust for TAs

3.2.1 *Specification*

This mechanism requires the implementation of the following algorithms, already mentioned previously:

- NewClient(pub_k): key certification algorithm
- Compile(P, priv_k): program compilation and signing algorithm

- Load(P*, pub_k, c): program loading algorithm
- VerifyProg(P*, c): program verification algorithm

The first algorithm is executed only once per Client, as it corresponds to the certificate issuing operation. The second one is executed eachtime a Trusted Application is compiled, allowing the same key to sign different Trusted Applications. The last two algorithms are executed each time a third party TA is invoked, previous to its execution, as part of the load process.

3.2.2 *Practical Considerations*

Adding support for third party Trusted Applications presents several challenges which must be considered when designing the approach. On one hand, as of the time of this work, the mainline version of OP-TEE lacks support for features such as handling X.509 certificates or parsing ASN.1 signatures inside the OP-TEE core. Thus the third party key "certificates" must be represented using a simple format whose structure and limitations will be described in the following section. Another challenge introduced by this feature deals with the compilation process, as this is the only stage that can be executed outside the Secure World. Particularly, the compilation process requires the private key to be present to perform the signature operation, which is often performed in a non-secure environment, namely running on consumer OSs on x86 machines with no trusted hardware features. While this is out of the scope of this work, it should be a main concern for users wishing to develop their own Trusted Applications, as an attacker may gain access to a valid private key, which will allow him to execute his own code inside the TEE.

4

DEVELOPMENT

The Secure Outsourced Computation protocol developed is made up of three main components/modules that work together to provide users with the ability to securely communicate with Trusted Applications running in the Normal World, install their own TAs and resort to the attestation mechanism to generate cryptographic proof of secure computation performed inside the TEE. The core component of this protocol is the attestation mechanism itself, which is embedded in the OP-TEE core and provides the attestation services to Trusted Applications, Pseudo or Usermode ones. This component is used by the second module in the protocol, the attested key exchange module, implemented by a Usermode TA, to sign the key exchange parameters it generates. This module allows Client Applications to securely communicate with other Trusted Applications in the Secure World by using the shared secret that results from the key exchange to encrypt the data that is stored in shared memory as part of the parameter passing procedure. The last module expands the scope of Trusted Applications that can be executed in the Secure World. By default, OP-TEE only loads Trusted Applications signed with a "master" RSA key (during the compilation process), whose public part is stored in the Secure World and used in the verification process when a TA is invoked, which excludes Trusted Applications developed by legitimate users that don't hold this "master" key. To prevent sharing the key between several users, which could compromise it, the last module introduces the concept of third party TAs that are signed with certified keys **i.e.** keys that are trusted, via a "certificate", by the device. This allows for a broader range of applications to be installed on the Secure World while maintaining the ability to revoke certificates issued to developers that "betray" this trust relationship.

The remainder of this chapter will be dedicated to analysing each of this modules in detail.

4.1 DECISIONS

As previously stated, to allow for a more flexible setup of the Secure World, which can take the form of a library providing a simple attestation interface or an OS such as OP-TEE, the attestation scheme was divided in two logical parts. The first part corresponds to a initialization phase where the attestation material is loaded by the EL3 runtime firmware, and decrypted by the S-EL1 software (**i.e.** OP-TEE). The second part

corresponds to the attestation mechanism itself, which should provide an interface for signing data **i.e.** producing cryptographic proof of computation, and fetching a copy of the device certificate.

Integrating the attestation mechanism in a component such as ARM Trusted Firmware-A poses several challenges due how early in the boot phase this component starts executing. Firstly, the boot process in ARMv8 SoCs, as previously mentioned, differs according to the manufacturer, which leads to an heterogeneous landspace of possible configurations for this process. However, the BL31 stage is shared across different boards as, along with invoking the BL32 payload (**i.e.** OP-TEE) and the BL33 one (**i.e.** Normal World bootloader), it initializes the Secure Monitor running at EL3. Thus, integrating the attestation mechanism at this stage reduces the dependency in platform-specific configurations. The transition between ATF-A, in EL3, and OP-TEE, in S-EL1, as part of the boot flow is another point of interest on the initialization phase. This transition corresponds to a change of Exception Levels, which in turn requires the CPU context of the source Exception Level to be saved and the context of the target Exception Level to be restored in its place. Being the first invocation of OP-TEE, and thus the first switch to S-EL1, the context doesn't exist and must be created by the firmware, which also uses this initialization phase to set the arguments that will be passed to OP-TEE. In ARM Trusted Firmware-A this task is handled by the Secure-EL1 Payload Dispatcher (SPD), which is specific to the payload running at S-EL1 and, in addition to branching to the entrypoint of this payload, also exposes the Secure Monitor functionality to the Secure and Normal Worlds along with routing the Secure Monitor Calls to the Secure World. Hence, passing the buffer where attestation key and device certificate are stored, to OP-TEE will require setting one of the registers, of the S-EL1 context, as a pointer to this data.

On the other hand, the attestation mechanism was implemented as part of the OP-TEE core, making it available to TAs running in the Secure World while preventing untrusted code, such as the Client Applications running the Normal World, from interacting with this service. By being embedded in the OP-TEE core, the mechanism is able to run at the same Exception Level as the OS and have direct access to system calls, required for fetching the key used to decrypt the attestation material. The initialization protocol used by OP-TEE was also modified to call the function responsible for loading and decrypting the attestation key and the device certificate, passed by the SPD in BL31.

4.2 IMPLEMENTATION

Due to the impact and requirements of this scheme, and considering the development practices of OP-TEE and ARM Trusted Firmware, a new flag, `CFG_DEVICE_ATTESTATION`, was introduced which must be set at compile time to enable the attestation scheme:

```
ifdef CFG_DEVICE_ATTESTATION
        $(eval $(call add_define,CFG_DEVICE_ATTESTATION))
endif
```

which is ignored otherwise via `#ifdef` macro blocks.

4.2.1 *Initialization*

The initialization phase of the attestation mechanism was implemented in two different components, the ARM Trusted Firmware-A component, responsible for loading the device certificate and the encrypted key blob at boot time, and the OP-TEE component which deals with decrypting the attestation key and storing the device certificate in secure storage once the Normal World has executed.

In ATF-A, and due to the amount of SoCs supported, each platform has the ability to provide implementations for the functionality that depends on the underlying hardware, namely fetching the Root of Trust Public Key from a dedicated hardware module. These implementations take precedence over the default ones, which are annotated with the `#pragma weak` directive. To facilitate future porting of an attested computation scheme, a default implementation for fetching the attestation material, identitical to the one developed for the Pine A64+ and QEMU, was provided. This implementation adds three new platform specific functions that enable the firmware to retrieve the attestation key and the device certificate, either separately:

```
int plat_get_acc_key(void **ak_ptr, unsigned int *ak_len);
int plat_get_device_cert(void **dc_ptr, unsigned int *dc_len);
```

or concatenated in the same buffer:

```
int plat_get_dck_blob(void **dck_ptr, unsigned int *dc_len, unsigned int *ak_len);
```

The values for the device certificate and the encrypted attestation key returned by these implementations are stubs/fixed however, ideally these would be stored in a cryptographic module and would only be available to the firmware at boot.

Before invoking OP-TEE for the first time, the SPD retreives the device certificate, the encrypted attestation key and the public key in concatenated form, and passes a pointer to this buffer, as well as the size of the certificate and key, to OP-TEE:

```
rc = plat_get_dck_blob(&dck_blob, &dc_len, &ak_len);
if(rc)
    WARN("Failed to fetch the attestation material\n");

assert(rc == 0);
```

```
optee_entry_point->args.arg3 = (uint64_t) dck_blob;
optee_entry_point->args.arg4 = dc_len;
optee_entry_point->args.arg5 = ak_len;
```

The `assert` statement halts the boot process if the device certificate or attestation key is unavailable, which is expected due to the integral role these play when enabled.

As explained in the previous section, before branching to OP-TEE, the SPD must save the CPU context for the current EL and create a new one for the target, S-EL1. Being the first EL switch to this level, there is no CPU context for the `el3_exit()` function to restore, so the variable `optee_entry_point`, a structure of type `entry_point_info_t` used by ATF-A to progress through the different boot stages, is used by the Secure-EL1 Payload Dispatcher to populate the new CPU context. The entry point structure stores several items of information, from the address of first instruction of OP-TEE (**i.e** the entry point) to the arguments it expects as part of the initialization phase, which are a direct mapping to ARM registers. Upon switching Exception Levels and entering OP-TEE, these arguments are written to temporary registers to prevent them from being overwritten by subsequent functions:

```
#if defined(CFG_DEVICE_ATTESTATION)
    mov x21, x3     /* Attestation blob address */
    mov x22, x4     /* Device certificate size */
    mov x23, x5     /* Attestation key size */
#endif
```

The final part of the initialization phase involves loading and decrypting the attestation key and the device certificate to OP-TEE's memory space. It's important to make this distinction between the Secure Monitors and OP-TEE's memory space due to the restrictions imposed by TrustZone, via the NS bit and different MMUs, which segment the physical memory in regions allocated to a single Exception Level. As a consequence, addresses in EL3 cannot be accessed by instructions executing with a lower privilege. Since this is the case for the attestation key and the device certificate, which were placed in a memory region used by EL3, the buffers must be relocated to a valid region of the S-EL1 memory space. However, this regions are configured by OP-TEE as part of the MMU configuration process, which is then enabled to enforce memory isolation. This introduced the need to implement a relocation function that is called before the MMU is enabled, so it can read from the memory allocated to EL3, but after the MMU has been configured, so that it has access to the addresses that make up the configured memory regions.

```
    adr     x1, boot_mmu_config
    bl      core_init_mmu_map

#ifdef CFG_DEVICE_ATTESTATION
```

```
    /*
     * Copy the device certificate from EL3 memory to the shared
     * memory region between EL1 and S-EL1
     */
    mov x0, x21
    mov x1, x22
    mov x2, x23
    bl relocate_dckb_vec
#endif
...
    bl    enable_mmu
```

Amongst the different regions set up by OP-TEE, the one used to temporarily store the attestation material corresponds to the memory used to share data with the Normal World, whose base (physical) address is identified by the `TEE_SHMEM_START` macro. At this stage of the boot process, no untrusted code has had the chance to execute and only one CPU is executing instructions, so it's safe to use this region. The relocation function, `relocate_dckb_vec`, is given the physical address of the attestation material (in EL3), the size of the device certificate and of the attestation key and copies this data to the start of the shared memory region, checking if it fits beforehand.

```
LOCAL_FUNC relocate_dckb_vec , :
    mov_imm x3, TEE_SHMEM_START
    mov_imm x4, TEE_SHMEM_SIZE
    add x1, x1, x2
    cmp x1, x4
    b.gt     3f

    mov x4, x3
    add x5, x0, x1
    sub x5, x5, #16

1:  ldp x6, x7, [src], #16
    stp x6, x7, [dst], #16
    cmp src, x5
    b.lt    1b

    add x5, x5, #16
    b       #12
```

```
2:  ldr w6, [src], #4
    str w6, [dst], #4
    cmp src, x5
    b.lt    2b

3:  mov x21, x3
    sub x1, x1, x2
    mov x22, x1
    mov x23, x2
    ret
END_FUNC relocate_dckb_vec
```

In the initial stages of development, each loop iteration would load/store a single 64 bit value but, in order to amortize the cost of the memory access, as well as reduce the total number of instructions by half, the function was optimized to load/store a pair of 64 bit values at a time, using the `ldp` and `strp` instructions. This optimization, although minor, is important to ensure the system is able to boot in the smallest amount of time possible, something which is necessary to ensure minimal downtime in systems that perform mission-critical functions. The `src` and `dst` values are macros set to `x0` and `x4` respectively. After copying the attestation material to shared memory, the pointer is updated with the base (physical) address of this region, which will be translated to a virtual one once the MMU is enabled. The (shared) memory used for temporarily storing the attestation material during then initialization phase is zero filled before handing execution over to the Normal World.

The decryption of the key is handled by the attestation TA, namely when the `import_attestation_key` is invoked as part of OP-TEE's boot process, which serves the purpose of loading the certificate to memory as well as the attestation key. This functions is invoked as part of the initialization procedure of the Pseudo TA, which also zero fills the shared memory when the function returns:

```
static void init_attestation_ta(unsigned long dcak_b, size_t dc_l, size_t ak_l){
    TEE_Result res;
    uint64_t* dcak_p = phys_to_virt(dcak_b, MEM_AREA_NSEC_SHM);

    DMSG("Attestation blob at address: %#"PRIx64, dcak_b);

    res = import_attestation_key(dcak_p, dc_l, ak_l);
    if(res)
        DMSG("Failed to import the attestation certificate");

    memset(dcak_p, 0, dc_l+ak_l); //zero fill the buffer
```

```
    assert(res==0);
}
```

One important detail of this block of code is the fact that the `assert` invocation was placed after the zero fill operation. This ensures that the zero fill takes place even if the TA fails to import the attestation key, preventing attackers from reading the memory region where the attestation material was temporarily stored.

The plaintext corresponding to the decrypted attestation key, which is encrypted using AES in CTR mode, is only written in secure RAM , ensuring that this value never leaves the memory region allocated to OP-TEE and that it cannot be accessed from the Normal World. This operation is carried out by the `bin_2_ecckey` function, which also loads the "plain text" public key to secure RAM:

```
static inline TEE_Result bin_2_ecckey(uint8_t *blob, size_t ak_l){
    TEE_Result res = TEE_SUCCESS;
    size_t key_size = ak_l/3;

    res = decrypt_ak(blob, key_size);
    if(res)
        return res;

    ctx_i.kp = calloc(1, sizeof(struct ecc_keypair));
    if(!ctx_i.kp)
        return TEE_ERROR_OUT_OF_MEMORY;

    res = crypto_acipher_alloc_ecc_keypair(ctx_i.kp, 256);
    if(res)
        return res;

    res = crypto_bignum_bin2bn(blob, key_size, ctx_i.kp->d);
    if(res)
        return res;

    blob += key_size;
    res = crypto_bignum_bin2bn(blob, key_size, ctx_i.kp->x);
    if(res)
        return res;

    blob += key_size;
    res = crypto_bignum_bin2bn(blob, key_size, ctx_i.kp->y);
    if(res)
```

```
        return res;

    ctx_i.kp->curve = TEE_ECC_CURVE_NIST_P256;

    return res;
}
```

The `inline` modifier, which replaces a function's invocation with its body, removes the overhead introduced by setting up the stack when invoking a function. As this function invocation happens only once, this modification won't increase the size of the compiled binary, which is an important factor when working with resource constrained targets, such as embedded systems.

The key used for decryption is derived from a hardware key, unique to the device and set by the manufacturer, called Hardware Unique Key (HUK). The derived key is obtained by invoking the `huk_subkey_derive` function and specifying the use of the derived key, from a set of possible values defined in an `enum`:

```
TEE_Result huk_subkey_derive(enum huk_subkey_usage usage,
                const void *const_data, size_t const_data_len,
                uint8_t *subkey, size_t subkey_len);
...
res = huk_subkey_derive(HUK_SUBKEY_ACC, NULL, 0, key, b_len);
```

Similarly to the attestation key, the device attestation certificate is loaded into secure RAM, with the attestation context storing pointers to both buffers:

```
struct attest_ctx{
    struct ecc_keypair *kp;
    void *dc;
    size_t dc_l;
};
```

After the first invocation to the attestation service is made (from a Usermode TA), the device certificate is stored in secure storage and the correspoding memory is freed. The reason for not implementing this as part of the initialization procedure stems from the fact that secure storage operations are routed to the Normal World, hence it must be executing to store the certificate. To ensure the certificate is only stored at the first call, this code is executed as part of the handler function for the entry point creation, registered at compile time:

```
pseudo_ta_register(.uuid = PTA_ATTEST_UUID, .name = PTA_NAME,
                   .flags = PTA_DEFAULT_FLAGS,
```

```
                    .create_entry_point = create,
                    .open_session_entry_point = open_session,
                    .invoke_command_entry_point = invoke_command);
```

The `pseudo_ta_register` function also stores other metadata about the PTA, namely it's flags, UUID and the handlers for other events, such as command invocations.

4.2.2 *Attestation*

A Pseudo TA embedded in the OP-TEE core is responsible for providing attestation services to Usermode TAs running in the secure world. To ensure that only Usermode TAs have access to the attestation services, when a new session is opened, the `open_session` handler checks if the caller session context matches that of a Usermode TA:

```
static TEE_Result open_session(uint32_t pt __unused,
                               TEE_Param params[TEE_NUM_PARAMS] __unused,
                               void **psess_ctx){
    TEE_Result res = TEE_SUCCESS;
    struct tee_ta_session *s = tee_ta_get_calling_session();

    if(!s || !is_user_ta_ctx(s->ctx))
        return TEE_ERROR_ACCESS_DENIED;
...
}
```

This effectively reduces the attack surface by limiting the amount of TAs that are able to request attestation services, which if exploited could allow an attacker to sign arbitrary data. These services are made available through two distintct commands:

```
switch(cmd){
    case ATTEST_CMD_SIGN:
        return sign_blob(pt, params);
    case ATTEST_CMD_GET_CERT:
        return dump_dc(pt, params);
    default:
        break;
}
```

The `ATTEST_CMD_SIGN` command exposes the attestation mechanism itself, while the `ATTEST_CMD_GET_CERT` command allows TAs to fetch the device certificate to verify an attestation signature.

Regarding the `sign_blob` function, there are some details that ought to be mentioned:

```
static TEE_Result sign_blob(uint32_t pt, TEE_Param params[4]){
    TEE_Result res = TEE_SUCCESS;
    void *hash = NULL;
    uint32_t e_pt = TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_INPUT,  //cert blob
                                    TEE_PARAM_TYPE_MEMREF_OUTPUT, //signature
                                    TEE_PARAM_TYPE_NONE,
                                    TEE_PARAM_TYPE_NONE);

    uint32_t ecdsa_alg = TEE_ALG_ECDSA_P256, hash_alg;
    size_t hash_size = 0;

    ...

    hash_alg = TEE_DIGEST_HASH_TO_ALGO(ecdsa_alg) + 1;

    ...
    if(crypto_hash_alloc_ctx(&hash_ctx, TEE_ALG_SHA256) || crypto_hash_init(hash_ctx))
        return TEE_ERROR_GENERIC;

    ...

    res = tee_hash_createdigest(hash_alg, params[0].memref.buffer, params[0].memref.size
                                hash, hash_alg);
    if(res)
        goto out;

    res = crypto_acipher_ecc_sign_asn(ecdsa_alg, ctx_i.kp,
                                      hash_alg,
                                      hash, hash_size,
                                      params[1].memref.buffer, &(params[1].memref.size))
    ...
}
```

Firstly, the use of the SHA-256 hash function to produce a digest of the input buffer, which will be signed by the signature algorithm. Secondly, the use of ECDSA, over RSA, as the signature algorithm allows for smaller keysizes, while maintaining the security level. This reduced key size is paramount in embedded systems due to the lack of resources, namely main memory. The signature operation itself is

performed by the `crypto_acipher_ecc_sign_asn` function, which was implemented to add support for generating signatures in the ASN.1 format, required for cross compatibility between different cryptographic libraries, namely the ones external to OP-TEE. The default ECC signing implementation used by OP-TEE, `crypto_acipher_ecc_sign`, generates signatures in their raw format. In the case of ECDSA, this means that the values and , represented internally by the `mbedtls_mpi` structure, are copied to the signature buffer in their raw/byte format, using the `mbedtls_mpi_write_binary` function, which can result incompatibilities with other cryptographic libraries:

```
TEE_Result crypto_acipher_ecc_sign(uint32_t algo, struct ecc_keypair *key,
                   const uint8_t *msg, size_t msg_len,
                   uint8_t *sig, size_t *sig_len)
{
    ...
    lmd_res = mbedtls_ecdsa_sign(&ecdsa.grp, &r, &s, &ecdsa.d, msg,
                                 msg_len, mbd_rand, NULL);
    if (lmd_res == 0) {
        *sig_len = 2 * key_size_bytes;
        memset(sig, 0, *sig_len);
        mbedtls_mpi_write_binary(&r, sig + *sig_len / 2 -
                     mbedtls_mpi_size(&r),
                     mbedtls_mpi_size(&r));

        mbedtls_mpi_write_binary(&s, sig + *sig_len -
                     mbedtls_mpi_size(&s),
                     mbedtls_mpi_size(&s));
        res = TEE_SUCCESS;
    ...
}
```

To circumvent this, the `crypto_acipher_ecc_sign_asn` invokes a different function to generate the signature, `mbedtls_ecdsa_write_signature`, which computes the message signature and converts it to ASN.1:

```
TEE_Result crypto_acipher_ecc_sign_asn(uint32_t algo, struct ecc_keypair *key,
                                       uint32_t md_alg,
                                       const uint8_t *msg, size_t msg_len,
                                       uint8_t *sig, size_t *sig_len)
{
    ...
```

```
    lmd_res = mbedtls_ecdsa_write_signature(&ecdsa,
                                            TEE_ALG_GET_MAIN_ALG(md_alg) + MBEDTLS_MD_MD
                                            msg, msg_len,
                                            sig, sig_len,
                                            mbd_rand, NULL);
    ...
}
```

While it could be thought that the default signing function could be modified instead of implementing a new one, this would break other functionalities of OP-TEE that used interface, as these are system calls provided by OP-TEE. Similarly to other functions pertaining to the attestation mechanism, this new function is only compiled when the attestation mechanism is enabled, reducing the size of the binary.

The `dump_dc` function fetches the DER encoded device certificate from secure storage, where it is encrypted, and returns a copy of it.

4.2.3 *Tools*

Although not directly related to the attestation mechanism itself, the generation of attestation keys, the corresponding device certicates, as well as the full certificate chain, and the subsequent conversion of these in representations compatible with C variables **i.e.** arrays of raw bytes, can be a complex task. Thus, a system for generating and encrypting the attestation key, along with a device certificate is provided as part of this work. It's worth pointing out that the encryption of the key is only possible due to the use of a fixed HUK, which in turn results in the same (sub)key being derived. To provide a greater degree of flexibility, the encryption function emulates the generation procedure, using the same HUK key as a starting point.

4.2.4 *Third Party TA Support*

As described in section 2.5.4, Trusted Applications are ELF binaries with additional metadata used by OP-TEE to verify their authenticity. For this verification to succeed they must be signed with an RSA key, and optionally encrypted, whose public part must be trusted by OP-TEE. Although this works as an effective access control mechanism, which limits the code that can execute in the Secure World to the one signed by a single party, it also hinders the ability of legitimate users (whether individuals or organizations) to develop and run their own applications inside a Trusted Execution Environment. This behaviour is unlike the one displayed by Intel SGX, which allows developers to mark blocks of code to be executed in a the secure enclave, benefiting from all the security guarantees it provides. However, allowing any code to execute inside TrustZone poses a risk to the isolation properties it claims, as Trusted Applications have access to system

level functionality, such as the aforementioned attestation mechanism. To overcome this, a compromise has to be made that takes into account both usability and security, in order to increase the range of usage of TrustZone-enabled devices. The support for third party TAs was implemented with these factors in mind, allowing certified developers, **i.e.** with validly signed RSA keys, to sign Trusted Applications with a key different than the one installed by the manufacturer.

Binary Structure

It's clear that the implementation of such feature involves modifying the binary structure of Trusted Applications described in 7 as well as the TA loading and verification process. The new execution flow for this process is ilustrated in the following diagram:

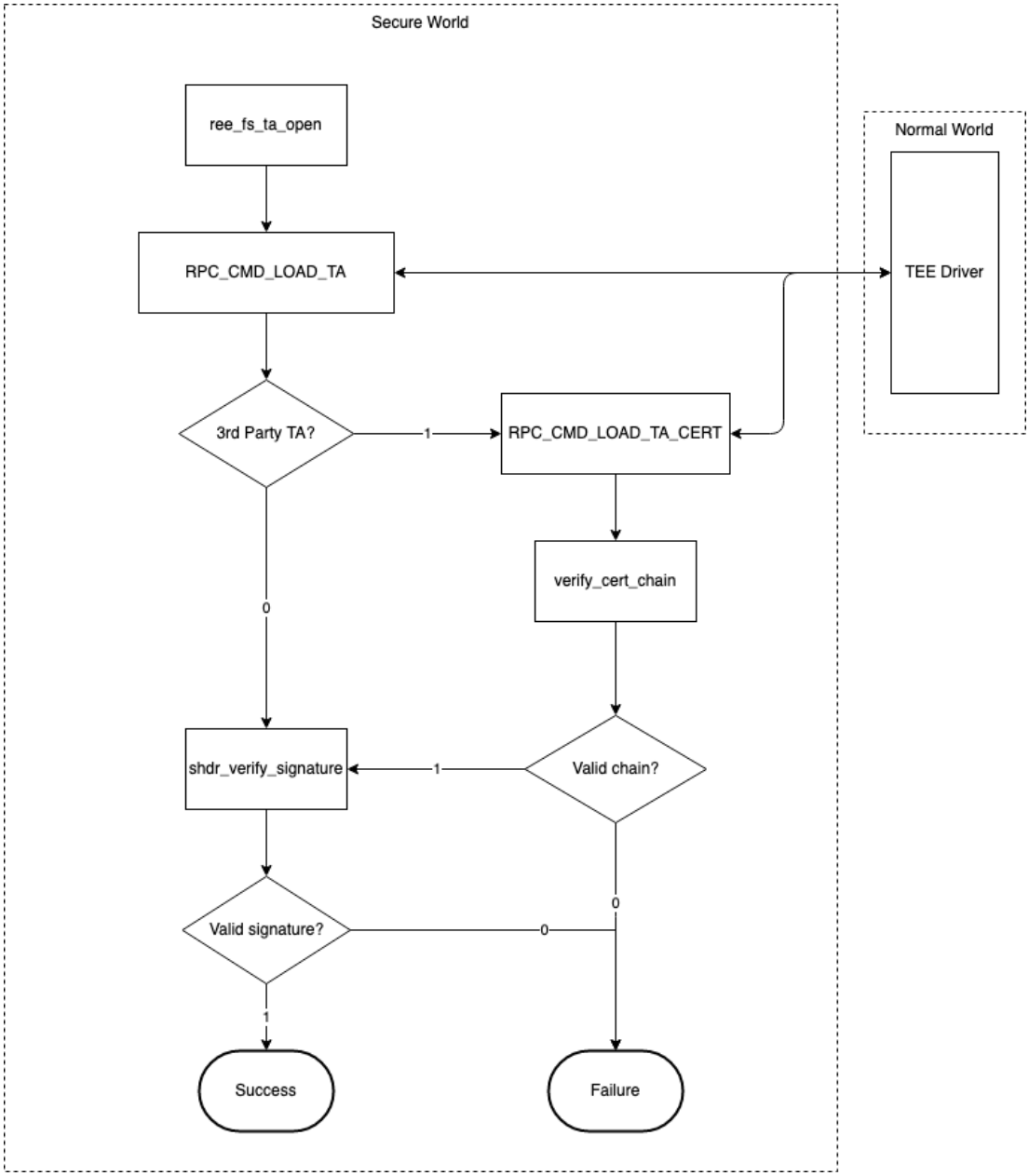


Figure 11: TA load flow

When an application, whether a Client of a Trusted Application, issues a comand to a TA, OP-TEE searches in secure memory for all open TAs to check whether the requested TA has been loaded. In the cases where it hasn't, the `ree_fs_ta_open` function is invoked which performs an RPC to load the TA from the REE filesystem, in the Normal World, into shared memory. It then uses the default signing key to check the signature on the binary image, moving it from shared memory to secure memory beforehand to prevent modifications. If the signature verifcation succeeds, the TA is executed as requested. The support for third party TAs signed with valid public keys was implemented in the form of a new Trusted Application type, which when loaded triggers an additional Remote Procedure Call to the Normal World, that loads a "certificate" for

the public key used in the verification process. This certificate is then verified using the default RSA key, used to sign TAs, and if the verification succeeds the TA signature is verified using the third party public key. Due to the amount of information necessary to deal with third party keys, adding support for this feature required modifying the existing Signed Header structure to allow OP-TEE distinguish between a third party TA and a "normal" one, as well as perform the deserialization and key verification procedures in the former case. After examining the Signed Header structure:

```
struct shdr {
    uint32_t magic;
    uint32_t img_type;
    uint32_t img_size;
    uint32_t algo;
    uint16_t hash_size;
    uint16_t sig_size;
};
```

the approach used for adding support for third party TAs involved the creation of two new types of TA binaries: `SHDR_THIRD_PARTY_TA` and `SHDR_THIRD_PARTY_ENC_TA`, to identify Trusted Applications signed with keys other than the default one, both Bootstrap and Encrypted TAs respectively. Furthermore, the metadata necessary to deal with third party "certificates"/keys was stored in a new sub-header, which changed the binary format described in 7 by "injecting" this sub-header before the Bootstrap TA sub-header:

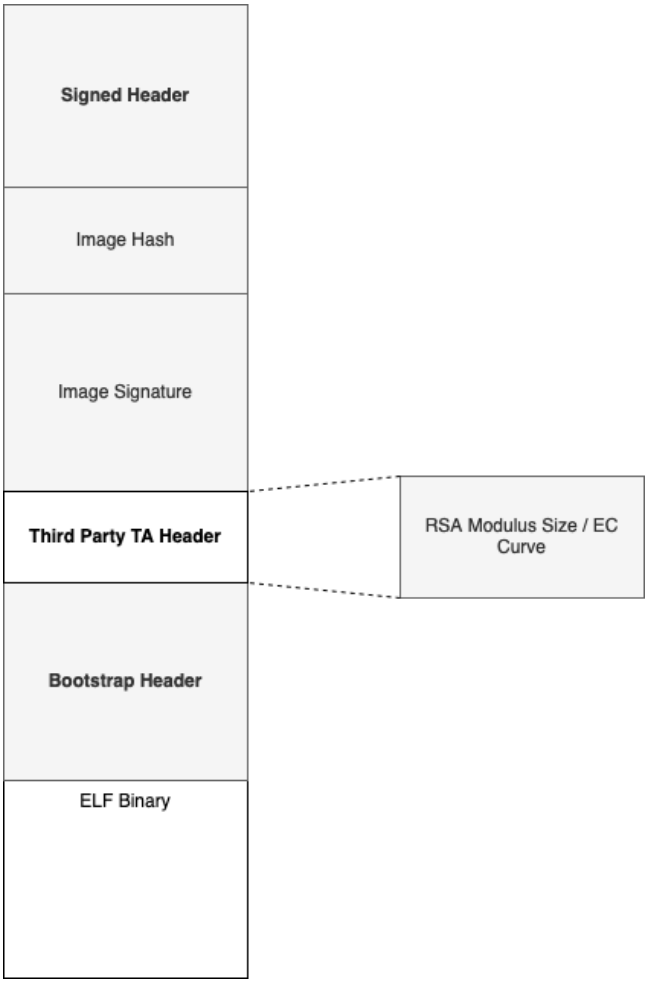


Figure 12: Third party TA binary format

This sub-header contains a structure with a single union member that can either store the size of the RSA modulus (in bytes) or, for added flexibility, the elliptic curve used by the signing algorithm, when dealing with an ECDSA key:

```
struct shdr_thirdparty_ta {
    union {
        size_t ta_pub_key_modulus_size;
        uint32_t ec_curve;
    } key_info;
};
```

The use of a separate header for Third Party TAs ensures backwards compatibility with non-third party TAs developed for versions of OP-TEE that don't support this feature as all the (additional) data is stored in a separate header.

Loading procedure

The `shdr_verify_signature` function is invoked each time a TA is requested and is not found in secure memory. In addition to loading the binary from the REE (or RPMB) filesystem, this function also performs the necessary security checks to ensure the authenticity (and confidentiality) of Trusted Applications. One such security check is the signature verification operation, which takes the signature present in the TA metadata, as well as the image hash, and validates it, using the default RSA public key (the hash of the TA image is calculated at a later stage of this process, by the `check_digest` function before invoking the TA). To handle third party Trusted Applications, the `ree_fs_ta_open` function was modified to request tee-supplicant for the third party key "certificate":

```
    /* Request TA from tee-supplicant */
    res = rpc_load(uuid, &ta, &ta_size, &mobj);

    ...

#ifdef CFG_THIRD_PARTY_TA
    if(shdr->img_type == SHDR_THIRD_PARTY_TA ||
       shdr->img_type == SHDR_THIRD_PARTY_ENC_TA){
        res = rpc_load_cert(uuid, &custom_key, &custom_key_len, &ta_cert_mobj);

        ...

        custom_key = cert_alloc_and_copy(custom_key, custom_key_len);

        ...

        res = verify_cert(custom_key, custom_key_len, shdr->sig_size, shdr->algo);
    }
#endif

    ...
    res = shdr_verify_signature(shdr, tp_hdr, custom_key);
```

The `rpc_load_cert` function performs three different Remote Procedure Calls to the Normal World, the first one to request the size of the "certificate", the second one to allocate shared memory to copy the certificate into and the last one to load the actual "certificate":

```
static TEE_Result rpc_load_cert(const TEE_UUID *uuid, void **payload, size_t *len, struc
```

```
    TEE_Result res = TEE_SUCCESS;
    struct thread_param params[2];
    ...
    res = thread_rpc_cmd(OPTEE_RPC_CMD_LOAD_TA_CERT, 2, params);
    ...
    mobj = thread_rpc_alloc_payload(params[1].u.memref.size);
    ...
    res = thread_rpc_cmd(OPTEE_RPC_CMD_LOAD_TA_CERT, 2, params);
    ...
    *payload = mobj_get_va(mobj, 0);
    ...
}
```

Each of these Remote Procedure Calls triggers a context switch to the Normal World, where the OP-TEE driver will store the data associated with each call, **i.e.** command ID and parameters, in a queue. This queue is constantly monitored by the tee-supplicant threads, which process each request on the queue and return the result to the Secure World via the OP-TEE driver. To load a TA certificate, when the `OPTEE_RPC_CMD_LOAD_TA_CERT` is received, the (tee-supplicant) thread invokes the `load_ta_cert` function:

```
static bool process_one_request(struct thread_arg *arg){
    ...
    if (!read_request(arg->fd, &request))
        return false;

    if (!find_params(&request, &func, &num_params, &params, &num_meta))
        return false;
    ...
    switch (func) {
        case OPTEE_MSG_RPC_CMD_LOAD_TA:
            ret = load_ta(num_params, params);
            break;
        ...
        case OPTEE_MSG_RPC_CMD_LOAD_TA_CERT:
            ret = load_ta_cert(num_params, params);
            break;
        ...
    }
    ...
```

```
}
```

which, like the `load_ta` function which loads a Trusted Application from the REE filesystem, returns the size of the certificate when no pointer to shared memory is passed or writes the certificate to this buffer. To add support for loading TA certificates, the function used for loading the actual Trusted Applications, `TEECI_LoadSecureModule`, was modified to receive a flag indicating whether the UUID it received referred to the TA itself, or to its certificate:

```
static uint32_t load_ta(size_t num_params, struct tee_ioctl_param *params)
{
    ...
    ta_found = TEECI_LoadSecureModule(ta_dir, &uuid, shm_ta.buffer, &size, 0);
    ...
}

static uint32_t load_ta_cert(size_t num_params, struct tee_ioctl_param *params)
{
    ...
    cert_found = TEECI_LoadSecureModule(ta_dir, &uuid, shm_ta.buffer, &size, 1);
    ...
}
```

This function servers as a wrapper for the `try_load_secure_module` function which looks for the file in the diretory pointed to configured in the `TEEC_LOAD_PATH` macro, which defaults to "/lib/optee_armtz", and loads the file if it is found, converting the raw UUID to it's text format, as per RFC-4122:

```
static int try_load_secure_module(const char* prefix,
                                  const char* dev_path,
                                  const TEEC_UUID *destination, void *ta,
                                  size_t *ta_size,
                                  uint8_t cert)
{
    n = snprintf(fname, PATH_MAX,
                 "%s/%s/%08x-%04x-%04x-%02x%02x%s%02x%02x%02x%02x%02x%02x.%s",
                 prefix, dev_path,
                 destination->timeLow,
                 destination->timeMid,
                 destination->timeHiAndVersion,
                 destination->clockSeqAndNode[0],
```

```
                destination->clockSeqAndNode[1],
                first_try ? "-" : "",
                destination->clockSeqAndNode[2],
                destination->clockSeqAndNode[3],
                destination->clockSeqAndNode[4],
                destination->clockSeqAndNode[5],
                destination->clockSeqAndNode[6],
                destination->clockSeqAndNode[7],
                cert ? "cert" : "ta");
    ...
    file = fopen(fname, "r");
    if (file == NULL) {
        DMSG("failed to open the ta %s TA-file", fname);
    ...
}
```

After returning from the Remote Procedure Call, the address of the certificate in shared memory is stored in `payload` parameter. Similarly to what happens with TA binaries, the certificate is copied to secure memory before performing the verification operations, to prevent the Normal World from modifying it:

```
custom_key = cert_alloc_and_copy(custom_key, custom_key_len);
```

If the verification succeeds, the third party header offset, from the base address of the TA, is calculated and used to parse the header which is passed to the Signed Header verification function:

```
offs = SHDR_GET_SIZE(shdr);

tp_hdr = calloc(sizeof(*tp_hdr), 1);
if(!tp_hdr)
    goto error_free_payload;

memcpy(tp_hdr, (uint8_t*)ta + offs, sizeof(*tp_hdr));
...
res = shdr_verify_signature(shdr, tp_hdr, custom_key);
```

This function checks the type of the Trusted Application before verifying the signature in order to determine which key to use. If the TA is a third party one, it invokes the `extract_key` function which converts a byte buffer into an RSA key:

```
TEE_Result shdr_verify_signature(const struct shdr *shdr,
                                 struct shdr_thirdparty_ta *tp_hdr,
                                 void *custom_key)
{
    ...
    if(shdr->img_type != SHDR_THIRD_PARTY_TA &&
       shdr->img_type != SHDR_THIRD_PARTY_ENC_TA){
        res = crypto_acipher_alloc_rsa_public_key(&key, shdr->sig_size);
        if (res)
            return TEE_ERROR_SECURITY;

        res = crypto_bignum_bin2bn((uint8_t *)&e, sizeof(e), key.e);
        if (res)
            goto out;
        res = crypto_bignum_bin2bn(ta_pub_key_modulus, ta_pub_key_modulus_size,
                                   key.n);
        if (res)
            goto out;
    } else {
        res = extract_key(tp_hdr, shdr->sig_size, custom_key, &key);
        if(res)
            goto out;
    }
    res = crypto_acipher_rsassa_verify(shdr->algo, &key, shdr->hash_size,
                                       SHDR_GET_HASH(shdr), shdr->hash_size,
                                       SHDR_GET_SIG(shdr), shdr->sig_size);
    ...
}
```

The advantage of this modular implementation is the ability to use different formats for serializing the third party key, which only require modifications to the `extract_key` function. Furthermore, adding support for elliptic curve signature algorithms only involves modifying the `shdr_verify_signature` function and the `extract_key` function.

Build System

To generate the binary structure that supports Trusted Applications, OP-TEE comes with a build system configured to use GNU-make for the compilation of the ELF binaries which are then piped into a Python

script that converts the metadata in the format represented by the C structures, into raw bytes that are written to files. A normal invocation of this script, to generate a generic Bootstrap TA would be the following:

```
./sign_encrypt.py --key <ta-sign-key-pem> --uuid <ta-uuid> --ta-version <ta-version> \
                  --in <ta-stripped-elf-file>
```

which outputs a signed Trusted Application with the name **<uuid>.ta**.

The conversion process is achieved via the `struct` library by packing the structure members into a byte array in big endian order. The SHA-256 hash of this array is then signed using the key loaded from the file pointed to by the `--key` flag, and both the hash and the signature are written in the output file.

To create a third party TA the script was modified to receive four additional (optional) arguments:

- `-tp`: indicates that key used to sign the TA is a third party one
- `--tp-key-type`: sets the type of the key (RSA by default)
- `--tp-cert`: path to an existing key certificate
- `--master-key`: path to the master (default) key to generate a key certificate if none exists

Although the last two are mutually exclusive, one of them has to be passed onto the script for the signing process to be successful. Upon receiving the `-tp` flag, the script assumes the key pointed by the `--key` flag is a third party one and, in addition to loading it according to it's type:

```
if args.tp:
    pub_key = key.publickey()
    if args.tp_key_type == 'ecdsa':
        key_info = TEE_CURVES_MAP[pub_key.curve]
    else:
        key_info = pub_key.size_in_bytes()
    shdr_tp_key_info = struct.pack('<Q', key_info)
```

it also generates a key certificate signed by the master key (`--master-key`) or it makes a copy of the certificate pointed to by the `--tp-cert` flag:

```
def create_ta_cert():
    if args.tp_cert:
        shutil.copy(args.tp_cert, str(args.uuid) + '.cert')
    else:
        with open(args.master_key, 'rb') as f:
            master_key = RSA.import_key(f.read())
```

```
if not(master_key.has_private()):
    logger.error('Provided key can\'t be used for signing')
    sys.exit(1)

if args.tp_key_type == 'ecdsa':
    raw_key = serialize_ecdsa_key(key)
else:
    raw_key = serialize_rsa_key(key)

md = SHA256.new(raw_key)
sig = pss.new(master_key).sign(md) #sign third party key
with open(str(args.uuid) + '.cert', 'wb') as f:
    f.write(sig)
    f.write(raw_key)
```

The `args` object holds the value of the flags supported by the script. Thus, to create a third party Trusted Application, the script should be executed as follows:

```
./sign_encrypt.py --key <ta-sign-key-pem> \
                  -tp --tp-key-type <key-type> --tp-cert <key-cert> \
                  --uuid <ta-uuid> --ta-version <ta-version> \
                  --in <ta-stripped-elf-file>
```

Alternatively the development kit provided by OP-TEE was also modified to check for the presence of the `CFG_TP_TA` macro and automatically generate the previous command. In this situation, to use an existing certificate the value of the `TP_CERT` macro should be set otherwise a master key, which defaults to the TA signing key, should be specified with the `M_KEY` variable. All these macros should be set in the Makefile of the TA.

4.2.5 *Attested Key Exchange*

The final piece of the "puzzle" for developing a Secure Outsourced Computation scheme is the implementation of an attested key exchange protocol. This protocol, when combined with a symmetric cipher such as AES in an authenticated encryption mode, such as GCM or CCM, provides both authentication and confidentiality guarantees of the data exchanged. One of the key differences between this protocol and a "normal" key exchange protocol stem from the attestation of the public data generated by OP-TEE when performing the handshake. In "normal" key exchange protocols, such as Station-to-Station (STS), which aim to provide

authentication in addition to confidentiality, ensuring protection against Man-in-the-Middle attacks where an attacker could intercept the communications and act as the server/client by sending his own public parameter, the server and the client exchange public key certificates in addition to the public parameters of the key exchange. These certificates are then used by each party to verify the authenticity of the public parameter it received. Attestation aims to provide stronger security guarantees, ensuring not only the origin of the public parameter generated by OP-TEE but also providing cryptographic proof that the generation took place inside the Trusted Execution Environment.

Due to the nature of this protocol, which involves accepting remote connections from (possibily) malicious clients, Pseudo Trusted Applications are unsuitable for its implementation, as this would greatly increase the attack surface of OP-TEE by exposing the kernel, of which Pseudo TAs are a part of, to these remote parties. As such, the decision was made to implement this functionality as a Trusted Application running in EL0, reducing the impact of potential vulnerabilities as well increasing the flexibility of the key exchange protocol by separating it from the kernel itself. This TA allows Client Application to remotely issue commands to other Trusted Applications running in the secure world, whilst ensuring the confidentiality and integrity of the inputs and outputs.

For simplicity sake, the communication between the Client Application and the Trusted Application is achieved via the Normal World TEE library and driver, however, a remote setup could be achieved by developing a Client Application that acted as a proxy for the communications between the Client and the Trusted Application. While at first it might seem like this approach would introduce a security problem, by considering the proxy as a Man-in-the-Middle, it's possible to map this situation into one where an active adversary would try to listen or modify the data exchanged between the parties, which an authenticated key exchange protects against. In addition, while OP-TEE provides internal implementation for key exchange algorithms such as Diffie-Hellman or Elliptic-curve Diffie–Hellman, Mbed TLS was used to ensure cross compatibility with other libraries, as it supports the generation of parameters in the format used by TLS.

Key Exchange

The key exchange algorithm used in this implementation is Elliptic-curve Diffie–Hellman as, as mentioned, elliptic curves allow for smaller key sizes, which is essential for systems with small physical memories and low computing power. When the Client Application initiates a session with the key exchange TA, the latter sets up an ECDH context in Mbed TLS and generates it's private value and the corresponding public value based on the elliptic curve that was set. The public value, which corresponds to a point in the elliptic curve, is signed using the attestation PTA:

```
TEE_Result TA_OpenSessionEntryPoint(uint32_t pt, TEE_Param params[4], void **s_id_ptr){
    ...
```

```
    mbedtls_ecdh_init(dh_ctx);
    if(mbedtls_ecdh_setup(dh_ctx, MBEDTLS_ECP_DP_SECP256R1)){
        res = TEE_ERROR_SECURITY;
        goto free_ecdh;
    }

    res = mbedtls_ecdh_make_params(dh_ctx, &olen,
                                   params[0].memref.buffer, params[0].memref.size,
                                   f_rng, NULL);
    if(res){
        res = TEE_ERROR_SECURITY;
        goto free_ecdh;
    }

    params[0].memref.size = olen;

    res = attest(params);
    if(res){
        res = TEE_ERROR_SECURITY;
        goto free_ecdh;
    }
    ...
}
```

For the verification process of the attestation signature, the Client Application can also request the device certificate by setting the third parameter type to `TEE_PARAM_TYPE_MEMREF_OUTPUT`, which will lead the TA to fetch this certificate and write it to the third parameter:

```
// Check if Caller/Client expects device certificate
if(TEE_PARAM_TYPE_GET(pt, 2) == TEE_PARAM_TYPE_MEMREF_OUTPUT){
    pt = TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_OUTPUT,
                         TEE_PARAM_TYPE_NONE,
                         TEE_PARAM_TYPE_NONE,
                         TEE_PARAM_TYPE_NONE);

    res = get_device_cert(NULL, pt, &params[2]);
    if(res){
        res = TEE_ERROR_SECURITY;
        goto free_ecdh;
```

```
    }
}
```

The ECDH context is stored in the session information structure of the TA, which will also point to the shared key once the TA receives the public value of the Client:

```
struct ke_session {
    mbedtls_ecdh_context *dh_ctx;
    TEE_ObjectHandle sh_key;
};

...

sess = TEE_Malloc(sizeof(struct ke_session), TEE_MALLOC_FILL_ZERO);
if(!sess){
    res = TEE_ERROR_OUT_OF_MEMORY;
    goto free_ecdh;
}

sess->dh_ctx = dh_ctx;
*s_id_ptr = sess;
```

On the Client side, the initial exchange up to the generation of the Client's own private and public values are implemented by a boostrap function called `bootstrap_ecdh`. The first invocation to the TA is followed by the verification of the attested output, which assumes the device certificate is stored in the `DC.der` file:

```
void bootstrap_ecdh(TEEC_Context *ctx, TEEC_Session *sess, TEEC_Operation *op,
                    TEEC_UUID *uuid, mbedtls_ecdh_context **dh_ctx,
                    mbedtls_ctr_drbg_context *rng, void *pub_k, size_t *pub_k_len){
    ...
    res = TEEC_OpenSession(ctx, sess, uuid,
                           TEEC_LOGIN_PUBLIC, NULL, op, &eo);
    ...
    res = verify_attested_output(op->params[1].tmpref.buffer, op->params[1].tmpref.size,
                                 op->params[0].tmpref.buffer, op->params[0].tmpref.size)

    if(res){
        mbedtls_strerror(res, error_buf, 1024);
        errx(1, "verify_attested_output: %x (error meaning %s)", res, error_buf);
    }
```

```
    res = create_dh_key(dh_ctx,
                        rng,
                        op->params[0].tmpref.buffer, op->params[0].tmpref.size,
                        pub_k, pub_k_len);
}
```

As part of this verification, the public key of the device is extracted from the device certificate and used to verify the attestation signature. If this procedure succeeds, the Client generates it's private and public values after initializing and ECDH context using the public value of the TA to set the parameters used in the key exchange (**i.e.** the curve and the generator):

```
static int create_dh_key(mbedtls_ecdh_context **dh_ctx,
                         mbedtls_ctr_drbg_context *rng,
                         void *dh_params_p, size_t dh_params_s,
                         void *buf, size_t *len){
    ...

    mbedtls_ecdh_init(*dh_ctx);
    res = mbedtls_ecdh_read_params(*dh_ctx, &dh_params, dh_params_end);

    ...

    res = mbedtls_ecdh_make_public(*dh_ctx, &tmp,
                                   buf, *len,
                                   mbedtls_ctr_drbg_random, rng);
    ...
}
```

To finish the key exchange the Client must invoke the `TA_KE_INIT` command, and send its signed public ECDH key along with its Client Certificate as parameters, ensuring mutual authentication of all the parties involved. If the invocation is successful, the Client computes the shared key and makes use of its SHA-256 hash as the key for the symmetric cipher that will be used in encrypting the data exchanged between the parties:

```
static int compute_shared_key(mbedtls_ecdh_context *dh_ctx,
                              mbedtls_cipher_context_t **enc_ctx,
                              mbedtls_cipher_context_t **dec_ctx,
                              mbedtls_ctr_drbg_context *rng){
```

```
    int res = 0;
    const mbedtls_md_info_t *md;
    uint8_t tmp[32], sh_key[32];
    size_t len = 32;

    res = mbedtls_ecdh_calc_secret(dh_ctx, &len,
                                   tmp, len,
                                   mbedtls_ctr_drbg_random, rng);

    md = mbedtls_md_info_from_type(MBEDTLS_MD_SHA256);
    res = mbedtls_md(md, tmp, len, sh_key);

    ...

    mbedtls_cipher_init(*enc_ctx);
    res = mbedtls_cipher_setup(*enc_ctx,
                               mbedtls_cipher_info_from_type(MBEDTLS_CIPHER_AES_256_GCM)

    ...

    res = mbedtls_cipher_setkey(*enc_ctx, sh_key, 256, MBEDTLS_ENCRYPT);

    ...

}
```

When the TA receives the `TA_KE_INIT` command, it verifies the identity of the Client using the public key certificate it received, which must be issued by the CA whose public key certificate is stored in the TA's main memory. If this verification succeeds, the public key stored in the certificate is used to validate the signature of the ECDH key the client sent:

```
static TEE_Result verify_cli_params(TEE_Param params[4]){
    TEE_Result res = TEE_SUCCESS;
    mbedtls_x509_crt ca_crt;
    mbedtls_x509_crt client_crt;
    void *buf = TEE_Malloc(64, TEE_MALLOC_FILL_ZERO);
    uint32_t flags;
```

```
    ...

    res = mbedtls_x509_crt_parse_der(&ca_crt, ca_cert, sizeof(ca_cert));

    ...

    res = mbedtls_x509_crt_parse(&client_crt, params[2].memref.buffer, params[2].memref.

    ...

    res = mbedtls_x509_crt_verify(&client_crt, &ca_crt, NULL, NULL, &flags, NULL, NULL);

    ...

    res = mbedtls_pk_verify(&client_crt.pk, MBEDTLS_MD_SHA256,
                            buf, 64,
                            params[0].memref.buffer, params[0].memref.size);
   ...
}
```

This procedure, when successful, ensures mutual authentication between the Client and the Secure World, preventing adversaries from accessing services in the Secure World. The generation of the shared key is executed after reading the Client's public value, and its SHA-256 hash is converted to a transient object which OP-TEE uses to store cryptographic keys:

```
static TEE_Result init_ke(void *s_ptr, uint32_t pt, TEE_Param params[4]){
    TEE_Result res = TEE_SUCCESS;
    TEE_OperationHandle md = TEE_HANDLE_NULL;
    TEE_Attribute attr[1];
    struct ke_session *sess = (struct ke_session*)s_ptr;
    void *tmp = NULL , *sh_key = NULL;
    size_t tmp_l;
    uint32_t sh_key_l, e_pt = TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_INPUT, // Signature
                                              TEE_PARAM_TYPE_MEMREF_INPUT, // DH Paramet
                                              TEE_PARAM_TYPE_MEMREF_INPUT, // Certificat
                                              TEE_PARAM_TYPE_NONE);

    ...

    res = verify_cli_params(params);
```

```
    ...

    res = mbedtls_ecdh_read_public(sess->dh_ctx, params[1].memref.buffer,
                                   params[1].memref.size);

    ...

    res = mbedtls_ecdh_calc_secret(sess->dh_ctx, &tmp_l,
                                   tmp, tmp_l,
                                   f_rng, NULL);

    // Convert shared secret to AES key
    res = TEE_AllocateOperation(&md, TEE_ALG_SHA256, TEE_MODE_DIGEST, 0);

    ...

    sh_key = TEE_Malloc(32*sizeof(uint8_t), TEE_MALLOC_FILL_ZERO);

    ...

    res = TEE_DigestDoFinal(md, tmp, tmp_l, sh_key, &sh_key_l);

    ...

    res = TEE_AllocateTransientObject(TEE_TYPE_AES, 256, &sess->sh_key);

    ...

    TEE_InitRefAttribute(attr, TEE_ATTR_SECRET_VALUE, sh_key, sh_key_l);
    res = TEE_PopulateTransientObject(sess->sh_key, attr, 1);
    ...
}
```

In contrast to the key exchange protocol, in OP-TEE the encryption and decryption algorithms used in secure communication were implemented using the Cryptographic Operations API provided by the TEE Internal Core API. The advantage of this solution is the ability for hardware manufacturers to replace the cryptographic library used by OP-TEE with one that supports hardware acceleration, considerably

increasing the speed of cryptographic operations such as AES encryption, which is paramount in real-time communication.

Secure TA invocation

The implementation of an attested key exchange protocol to establish a secure communication channel that provides confidentiality and authenticity guarantees has several interesting applications in the context of OP-TEE. Some of these include ability to send and receive data to and from Trusted Applications without leaking the plain text to potential adversaries while this data is in a shared memory region or the ability to remotely install/update Trusted Applications given the Client has a valid private key. Considering the former case, the use of authenticated encryption to ensure the confidentiality and authenticity of the data exchanged between the Client and a Trusted Application addresses a real problem, whereby an adversary in the Normal World could read and modify the data stored in the shared memory region used to pass information between the Secure and Normal World. After performing the key exchange protocol the Client can issue the `TA_KE_CMD_TA` command to perform a secure invocation of a target Trusted Application installed on the device. At a high level, the key exchange TA acts as a proxy for the communication between the Client and the target TA, decrypting and encrypting data. Upon receiving the `TA_KE_CMD_TA` command, the proxy TA expects the target TA's UUID as the first parameter, followed by the command ID in the second paramater along with the input data in the third parameter, which will also be used for the output data of the target TA:

```
static TEE_Result cmd_ta(void *s_ptr, uint32_t pt, TEE_Param params[4]){
    uint32_t eo, e_pt = TEE_PARAM_TYPES(TEE_PARAM_TYPE_MEMREF_INPUT, //TA UUID
                                        TEE_PARAM_TYPE_VALUE_INPUT,  //Command ID
                                        TEE_PARAM_TYPE_MEMREF_INOUT, //Data
                                        TEE_PARAM_TYPE_NONE);
    ...

    tee_uuid_from_str(&ta_uuid, params[0].memref.buffer);

    ...

    res = decrypt_payload(s_ke,
                          params[2].memref.buffer, params[2].memref.size,
                          ta_params[0].memref.buffer, &ta_params[0].memref.size);
    ...

    res = TEE_OpenTASession(&ta_uuid, TEE_TIMEOUT_INFINITE, 0, NULL, &sess, &eo);
```

```
    if(res)
        return res;
    res = TEE_InvokeTACommand(sess,
                              TEE_TIMEOUT_INFINITE,
                              params[1].value.a,
                              pt, ta_params,
                              &eo);
    TEE_CloseTASession(sess);

    ...

    res = encrypt_payload(s_ke,
                          ta_params[0].memref.buffer, ta_params[0].memref.size,
                          params[2].memref.buffer, &params[2].memref.size);
    ...
}
```

Before invoking the target TA, the proxy TA decrypts the data stored in the input buffer using the shared secret established in the key exchange protocol. The decryption procedure expects the input buffer to be in the following format: **tag || IV || encrypted plain text** and performs the decryption along with the message authentication:

```
static TEE_Result decrypt_payload(struct ke_session *sess,
                                  void *buf, uint32_t len,
                                  void *ptxt, uint32_t *ptxt_l){
    ...
    res = TEE_AEInit(dec_op, (uint8_t *)buf + 16, 12, 16*8, 0, len - 16 - 12);
    if(res)
        goto out;

    res = TEE_AEDecryptFinal(dec_op,
                             (uint8_t *)buf + 16 + 12, len - 16 - 12,
                             ptxt, ptxt_l,
                             buf, 16);
    ...
}
```

If the tag matches the expected value, the data is decrypted and sent to the target TA. One key detail when decrypting this data is the location of the destination buffer, which must be a region in secure memory,

as placing the plain text data in shared memory would defeat the purpose of encrypting it due to the lack of access control in this region. As such, before decrypting the data, the proxy TA allocates a buffer in secure memory the size of the input data (including the tag and IV) and resizes the buffer after the decryption operation to save memory:

```
ta_params[0].memref.size = params[2].memref.size;
ta_params[0].memref.buffer = TEE_Malloc(params[2].memref.size, TEE_MALLOC_FILL_ZERO);
if(!ta_params[0].memref.buffer)
    return TEE_ERROR_OUT_OF_MEMORY;

res = decrypt_payload(s_ke,
                      params[2].memref.buffer, params[2].memref.size,
                      ta_params[0].memref.buffer, &ta_params[0].memref.size);
if(res)
    return res;

if(ta_params[0].memref.size < params[2].memref.size){
    tmp = TEE_Realloc(ta_params[0].memref.buffer, ta_params[0].memref.size);
    if(!tmp){
        TEE_Free(ta_params[0].memref.buffer);
        res = TEE_ERROR_OUT_OF_MEMORY;
    }else ta_params[0].memref.buffer = tmp;
}
```

After executing the requested command, the returned data is encrypted using the shared secret as input for AES in GCM mode, with the output format being the same as the expect input format. On the Client side, when the user wishes to securely invoke a TA running in the Secure World, it must provide the UUID of the target TA along with the command ID and the input data, if any:

```
op.paramTypes = TEEC_PARAM_TYPES(TEEC_MEMREF_TEMP_INPUT, //TA UUID
                                 TEEC_VALUE_INPUT,       //Command ID
                                 TEEC_MEMREF_TEMP_INOUT, //Data
                                 TEEC_NONE);

...

if(!encrypt_payload(enc_ctx, &rng,
                    &op.params[2].tmpref.buffer, &op.params[2].tmpref.size)){
    res = TEEC_InvokeCommand(&sess, TA_KE_CMD_TA, &op, &eo);
```

```
    if(res) errx(1, "TEEC_InvokeCommand %#" PRIx32 " (error origin %#" PRIx32 ")",
                res, eo);
    else
        if(!decrypt_payload(dec_ctx,
                            &op.params[2].tmpref.buffer, &op.params[2].tmpref.size))
            printf("Command run successfully!\n");
        else printf("Invalid ciphertext payload!\n");
}
```

As mentioned, the data is encrypted using AES-GCM, which provides data authenticity and confidentiality. The choice for this combination of cipher and block cipher mode was motivated by the performance characteristics of this mode, as it allows for the encryption and decryption of data to be executed in parallel on the different blocks. Other modes such as CCM also provide data authentication guarantees, but require the length of the message to be known in advance (for computing it's tag using CBC) and present lower performance due to the sequential nature of CBC. The use of a stream cipher such as ChaCha20, in combination with a MAC such as Poly1305, was considered due to the high performance nature of stream ciphers and the existence of high performance software implementations [9] of these two primitives but the lack of support for these in OP-TEE prevented their use.

Most block cipher modes of operation require the use of a Nonce or IV to prevent identical blocks of plaintext to be encrypted into the same ciphertext when the same key is in use. As such, the IV/Nonce generation recommendation is a key phase in the encryption procedure. Following NIST's recommendations for block cipher modes GCM and GMAC [12] with regards to Nonce/IV construction, there are two frameworks for constructing IVs based on their length:

- **Deterministic Construction**: the IV can be considered as a two part vector, with the first part being a fixed value (associated with the device or the current instance of the encryption function) and the second, called the invocation field, a value "associated" with the input, which mustn't repeat itself for the same key. To ensure this, the invocation field can be implemented using a counter that is incremented between invocations or a linear feedback shift register.

- **RBG-based construction**: in this framework, the IV is also divided in two different fields, a random field and a free field. The length of each is fixed for the duration of the key, with the random field requiring at least 96 bits while the free field may be empty. The random field can be the direct result of a Random Bit Generator, called a directo random string, or the combination of this output with the preceding IV value. The free field is recommended to be empty to ensure the whole IV is random.

Due to the state keeping requirement of the first framework and the since the length of IV used is larger than 96 bits, the second framework was used, setting the IV to the direct result of the RBG used:

```
static int encrypt_payload(mbedtls_cipher_context_t *enc_ctx,
                           mbedtls_ctr_drbg_context *rng,
                           void **buf, size_t *len){
    int res = 0;
    uint8_t *ctxt = NULL;
    size_t iv_len = mbedtls_cipher_get_iv_size(enc_ctx), ctxt_len;

    ...

    res = mbedtls_ctr_drbg_random(rng, ctxt + 16, iv_len);
    if(res)
        goto out;

    res = mbedtls_cipher_auth_encrypt(enc_ctx, ctxt + 16, iv_len, NULL, 0,
                                      *buf, *len,
                                      ctxt + 16 + iv_len, &ctxt_len, ctxt, 16);
    ...
}
```

resorting to Mbed TLS's Random Number Generator module, which provides a Random Bit Generator as specified in NIST's "Recommendations for random number generation using deterministic random bit generators" [5]. This generator is instatiated in the initial key exchange and is reseeded automatically, providing a simple API for fetching a cryptographically secure random bit sequence when necessary.

4.3 SUMMARY

The development of an attestation mechanism that is anchored in the secure boot process of a TrustZone enabled device allows outside users to attest that a result was securely generated/computed inside a Trusted Execution Environment. However, several critical factors required consideration when designing an approach that combined flexibility, security and performance, as all of these are paramount in critical systems. And while the attestation mechanism itself is simple, this factors greatly increased the complexity of the development. Additionally, the implementation of such mechanism in ARM TrustZone allowed for other schemes, such as a key exchange, to harness it for authentication purposes, benefiting from the security properties of TrustZone to provide security guarantees to other parties.

On the other hand, by default OP-TEE restricts the amount of code that can be executed inside the TEE. While this is motivated by the need to reduce the attack surface and prevent adversaries from running arbitrary code in the Secure World, it results in a limited scope of legitimate applications that can be executed in the TEE. The support for third party TAs signed with valid keys addresses this problem by increasing the

number of keys that can be used to sign Trusted Applications while requiring that these keys be signed by the "master" key. This results in a middle ground between the default OP-TEE behaviour and the one provided by Intel SGX.

The development of these three modules required the modification of three major open source projects related to OP-TEE and ARM TrustZone, as a result the projects were forked and the modified versions can be found in the following repositories:

- OP-TEE OS: https://github.com/MQuaresma/optee_os
- ARM Trusted Firmware: https://github.com/MQuaresma/arm-trusted-firmware/
- OPTEE Client: https://github.com/MQuaresma/optee_client

5

CASE STUDIES / EXPERIMENTS

The applicability of the system that has resulted of this work has been mentioned several times throughout this thesis. However, many of the use cases listed were abstract and their connection to real world problems where embedded systems are involved wasn't made clear. On the same note, the description of the modules that were developed left out key details regarding the configuration/installation of these modules in real hardware and the requirements and challenges that such process might entail. These chapter will focus on describing the setup process of OP-TEE, ARM Trusted Firmware-A and the components developed as part of this work on the Pine A64+ development board, which is equiped with a TrustZone enabled ARM processor. In addition, three main use cases will be presented, taking into account not only the problems addressed but also how the system developed is harnessed by each use case, to achieve its goal.

5.1 EXPERIMENT SETUP

Although emulation software such as QEMU is able to provide full system emulation for ARMv8 SoCs, the development of firmware and operating system level functionality for embedded systems benefits from the use of "real hardware" as this makes it easier to identify potential bottlenecks introduced by the resource constraint nature of these devices. To overcome this barrier, a development board equiped with an ARM processor is key, although factors such as community support for the major firmware and software components as well as the ISA version of the SoC ought to be considered when researching which hardware to acquire. The device used as a test bench for the present work was the Pine A64+ board running an A64 SoC.

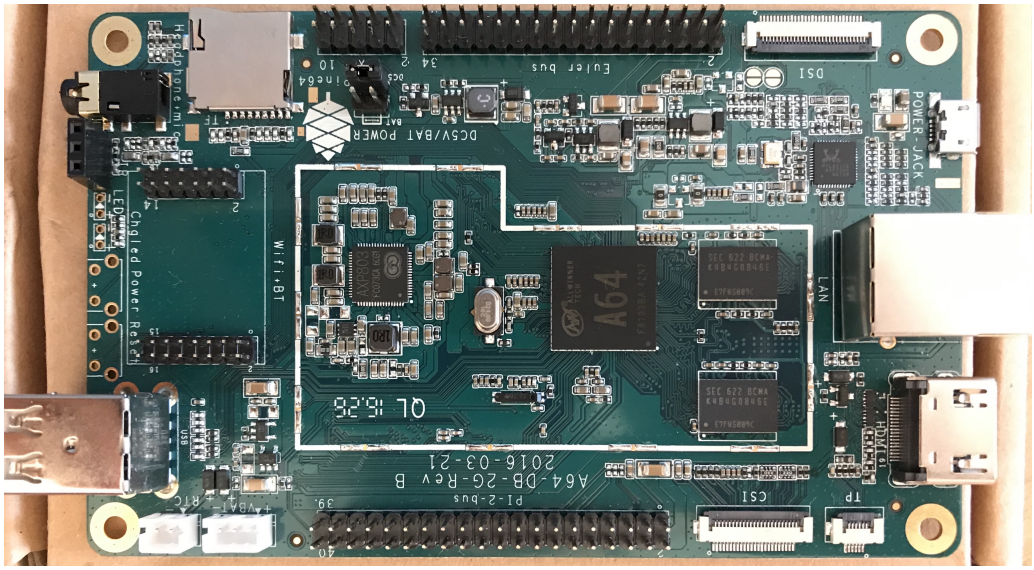


Figure 13: Pine A64+

This SoC is made up of a Quad-core Cortex A53 ARM CPU which implements the AArch64 ISA, thus supporting ARM Trusted Firmware-A. This support was one of the main requirements when researching development boards as it ensures that, barring hardware specific limitations, the modifications introduced in ARM Trusted Firmware to support features such as the attestation mechanism are cross compatible between different ARMv8 SoCs.

As a complement to this setup, and to streamline the development and testing/debugging process, OP-TEE's QEMU configuration was used for the bulk of the development process. Unlike the Pine A64+ setup, this QEMU one facilitates the attachment of a debugger to a running instance of OP-TEE, while the development board requires the use of a UART cable to establish a serial console from which to debug the system.

5.1.1 *Board Setup*

It's common practice for board manufacturers to provide a Board Support Package (BSP) with proprietary drivers and patches for the firmware/software components needed to get the system working. However, the closed source nature of some of the components makes debugging and patching, needed to add new features, harder. Therefore, if the open source support for the board is "mature" enough, it's preferrable to compile all the required components from their mainline repositories, using only open source code bases. As expected, the components that are required depend on the target (software) setup, which in this work involves using ARM Trusted Firmware for the firmware layer, OP-TEE as the Trusted OS and Linux as the Normal World OS (although other Normal World OSs can be used, as will be made evident in one of the use cases). Besides these main components, for a system to be functional it requires a bootloader that

will invoke the firmware layer when the system is powered on. U-Boot is a common bootloader of choice for use in embedded systems and, due to it's popularity and well documented code base, it was chosen as the bootloader for this setup. It's worth noting that, although the mainline versions of each of these components provide support for the Pine A64+, this support is limited to a headless setup and, at the time of this work, there is no existing configuration for running OP-TEE and Linux in parallel. Due to this limitation, some patches had to be applied to both the Linux kernel and U-Boot to add support for running OP-TEE and Linux in parallel in the Pine A64+. Before describing the patches and the remainder of the configuration in more detail, it's paramount to be acquainted with the bootflow of ARMv8 systems running a Linux + OP-TEE configuration, with ARM Trusted Firmware executing at EL3:

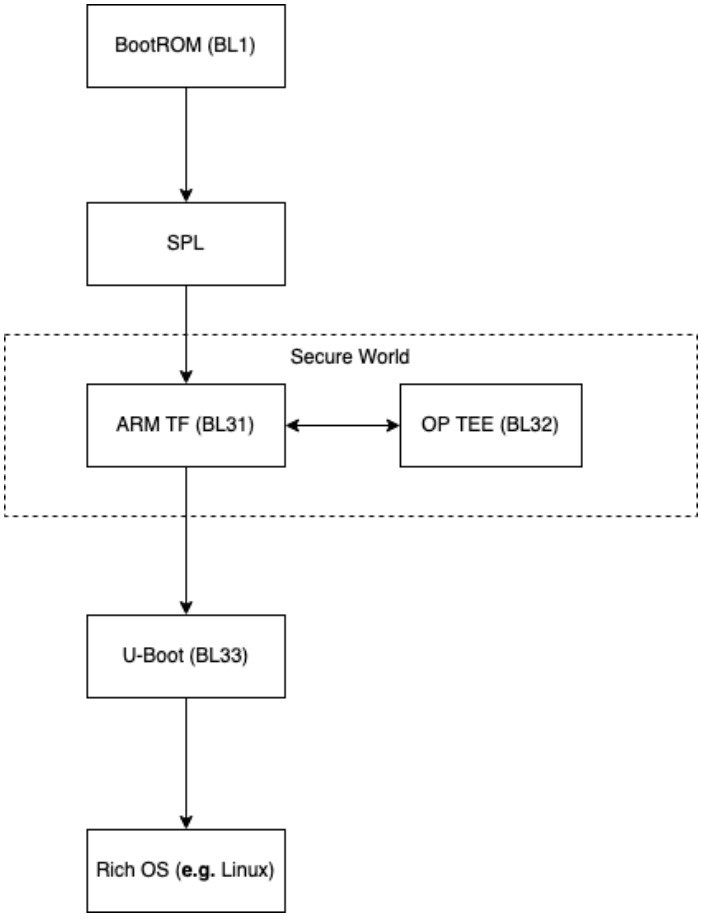


Figure 14: ARMv8 Boot flow

This process, although similar across ARMv8 SoCs, may differ when it comes to the components that implement each of the boot stages and the order by which the Secure OS and the Normal World bootloader are called, which directly affects the Chain of Trust described in 2.4. In the standard secure boot the BootROM, called BL1 in TBB, is the first code to be executed when the device is powered on and is responsible for loading the second stage of the bootloader, BL2, to SRAM. In some systems, the BootROM

is implemented by the Trusted Firmware component, which makes it easier to verify the full Chain of Trust as no proprietary code, which may contain unpatchable vulnerabilities or backdoors, is executed however, due to it's immutability, BL1 is trusted by default. A consideration that must be taken into account when loading BL2 to SRAM is the reduced size of this memory, which usually spans only a few Kilobytes (64Kb in the Pine A64+). In these cases BL2 is only responsible for setting up SRAM/DRAM and loading a full bootloader to memory which will take care of the remaining setup operations. In the Pine A64+ the BL2 stage is implemented by a component named Secondary Program Loader (SPL), which fits in SRAM and loads ATF-A, U-Boot (the Normal World bootloader) and OP-TEE into DRAM before branching to ATF-A. ARM Trusted Firmware-A is responsible for the BL31 stage of the boot process as well as providing the functionality that is expected of the EL3 firmware layer **i.e.** the Secure Monitor. During the boot process it uses a Secure-EL1 Payload Dispatcher (SPD) to initialize the (secure) CPU context for the Secure World, branching to the address that OP-TEE was loaded to by the SPL, which ensures that no untrusted code is executed before OP-TEE is initialized. Once it is done booting, OP-TEE returns to ATF-A, running in EL3, by issuing an SMC, which will save the current S-EL1 CPU context and hand execution over to BL33 **i.e.** U-Boot. It's this last component that will boot the Normal World OS.

In the Pine A64+, each of the boot stages are of the responsibility of the following components:

- **BL1**: on-chip Boot-ROM (BROM)
- **BL2**: Secondary Program Loader, which in this case is part of U-Boot
- **BL31**: EL3 runtime firmware, implemented by ARM Trusted Firmware-A
- **BL32**: Trusted OS, implemented by OP-TEE
- **BL33**: U-Boot bootloader

Patches

Even though mainline support for the Pine A64+ is present for each of these components individually, their combined setup requires some modifications to the base configurations provided. In particular, the default configurations of U-Boot and the Linux kernel for the Pine A64+ only work for setups where OP-TEE isn't running on the device and where the ARM Trusted Firmware-A component executes after U-Boot. The first patch was applied to U-Boot, more specifically to the script used to create the firmware image containing SPL + ATF-A + U-Boot, that is flashed onto the SD card from which the board boots. The default script and configuration create an image without OP-TEE and set U-Boot as the first image to be loaded by the Secondary Program Loader, which breaks the Chain of Trust by loading a untrusted code before all the Trusted OS has executed. To prevent this from happening, the script was modified to include an image of OP-TEE, compiled previously for the Pine A64+, load it to the 0x40000000 physical memory address and

hand control to ARM Trusted Firmware-A instead of U-Boot. ATF-A will then invoke the opteed Secure-EL1 Payload Dispatcher which expects OP-TEE to be stored at address 0x40000000 in physical memory. Once OP-TEE and ATF-A have finished initializing, U-Boot is invoked using the address passed by the SPL. The second patch was applied to the Pine A64+ Device Tree (DT), which is a file used by the Linux kernel to configure embedded systems at boot. This patch ensured that the Linux OP-TEE driver is loaded at boot to the correct address, so that the Normal World can communicate with the Secure World.

Assuming both of these patches have been applied the setup process can be summed up the following steps [1]:

1. Compile ARM Trusted Firmware-A for the Pine A64+ with the opteed SPD enabled:

```
$ make PLAT=sun50i_a64 SPD=opteed bl31
$ BL31=$(pwd)/build/sun50i_a64/release/bl31.bin
```

2. Compile OP-TEE for the Pine A64+

```
$ make CFG_ARM64_CORE=y \
       CFG_TEE_LOGLEVEL=4 \
       CFG_TEE_CORE_LOG_LEVEL=4 \
       CROSS_COMPILE32="ccache arm-linux-gnueabihf-" \
       CROSS_COMPILE64="ccache aarch64-linux-gnu-" \
       PLATFORM=sunxi-sun50i_a64
$ export TEE=$(pwd)/out/arm-plat-sunxi/core/tee-pager_v2.bin
```

3. Compile the OP-TEE Client

```
$ make
$ cd out/export
$ tar -cfv optee_client.tar.gz usr
```

4. Compile U-Boot, which will generate a full firmware image named **u-boot-sunxi-with-spl.bin**

```
$ make pine64_plus_defconfig
$ make
```

5. Compile Linux with the patched Device Tree

The full firmware image should be flashed to the SD card with an offset of 8Kb, to preserve the partition table, and the Linux kernel and DT binary should be placed in the first partition, along with the U-Boot configuration file.

1 The following blog post Setting up OPTEE and Linux for the Pine A64, which is a result of this work, can be used as an in-depth guide for setting up OP-TEE + Linux on Pine A64+

Figure 15: Board setup

5.1.2 *QEMU Setup*

Although the complexity of setting up QEMU to create a development environment for OP-TEE is low, some nuances were introduced in this process due to the modification of core components such as OP-TEE itself, ARM Trusted Firmware and the OP-TEE client. By default, OP-TEE provides a set of configurations that can be used to compile a full OP-TEE developer setup (**i.e.** including the Normal World bootloader, kernel, etc) for a target platform. This setup resorts to several (git) repositories that are managed using repo, a tool developed by Google to manage several git repositories from different sources using an XML file, called a manifest, specific to each target platform. The one used for QEMUv8 (**i.e.** the one that emulates ARMv8) uses the "original" versions of OP-TEE, ARM Trusted Firmware and OP-TEE Client:

```
<project path="optee_client"          name="OP-TEE/optee_client.git" />
<project path="optee_os"              name="OP-TEE/optee_os.git" />
...
<project path="trusted-firmware-a"    name="TF-A/trusted-firmware-a.git"
                                      revision="refs/tags/v2.2"
                                      clone-depth="1" remote="tfo" />
```

therefore a new manifest was created that replaced these versions with the ones that include the introduced modifications:

```
<project path="optee_client"          name="MQuaresma/optee_client.git"
```

```
                                    revision="master" clone-depth="1" />
<project path="optee_os"            name="MQuaresma/optee_os.git"
                                    revision="master" clone-depth="1" />
...
<project path="trusted-firmware-a"  name="MQuaresma/arm-trusted-firmware.git"
                                    revision="v2.2_head" clone-depth="1"/>
```

Using the introduced modifications, the following sequence of commands can be executed to clone, build and run an emulated version of OP-TEE + Linux on QEMUv8:

```
$ repo init -u https://github.com/haslab/Reassure.git
$ repo sync --no-clone-bundle
$ cd build
$ make toolchains
$ make CFG_DEVICE_ATTESTATION=y CFG_THIRD_PARTY_TA=y
$ make QEMU_VIRTFS_ENABLE=y QEMU_VIRTFS_HOST_DIR=/home/optee/bin run-only
```

The `QEMU_VIRTFS` flags allow QEMU to share a directory with the host, which facilitates the installation of Trusted Applications for testing purposes. After executing the last command, two serial terminals will launch, one corresponding to the Secure World output and the other one can be used to interact with the Normal World:

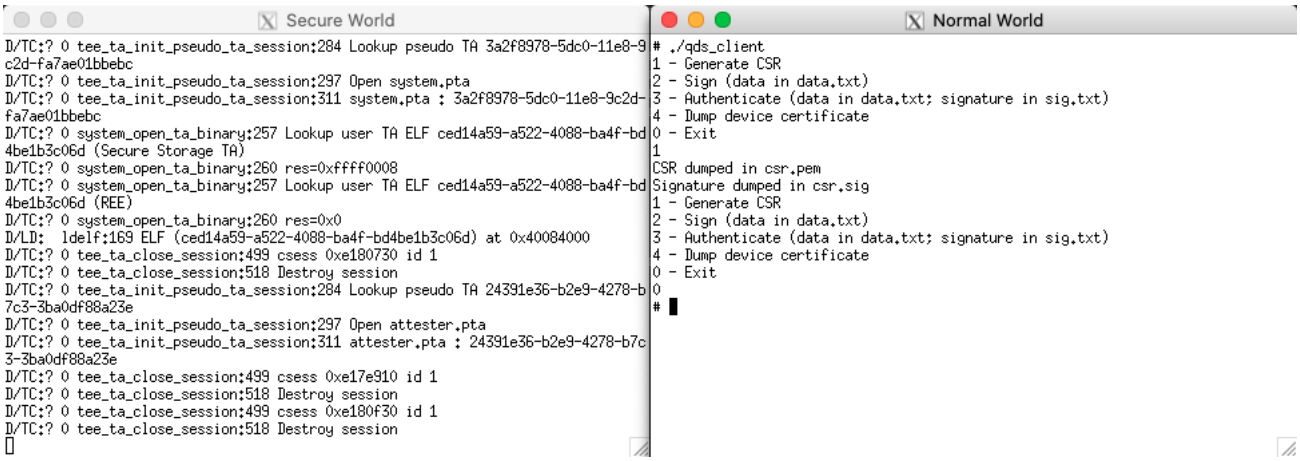


Figure 16: QEMU setup

5.2 USE CASES

The importance of attested computation schemes and trusted computing in general has been mentioned several times in this thesis. However, the results of this work in particular can be expressed in four distinct

use cases that are specific to the context of embedded, and in some cases critical, systems with high assurance guarantees. Therefore, the following sections will be dedicated to describing each of these in detail and how the attestation mechanism can be used to improve or replace some of the functionality involved.

5.2.1 *Trusted Peripherals*

The case for trusted sensors has been made several times in the past [50, 41, 11], and with the rise of applications that resorting to these devices for several purposes, ranging from authentication via biometric data (**e.g.** fingerprint scanner or IR cameras for face scanning) to the monitoring of insulin levels [34] or, as proposed in [22], to selectively grant access to a filesystem drive based on physical location **i.e.** GPS readings, the need for sensors that produce values with a high degree of authenticity is paramount. The prevalence of these sensors/peripherals in systems such as mobile phones or embedded critical systems such as autonomous driving ones, commonly equipped with TrustZone enabled ARM processors, makes the attestation mechanism developed an ideal framework for providing the degree of confidence required out of this peripherals. The use of the word peripherals in this context is intentional, as it includes not only sensors aimed at monitoring physical phenomena but also components such as CAN buses [44] used in vehicle systems, which play an integral role in transporting data between microcontrollers. The problem with "traditional" sensors **i.e.** with no mechanisms that ensure the authenticity of the readings they produce, comes from the inability of the readers to attest that the values weren't modified by a malicious actor. This becomes even more prevalent in a context where readings can be done remotely, allowing adversaries installed locally to modify data before it even leaves the device. To address this using ARM TrustZone with an attested mechanism embedded in OP-TEE, we propose a two step based approach, that relies on the hardware isolation capability of TrustZone as an access control mechanism to peripherals and on the attestation mechanism to cryptographically sign the data produced by this peripherals.

Considering a GPS sensor, ARM TrustZone can configure this peripheral, via the TrustZone Protection Controller, as secure, preventing the Normal World from being able to access it or map memory regions allocated to this peripheral. This isolation is enforced by the TrustZone Protection Controller and the Memory Management Unit, which prevents accesses to regions allocated to the Secure World when the NS bit is set. This isolation ensures that the attestation signature is performed on the original data, which no untrusted code has had access to. In addition, by attesting the sensor data, remote parties can verify it's authenticity and, similarly to the scenario presented in [22], due to the attestation key being bound to a particular device, remote parties can also establish an association between sensory data and its source. One of the problems introduced by attesting raw sensor data, also described in [22] comes from the fact that legitimate users might need to modify this data before submitting to a remote party. By modifying the data after it has been signed, the attestation is no longer valid for this modified data, and the remote party attesting its

authenticity has no way of distinguish between modifications introduced by a legitimate user or intentional data modification by an attacker. The solution proposed in the article was to execute the code tasked with performing this modifications in an isolated environment where its integrity was protected, and produce a signature over the code and the modified data. Remote parties could then use this information and decide whether to trust the code and the modifications performed by it. As an alternative to this approach, the third party TA mechanism developed could be used, whereby a minimal set of functionality required for common modifications to sensor data would be ported to OP-TEE in the form of Trusted Applications signed by a trusted third party. In this case, the attestation signature would include the UUID and version of the TA that had performed the modifications, which the remote party could use to decide whether or not to trust the data. Predictably, this approach presents both advantages and drawbacks. The main advantage being the strong isolation and confidentiality guarantees attained by running the code inside TrustZone, and the ability to uniquely identify the application that had modified this data and tie it to the device via its UUID and the device certificate. On the other hand, some of this code could introduce unwanted security bugs, however the isolation between Trusted Applications and the low privilege level at which they execute would limit the amount of harm that could be done.

In a similar way, a CAN bus could also be protected from untrusted code using a combination of the mechanisms provided by ARM TrustZone, namely configuring it as secure so it would only be directly accessible from the Secure World. Due to its central role in the communication of the different components in vehicle systems, allowing different microcontrollers to exchange data, and the lack of native security measures, by controlling access to this bus using TrustZone it would be possible to sign (attest) the data before transmitting it. This, in conjunction with the ability to bind an attestation signature to its device would allow the different systems involved to verify the source of the data they were collecting and act accordingly. The lack of support for native encryption and the fact that the data frames are broadcasted to all devices, even untrusted ones, makes the use of the attested key exchange protocol to secure the communications made via the CAN bus another clear use case for the system developed. The increasing presence of embedded systems performing critical functions that resort to this bus for communication in vehicle systems further reinforces the importance for security to be implemented even at this (low) level. However, even though it's out of the scope of this thesis, topics such as the overhead introduced by the signature and encryption operations must be considered due to the real-time nature of most of these systems.

While this mechanism doesn't protect against simulated sensors readings, whereby an attacker could alter or influence the physical events being monitored, it ensures that even if said attacker is able to compromise software in the Normal World, modifications to the readings returned from the Secure World are detected and can allow for corrective measures to be taken.

5.2.2 *IoT Aggregation*

Another scenario that would benefit from both the attestation and the attested key exchange mechanisms pertains to the data aggregation performed on the information gathered by IoT sensors. This devices and the aggregation of data have applicability in a broad range of contexts, from smart cities where they're used for traffic management, air quality monitoring or smart surveillance [52, 37] to the use of the data anlytics resulting from the aggregation process to improve the quality of a service regarding smart home devices. While not all of this use cases deal with critical systems where the compromise of a system could lead to loss of human lifes, the quality of the data that is gathered may directly impact the quality of life of a population. In addition, many of these use cases raise privacy concerns [7] due to the type of data gathered, hence the confidentiality of the data as much as its authenticity ought to be considered. It's these two problems, privacy and authenticity, that can be solved using the attested key exchange and the attestation mechanism itself. Focusing on the privacy (and confidentiality) problem, the use of an attested key exchange to securely connect an IoT node gathering data to the one aggregating, called the base node, provides not only the ability to establish a secure communication with protection against passive adversaries but also, due to the attestation mechanism used in setting up the shared secret, allows each node to verify the identity of the other party. This way, a node tasked with gathering data may selectively communicate that data to trusted nodes that perform the aggregation. On the other hand, in scenarios where an attacker deploys his own nodes, the attestation of data using manufacturer issued certificates allows the aggregation node to refuse the data received from these nodes, as it won't be signed with a trusted key. A less obvious consequence of using attestation for the aggregation code for privacy, along with the isolation guarantees provided by TrustZone, is the ability for users to verify (attest) the code performing the aggregation in order to ensure that it respects the privacy of the data while having the guarantee that sensitive data processed by the aggregation node didn't leave the device and that no external party had access to it due to the confidentiality guarantees provided by the TEE.

5.2.3 *Qualified Digital Signatures*

To demonstrate the usefulness of the attestation mechanism using ARM TrustZone, one can consider the case of using TrustZone enabled devices to manage qualified digital signatures, which are a special type of digital signatures that, in addition to providing non-repudiation and authenticity assurance, are legally binding and equivalent to handwritten signatures.

The eIDAS regulation [1], defined by the EU, sets several requirements with regards to qualified digital signatures, covering topics such as the certificates for eletronic signatures, called Qualified Digital Certificates, to the characteristics of the devices used to store the key material and perform the (qualified) digital signatures, referred to as **signature creation devices**. These devices usually take the form of hardware tokens

such as smart cards, and are associated with a single service, ensuring the confidentiality and integrity of the data they hold and providing, in some cases, features such as tamper resistance. However, being bound to a single service makes them impratical for users that interact with several services that resort to qualified digital signatures for authentication purposes. Examining the requirements set by eIDAS for the characteristics these tokens should possess to be considered as signature creation devices, it's possible to argue that devices with TrustZone enabled ARM processors can meet these requirements. In addition, the popularity of these processors in devices such as smartphones, would allow the same device to be associated with different services at the same time, addressing the problem described earlier. To see how this is the case, it's useful to be acquainted with the requirements these devices should fulfill when it comes to dealing with digital signatures:

1. **confidentiality** of the eletronic signature creation data
2. ensure that the eletronic signature creation data **cannot be derived**
3. reliably **protect** the eletronic signature **against forgery**
4. ensure that the data to be signed is **not modified**
5. ensure that the management of the eletronic signature creation data is exclusive to the qualified trust service provider

Considering the isolation properties of ARM TrustZone, along with the guarantees in terms of the confidentiality and integrity of code and data inside the TEE, it's clear that TrustZone enabled devices running a GP compliant TEE provide the two fundamental properties that are necessary for signature creation devices: integrity and confidentiality. As proposed in [39], hardware-based TEEs can serve as a foundation for implementing Virtual Smart Cards (VSCs) that replace their physical counterparts (smart-cards), such as the Finnish Eletronic Identity (FINeID) [45]. Integrating a VSC architecture to deal with Qualified Digital Signatures in a TrustZone backed TEEs such as OP-TEE provides an isolated environment in which to store signature keys and perform signing and verification operations, while also ensuring proper key validation via Qualified Digital Certificates issued by a Qualified Trust Service Provider.

In addition, implementing this architecture inside a TEE with attestation services would allow Qualified Trust Service Providers to attest the correct handling of the cryptographic material (**i.e.** keys and signatures) and ensure that their integrity is not compromised while the isolation provided by TrustZone ensures that the cryptographic keys used don't leave the secure world. The following diagram depicts the architecture of a Proof-of-Concept Virtual Smart Card that resorts to OP-TEE and the attestation Pseudo TA described earlier to emulate a signature creation device that fulfills the requirements set by eIDAS:

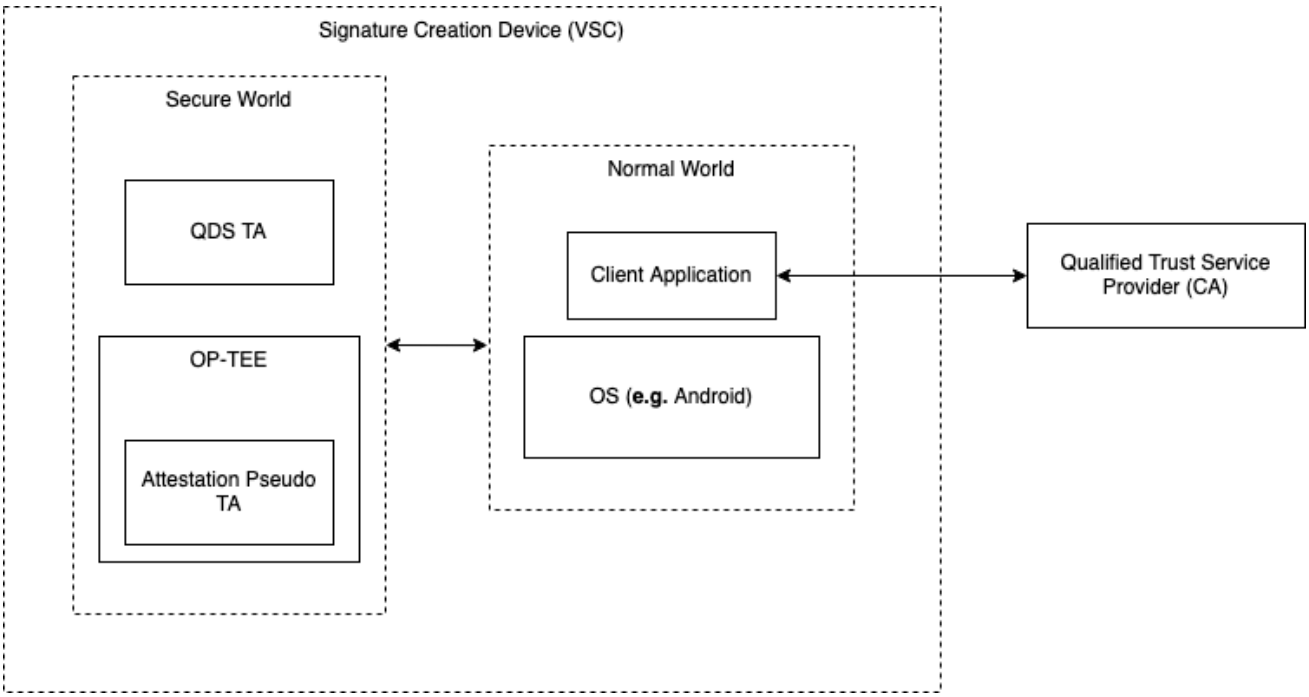


Figure 17: Proposed VSC Architecture

In this system, the QDS TA is responsible for generating and managing the private (and public) keys of the user, **i.e.** it's signature creation data, allowing the user to request the generation of a new keypair, which will trigger the creation of a Certificate Signing Request (CSR) that is signed using the attestation mechanism. Assuming the device manufacturer is trusted by the Qualified Trust Service Provider, the latter will be able to use the device certificate to verify the authenticity of the CSR, which also serves as proof that the key has been generated inside the TEE. In addition, the TA also provides commands to sign data using the generated keys.

The attestation mechanism plays a vital role in providing cryptographic proof that the keys used for performing the qualified digital signatures were generated in a secure enviroment, preventing malicious actors from emulating the behaviour of the QDS TA and generating keys on the user's behalf.

Referring back to the system architecture, the Client Application acts as the interface between the QDS TA and the "outside" world, requesting the generation of digital keys, and sending the correspoding Certificate Signing Request, via a secure channel, to the Qualified Trust Service Provider or allowing the device owner to sign data using the keys generated by the TA.

A Proof-of-Concept of this architecture was developed in an effort to provide an implementation that could be used as a reference for applications aiming to replace smart cards with Trusted Applications running in TrustZone enabled devices. The reference implementation uses the attestation mechanism to sign the Certificate Signing Request associated with the signature creation data (**i.e.** signing keys) to allow Qualified Trust Service Providers to verify it's legitimacy before issuing a Qualified Digital Certificate. The management

operations regarding the Qualified Digital Signatures and the associated keys are implemented by a Trusted Application that supports a multitude of operations commonly involved in the management lifecycle of digital identity tokens, exposed via the following interface:

```
switch(cmd_id){
    case TA_QDS_CMD_GEN_CSR:
        return gen_csr(s_ptr, pt, params);
    case TA_QDS_CMD_GET_KEY:
        return get_ecdsa_key(s_ptr, pt, params);
    case TA_QDS_CMD_SIGN:
        return sign(s_ptr, pt, params);
    case TA_QDS_CMD_VERIFY:
        return verify(s_ptr, pt, params);
    case TA_QDS_GET_DC:
        return get_device_cert(s_ptr, pt, params);
    default:
        return TEE_ERROR_NOT_SUPPORTED;
}
```

In addition to providing operations to create a Certificate Signing Request for a newly generated keypair, retreiving the public part of the keypair, and performing signing and verification operations, the TA also allows the client application to request a copy of the device certificate used by the attestation mechanism.

The `TA_QDS_CMD_GEN_CSR` command allows a user to request the generation of a key pair, along with a CSR that is signed by the attestation mechanism, providing cryptographic proof that the generation of the key material took place inside TrustZone:

```
static TEE_Result gen_csr(void *s_ptr, uint32_t pt, TEE_Param params[4]){

    ...

    if(mbedtls_x509write_csr_pem(csr_ctx,
                                 (uint8_t*)params[0].memref.buffer,
                                 params[0].memref.size, f_rng, NULL))
        return TEE_ERROR_GENERIC;

    res = attest_csr(params);
    if(res)
```

```
        return res;

    ...
}
```

Amongst other things, this ensures that the private key never leaves the Secure World, emulating the behaviour of a smart card. After verifying the signature on CSR, a Qualified Trust Service Provider can issue a Qualified Digital Certificate for the keypair. Evidently this process only works under the assumption that the device manufacturer, who issues the device certificate, can be trusted by the Qualified Trust Service Provider.

The implementation of commands to sign and verify data inside OP-TEE using previously generated keys, represents an additional step towards complying with the requirements set by the eIDAS standard, which delegates the function of performing digital signatures to the signature creation devices emulated by the TA.

This use case also benefits from the support for Third Party Trusted Applications when considering the case where a governmental agency resorts to qualified digital signatures to authenticate citizens. In this context, the developer (**i.e.** the governmental agency) has a legitimate use for TrustZone, managing credentials, and could request the device manufacturer to issue a third party certificate for its key, allowing it to sign Trusted Applications that perform tasks such as the ones described earlier.

5.2.4 *Secure DDS*

Data Distribution Service (DDS) [32, 14] is a data centric middleware standard for implementing a publish-subscribe pattern. In a publish-subscribe pattern, nodes that send data are called publishers and the data produced, instead of being addressed to specific nodes, is published according to its class. Receiver nodes, called subscribers, can show interest in a class of data by subscribing to it, automatically receiving the data publish by nodes on that category. DDS implements a topic-based publish-subscribe pattern where publisher nodes write data on a topic that is read by subscriber nodes. In addition, both types of nodes can apply filters to topics, such as time or content filters. One of the main features of DDS is the Global Data Space which provides nodes with the "illusion" that the whole domain is stored locally, with data readily accessible. However, only the data needed by each node is stored locally, with no central database. The usefullness of this abstraction stems from the fully distributed nature of it, which ensures that nodes can join and leave the domain without affecting its availability, which is paramount in critical systems. DDS also handles tasks such as data marshalling, setting Quality of Service parameters or dynamic node discovery.

Security Overview

Due to its focus on scalability, low latency data transmission and reliability, DDS is suitable for use in critical systems [42, 48], which makes it a primary target for attackers. The flexibility of DDS, where communication isn't established between two particular nodes but instead messages are multicasted to subcribers of the topic, coupled with features such as dynamic node discovery raises questions regarding the need for security measures that prevent unauthorized nodes from interacting with the system [49, 26, 25]. In an attempt to address some of these questions, the DDS Foundation devised a specification for security plugins [33] that aims to protect against the main threats that affect DDS-based systems:

1. **Unauthorized subscription**: an untrusted node on the same network infrastructure as the DDS domain may be able to inspect packets that weren't sent to it
2. **Unauthorized publication**: an unstrusted node on the same network infrastructure as the DDS domain may be able to inject packets with contents and headers of its choosing
3. **Tampering and replay**: in cases where the same shared secret is used between different nodes for encryption and MAC purposes, a node that is subscribed to a topic may be able to act as a publisher by using the shared secret to encrypt packets and compute their MAC using the shared secret
4. **Unauthorized access to data**: services, such as relay services, although trusted to perform their respective functionalities, that might involve access to metadata, should not be able to inspect a packet's contents

To address each of these threats, the specification proposes a set of Service Plugins named after the type of service they provide:

- Authentication
- AccessControl
- Cryptographic
- Logging
- Data Tagging

These plugins expose a (Service Plugin) Interface that is used by DDS to provide Information Assurance. While many embedded systems are equiped with TrustZone-enabled ARM processors, the default specification for these plugins doesn't contemplate the use of TrustZone for implementing them, which would further strengthen the security guarantees they provide.

LibDDSSec

The implementation of the security plugins while taking advantage of the properties provided by TrustZone enabled systems presents advantages over the use of "normal" OSs, mainly due to the reduced attack surface of the former coupled with the use of hardware and software mechanisms that ensure the confidentiality and integrity of code and data. Thus, harnessing TrustZone for this purpose reduces the ability for adversaries that control the Normal World to exploit the execution of the security plugins. LibDDSSec [2] is a library that implements the security services provided by the DDS Security Plugins, resorting to ARM TrustZone to implement the (security) operations involved. This way, even if a malicious agent is installed in the Normal World, the isolation properties provided by TrustZone prevent it from tampering with the execution of the security plugins.

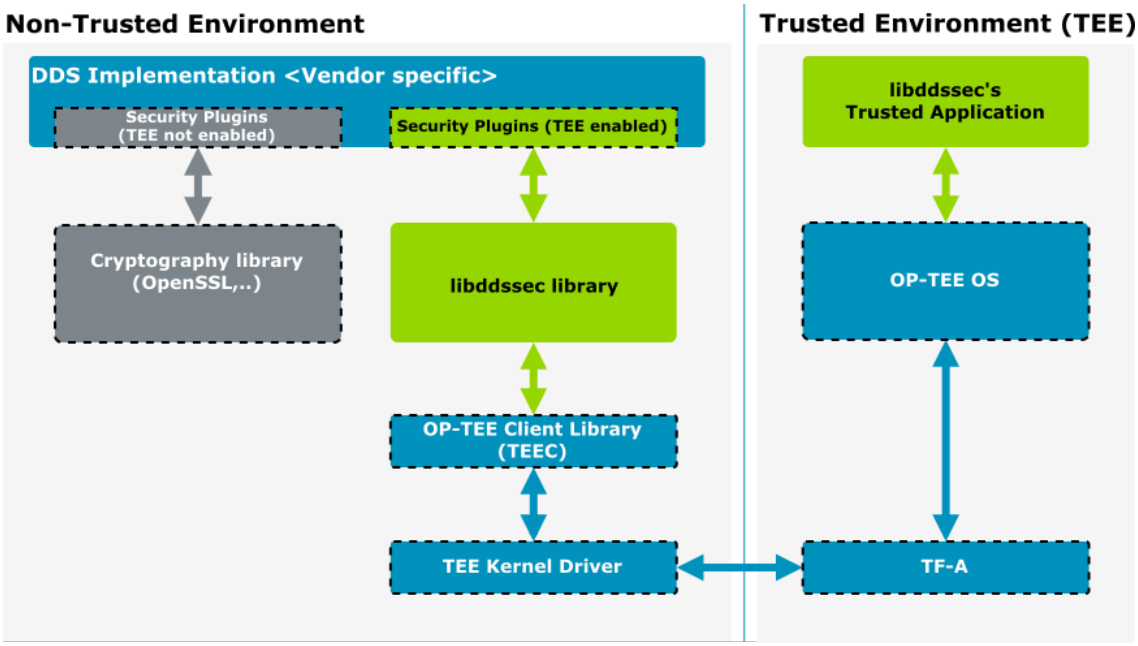


Figure 18: LibDDSSec Architecture [2]

The functionality of the "default" Security Plugins is fulfilled by two main components: a library running in the Normal World that translates the requests from the DDS implementation to commands to a Trusted Application running in the Secure World, which is the second component. The translation is carried out by the libddssec library, which acts as the Client Application and resorts to the TA for the following functionalities:

- Authentication
- AccessControl
- Cryptographic

Namely, it allows nodes to generate a CSR for a Certificate Authority to sign, which results in a X.509 certificate that nodes in the same domain, which have a copy of the CA certificate, can use to verify the identity of the node when establishing a connection. When two nodes wish to communicate, namely a subscriber and a publisher, an identity verification procedure takes place, followed by a key exchange to establish a secure communcations channel between them. The derived shared secret is used by the publisher node to encrypt the packets and calculate their MAC for authenticity purposes. If the identity verification procedure fails, the node isn't allowed to join the domain. The fact that a unique shared secret is established for each connection prevents malicious nodes from acting as publishers.

Focusing on the identity verification functionality, which resorts to certificate issued by a (trusted) Certificate Authority to authenticate two nodes when establishing a secure communication channel, it's clear how the attestation mechanism, and even the attested key exchange, could be used for this purpose. The authenticated key exchange is represented by the following sequence diagram:

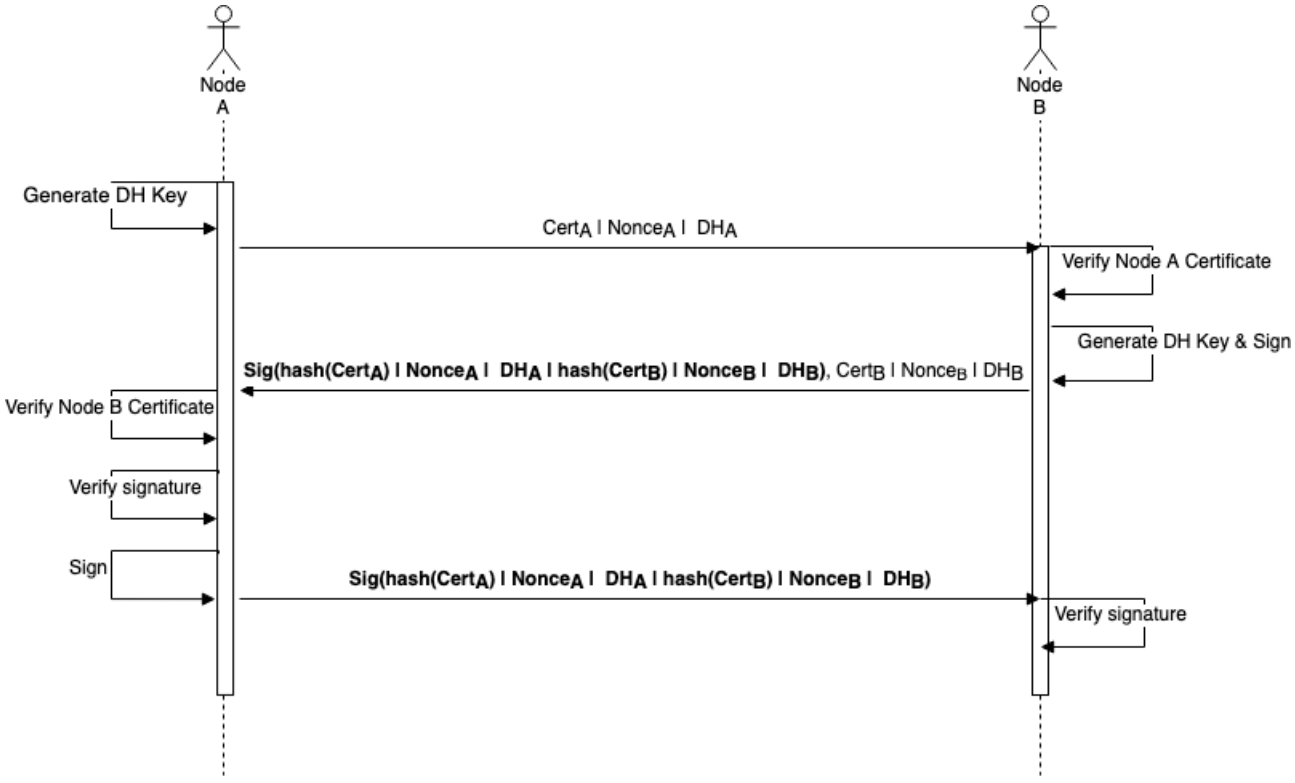


Figure 19: LibDDSSec 3-Way Handshake

Upon initiating a key exchange, the first node generates the Diffie-Hellman key, along with a nonce and sends them to the second node, along with its public key certificate, previously issued by a CA. The second node, verifies the chain for this certificate and proceeds to generate its own DH key and nonce, send them to the first node, along with a signature over these data and the one it received from the first node and its public key certificate. The first node then verifies the certificate chain of the second node and, if it succeeds,

verifies the signature of the data. Finally, the last message of the 3-Way handshake is a signature over the same data as the second node, that is sent from the first node and verified by the former. If all verifications succeed, both nodes can compute a shared secret that will be used for encrypting data using AES-GCM and authenticating it using a MAC. The use of a MAC instead of digital signatures is motivated by the fact that the former present considerably higher performance [33], which is critical in real time systems. One key advantage of the approach provided by LibDDSSec over the default one, besides the integrity of the code, is the fact that the private keys and shared secret never leave the TEE.

Attestation Mechanism & LibDDSSec

Considering the system developed as part of this work, we propose two different ways to integrate it in or even replace the LibDDSSec Trusted Application.

The first approach we propose resorts to the attestation mechanism for authentication purposes. Specifically, the signature performed as part of the authentication process in the 3-Way Handshake will correspond to the attestation of the signed data, while the certificates exchanged for verification of this signature will correspond to the device certificates that match the private key used in attestation. This way, each node will be identified by the device certificate issued by its manufacturer and therefore trusted by other nodes in the domain. If the Certificate Authority that issues the certificates for the manufacturer is common across the nodes, then the replacement of the signing process for the attestation mechanism can be done without any additionaly modifications as all devices already hold a copy of the CA certificate in Trusted Storage. The main advantage of resorting to the attestation mechanism over the "traditional" signature process for authentication purposes is the stronger guarantees it provides with regards to the environment in which hanshake was executed. Thus, by using the attestation mechanism, the nodes can not only verify the authenticity of the other node but also make sure that TrustZone was used for the handshake.

The second approach we propose follows as a natural successor of the first one and aims to replace the default LibDDSSec TA with a modified version of the attested key exchange TA (4.2.5). Similarly to the previous approach, the authentication is also achieved through the attestation mechanism and benefits from the same advantages stated earlier, with the device certificate being used for verification purposes. While the default implementation of the attested key exchange TA shares some of the features of the LibDDSSec TA, such as the use of AES-GCM for encrypting data, this replacement requires some modifications to comply with the enunciated properties. Firstly the implementation of a command to compute and verify MACs using the shared secret, which is required by the service nodes, that only have access to metadata, for authenticating packets. Furthermore, the attested data in the default attested key exchange TA only includes the DH key, therefore it would need to be modified to include the hash of the device certificate. Finally, the "client" authentication procedure would need to be modified to perform a verification of the certificate chain using the CA certificate in Trusted Storage. The main advantage of these approach, besides the stronger guarantees that come from the attestation mechanism, is the modularity of having separate components for

each of the “plugins”, which faciliates formally proving their correctness and provide the ability to upgrade each component separately.

5.3 SUMMARY

The use cases presented here are a good demonstration of the usability of the system developed, however there are several other scenarios that could make use of it. The choice of these specific use cases was mainly due to the relation with common scenarios where devices equipped with ARM processors, such as embedded devices and smartphones, are used for applications that require some degree of security and trust. As demonstraded, this requirements can be fullfiled by executing these in ARM TrustZone, and allowing remote parties to verify this execution using the attestation mechanism. Furthermore, through the deployment of the developed systems in real hardware it was shown that the requirements of embedded systems were taken into consideration when implementing these mechanisms. Amongst other things, this ensured that the design of these systems considered factors such as memory capacity and processing power of these devices, which can sometimes prevent the implementation of complex or computationally expensive algorithms.

6

CONCLUSION

The role and ubiquitousness of embedded systems across a wide range of applications makes security one of the primary concerns when designing these systems. While common security mechanisms enforced by the OS are vital in ensuring their correctness, the critical nature of data handled by these systems often asks for additional security guarantees. The use of Trusted Execution Environments to run code dealing with sensitive data in an environment isolated from the normal OS have been advocated as a valuable security mechanism. However, the trust in these environments can only be justified by the existence of a root of trust, either hardware of firmware based. In the case of TrustZone, this root takes the form of a secure boot process that ensures the integrity of every instruction executed by the SoC, from the instant it boots. Harnessing this root of trust in the context of embedded systems and the properties that rely on it, namely confidentiality and integrity of code and data, is a crucial step towards secure critical systems. We presented an attestation mechanism based on this root of trust that provides outside parties proof of secure computation, which implies that the attested data was computed inside the Secure World and therefore, its confidentiality and integrity were assured. The implementation of such mechanism has been complemented with some representative use cases discussed in 5.2, which benefit not only from using TrustZone for secure computation but also resorted to the attestation mechanism as proof of the trustworthiness of its computation.

Another contribution, aimed at increasing the flexibility of OP-TEE, dealt with adding support for Third Party Trusted Applications that are implemented by trusted developers with legitimate applications that make use of TrustZone. This feature presents a middleground between the default behaviour of OP-TEE, which restricts the code executed inside the TEE to the one signed by a "master" key, and the behaviour of Intel SGX, which gives developers the opportunity to mark blocks of code for execution inside the secure enclave.

Finally, we addressed the problem of establishing a secure communications channel between Client Applications and Trusted Applications, which by default uses shared memory for exchanging data. The proposed solution makes use of the attestation mechanism to implement an attested key exchange scheme that allows clients to establish a shared key with the Secure World which is used for encrypting the shared data, ensuring its confidentiality and integrity.

The applicability of these mechanisms was then discussed by examining four real world use cases that are common in the context of IoT and embedded (critical) systems. This examination showed clear benefits to the addition/implementation of the attestation mechanism as a way to reinforce existing security mechanisms or even to replace them completely.

6.1 FUTURE WORK

While the implemented system can be used as is and its applicability was shown in different use cases (5.2), there are several improvements that can be made not only to the system itself but also to the other components that make up the TEE.

Regarding the attestation mechanism, and as has been mentioned in its initial description, formally proving its security and correctness is paramount. The modularity of the system, divided in three main modules, facilitates proving its security under the UC framework, as already mentioned. In addition, having less complex components simplifies the process of formally proving the correctness of the whole system. The advantages of these proofs are obvious as they ensure that the implementations respect the properties expected out of the schemes.

Due to their clear applicability, another important step deals with devising a formal proposal to integrate these mechanisms in the respective projects, namely OP-TEE OS, ARM Trusted Firmware and OP-TEE Client. This proposal should state the benefits encountered in this work and provide guidelines regarding the design decisions made as part of the development process.

As for general improvements that can be made to the TEE "environment", we propose two major ones. Firstly the implementation of access control mechanisms in Trusted Applications to prevent leakage of confidential data [53], as currently, there is no restriction regarding the Client Applications that can request services to the TAs running in the Secure World. The second improvement deals with the compilation process itself, and whether it should be included in the threat model of Trusted Execution Environments. The reason for this is that compilers can be a possible attack vector for adversaries wishing to execute code on the Secure World or for vulnerabilities to be introduced as a consequence of compiler optimizations [10]. A solution to this problem can be the use of formally verified compilers [29, 21] to ensure that the executable compiled code respects the properties proved in the "original code". While this isn't a novel approach [6], it is lacking in the default development kit of OP-TEE which can lead to vulnerabilities in code that is considered critical/secure.

BIBLIOGRAPHY

[1] Regulation (eu) no 910/2014 of the european parliament and of the council of 23 july 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/ec. *OJ*, L 257:73–114, 2014-08-28.

[2] ARM. Libddssec. URL https://github.com/ARM-software/libddssec.

[3] ARM. Trusted board boot requirements client (tbbr-client) armv8-a. Technical report, ARM, 2018.

[4] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and application to SGX. *Proceedings - 2016 IEEE European Symposium on Security and Privacy, EURO S & P 2016*, pages 245–260, 2016. doi: 10.1109/EuroSP.2016.28.

[5] Elaine Barker, John Kelsey, National Institute of Standards, Technology, and U.S. Department of Commerce. *NIST Special Publication 800-90A: Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2012. ISBN 1478169311.

[6] Gilles Barthe, Pierre Courtieu, Guillaume Dufay, and Simão Melo de Sousa. Tool-assisted specification and verification of the javacard platform. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology*, pages 41–59, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45719-0.

[7] Stefano Bennati and Evangelos Pournaras. Privacy-enhancing aggregation of internet of things data via sensors grouping. *Sustainable Cities and Society*, 39:387 – 400, 2018. ISSN 2210-6707. doi: https://doi.org/10.1016/j.scs.2018.02.013. URL http://www.sciencedirect.com/science/article/pii/S2210670717310776.

[8] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. *Annual Symposium on Foundations of Computer Science - Proceedings*, 16:136–145, 2001. ISSN 02725428. doi: 10.1109/sfcs.2001.959888.

[9] F. De Santis, A. Schauer, and G. Sigl. Chacha20-poly1305 authenticated encryption for high-speed embedded iot applications. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 692–697, 2017.

[10] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *2015 IEEE Security and Privacy Workshops*, pages 73–87, 2015.

[11] Akshay Dua, Nirupama Bulusu, Wu-Chang Feng, and Wen Hu. Towards trustworthy participatory sensing. In *Proceedings of the 4th USENIX Conference on Hot Topics in Security*, HotSec'09, page 8, USA, 2009. USENIX Association.

[12] Morris J. Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. Technical report, Gaithersburg, MD, USA, 2007.

[13] Dave Evans. How the next evolution of the internet is changing everything. 2011.

[14] DDS Foundation. What is dds? URL https://www.dds-foundation.org/what-is-dds-3/.

[15] GlobalPlatform. Tee client api specification v1.0. Technical report, Global Platform, 2010.

[16] GlobalPlatform. Tee system architecture v1.2. Technical report, Global Platform, 2018.

[17] Seunghun Han and Jun-Hyeok Park. Shadow-box v2: The practical and omnipotent sandbox for arm. BlackHat Asia 2019, 2018. URL https://i.blackhat.com/briefings/asia/2018/asia-18-Seunghun-Shadow-Box_v2_The_Practical_and_Omnipotent_Sandbox_for_ARM.pdf.

[18] Gartner Inc. Gartner says 8.4 billion connected "things" will be in use in 2017, up 31 percent from 2016. URL https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-

[19] Jonathan Katz. Universally composable multi-party computation using tamper-proof hardware. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4515 LNCS:115–128, 2007. ISSN 03029743. doi: 10.1007/978-3-540-72540-4_7.

[20] Ben Lapid and Avishai Wool. Cache-Attacks on the ARM TrustZone Implementations of AES-256 and AES-256-GCM via GPU-Based Analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11349 LNCS:235–256, 2019. ISSN 16113349. doi: 10.1007/978-3-030-10970-7_11.

[21] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595930272. doi: 10.1145/1111037.1111042. URL https://doi.org/10.1145/1111037.1111042.

[22] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software abstractions for trusted sensors. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, page 365–378, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450313018. doi: 10.1145/2307636.2307670. URL https://doi.org/10.1145/2307636.2307670.

[23] ARM Ltd. Arm security technology (building a secure system using trustzone technology). Technical report, 2009.

[24] ARM Ltd. Arm architecture reference manual (armv8, for armv8-a architecture profile). Technical report, 2020.

[25] Michael Michaud, Sylvain Leblanc, and Thomas Dean. *Malicious use of OMG data distribution service(DDS)in real-time mission critical distributed systems*. PhD thesis. URL http://hdl.handle.net/11264/1241.

[26] Michael James Michaud, Thomas Dean, and Sylvain P. Leblanc. Attacking OMG Data Distribution Service (DDS) Based Real-Time Mission Critical Distributed Systems. *MALWARE 2018 - Proceedings of the 2018 13th International Conference on Malicious and Unwanted Software*, pages 68–77, 2018. doi: 10.1109/MALWARE.2018.8659368.

[27] Microsoft. How windows 10 uses the trusted platform module, 2017. URL https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/how-windows-uses-the-tpm#bitlocker-drive-encryption.

[28] Microsoft. Early launch antimalware, 2018. URL https://docs.microsoft.com/en-gb/windows/win32/w8cookbook/secured-boot?redirectedfrom=MSDN.

[29] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 83–94, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581131992. doi: 10.1145/349299.349314. URL https://doi.org/10.1145/349299.349314.

[30] Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: Hacking tesla from wireless to can bus. Technical report, Tencent, 2017. URL https://www.blackhat.com/docs/us-17/thursday/us-17-Nie-Free-Fall-Hacking-Tesla-From-Wireless-To-CAN-Bus-wp.pdf.

[31] Sen Nie, Ling Liu, Yuefeng Du, and Wenkai Zhang. Over-the-air: How we remotely compromised the gateway, bcm, and autopilot ecus of tesla cars. Technical report, Tencent, 2018. URL https://i.blackhat.com/us-18/Thu-August-9/

us-18-Liu-Over-The-Air-How-We-Remotely-Compromised-The-Gateway-Bcm-And-Autopi
pdf.

[32] OMG. Data Distribution Service. pages 1–20, 2015.

[33] OMG. DDS Security. 2018.

[34] Nathanael Paul, Tadayoshi Kohno, and David C. Klonoff. A review of the security of insulin pump infusion systems. *Journal of Diabetes Science and Technology*, 5(6):1557–1562, 2011. doi: 10.1177/193229681100500632. URL https://doi.org/10.1177/193229681100500632. PMID: 22226278.

[35] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. ftpm: A firmware-based tpm 2.0 implementation. Technical Report MSR-TR-2015-84, November 2015. URL https://www.microsoft.com/en-us/research/publication/ftpm-a-firmware-based-tpm-2-0-implementation/.

[36] Thomas Reed. New ios exploit checkm8 allows permanent compromise of iphones, 2019. URL https://blog.malwarebytes.com/mac/2019/09/new-ios-exploit-checkm8-allows-permanent-compromise-of-iphones/.

[37] F. Righetti, C. Vallati, and G. Anastasi. Iot applications in smart cities: A perspective into social and ethical issues. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 387–392, 2018.

[38] Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. Technical report, NCC Group, 2019. URL https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2019/hardwarebackedhesit.pdf.

[39] Sandeep Tamrakar. *Applications of Trusted Execution Environments (TEEs)*. PhD thesis, 2017. URL https://aaltodoc.aalto.fi/bitstream/handle/123456789/26624/isbn9789526074634.pdf?sequence=1{&}isAllowed=y.

[40] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. volume 49, pages 67–80, 02 2014. doi: 10.1145/2541940.2541949.

[41] Stefan Saroiu and Alec Wolman. I am a sensor, and i approve this message. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, HotMobile'10, page 37–42,

New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300056. doi: 10.1145/1734583.1734593. URL https://doi.org/10.1145/1734583.1734593.

[42] Douglas C. Schmidt, Angelo Corsaro, and Hans Van't Hag. Addressing the challenges of tactical information management in net-centric systems with DPS. *CrossTalk*, 21(3):24–29, 2008.

[43] Rohit Sinha, Sriram Rajamani, and Sanjit A. Seshia. A compiler and verifier for page access oblivious computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 649–660, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi: 10.1145/3106237.3106248. URL https://doi.org/10.1145/3106237.3106248.

[44] Craig P. Szydlowski. Can specification 2.0: Protocol and implementations. In *SAE Technical Paper*. SAE International, 08 1992. doi: 10.4271/921603. URL https://doi.org/10.4271/921603.

[45] Sandeep Tamrakar, Jan-Erik Ekberg, Pekka Laitinen, N. Asokan, and Tuomas Aura. Can hand-held computers still be better smart cards? volume 6802, pages 200–218, 12 2010. doi: 10.1007/978-3-642-25283-9_14.

[46] TrustedFirmware.org. Op-tee documentation. URL https://optee.readthedocs.io/en/latest/index.html.

[47] Sven Türpe, Andreas Poller, Jan Steffan, Jan Peter Stotz, and Jan Trukenmüller. Attacking the bitlocker boot process. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5471 LNCS:183–196, 2009. ISSN 03029743. doi: 10.1007/978-3-642-00587-9_12.

[48] N. Wang, D. C. Schmidt, H. van't Hag, and A. Corsaro. Toward an adaptive data distribution service for dynamic large-scale network-centric operation and warfare (ncow) systems. In *MILCOM 2008 - 2008 IEEE Military Communications Conference*, pages 1–7, 2008.

[49] Thomas White, Michael N. Johnstone, and Matthew Peacock. An investigation into some security issues in the DDS messaging protocol. *Proceedings of the 15th Australian Information Security Management Conference, AISM 2017*, pages 132–139, 2017. doi: 10.4225/75/5a84fcff95b52.

[50] Alec Wolman, Stefan Saroiu, Paramvir Bahl, and Victor Bahl. Using trusted sensors to monitor patients' habits. In *1st USENIX Workshop on Health Security and Privacy (HealthSec)*, January 2010. URL https://www.microsoft.com/en-us/research/publication/using-trusted-sensors-to-monitor-patients-habits/.

[51] Jingyu Yang, Chen Geng, Bin Wang, Zhao Liu, Chendong Li, Jiahua Gao, Guize Liu, Jinsong Ma, and Weikun Yang. Ubootkit: A worm attack for the bootloader of iot devices. Technical report, Tencent, 2018. URL https://i.blackhat.com/briefings/asia/2018/asia-18-Yang-UbootKit-A-Worm-Attack-for-the-Bootloader-of-IoT-Devices-wp.pdf.

[52] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014.

[53] Bo Zhao, Yu Xiao, Yuqing Huang, and Xiaoyu Cui. A private user data protection mechanism in trustzone architecture based on identity authentication. *Tsinghua Science and Technology*, 22:218–225, 04 2017. doi: 10.23919/TST.2017.7889643.

This work has been partially supported by the Portuguese Foundation for Science and Technology (FCT), project REASSURE (PTDC/EEI-COM/28550/2017), co-financed by the European Regional Development Fund (FEDER), through the North Regional Operational Program (NORTE 2020).