DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

RAFAEL RODRIGUES GAMEIRO

Bachelor of Computer Science and Engineering

TWALLET
ARM TRUSTZONE ENABLED TRUSTABLE MOBILE WALLET:
A CASE FOR CRYPTOCURRENCY WALLETS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

TWALLET
ARM TRUSTZONE ENABLED TRUSTABLE MOBILE WALLET:
A CASE FOR CRYPTOCURRENCY WALLETS

RAFAEL RODRIGUES GAMEIRO

Bachelor of Computer Science and Engineering

Adviser: Henrique João Lopes Domingos
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2022

TWallet
ARM TrustZone Enabled Trustable Mobile Wallet:
A Case for Cryptocurrency Wallets

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.7.0) [42].

Para a minha família. Amo-vos muito.

Agradecimentos

Em primeiro lugar quero demonstrar o meu mais profundo agradecimento ao meu orientador de dissertação, Professor Henrique João Lopes Domingos. Obrigado pela oportunidade em poder embarcar neste grande desafio, pela contínua ajuda prestada, pela atenção, disponibilidade, paciência e motivação demonstradas ao longo do desenvolvimento desta tese.

Aos meus amigos, dentro e fora de FCT/UNL e a todos com quem me cruzei ao longo desta grande caminhada, sinto-me imensamente grato por vos ter conhecido e espero ter contribuído para que a vossa jornada se tenha tornado tão interessante como a minha se tornou.

A todos os colegas da faculdade, não só pelos momentos mais exigentes de trabalho e dedicação, mas também pelos momentos lúdicos que tornaram a minha experiência universitária algo único. Um especial agradecimento a António Ferraz e Rafael Almeida, pela amizade que partilhamos e pelo grande impulso que me deram e me continuam a dar, para procurar continuar a trabalhar e tornar-me melhor do que sou.

A Pedro Paixão, Paulo Pereira e Luís Pereira, sem dúvida que sem vocês a minha vida não seria a mesma que é hoje. Sinto-me imensamente grato por todos os momentos que já passámos e pela grande amizade que nos une. Por nada a trocaria. Espero bem que saibam que estas frase nunca farão jus à vossa grandeza!

A Hugo Rodrigues e Diogo Martinho, creio que este pequeno parágrafo não será suficiente para descrever todo o respeito e afeição e que tenho por vocês, espero que saibam disso. É verdade que já nos conhecemos há bastante tempo e sei que se fosse preciso poderia ficar aqui, páginas e páginas, a debitar todas as aventuras que já vivemos. No entanto, sendo breve, por todos os momentos de alegria e de tristeza já passados, o meu mais generoso Obrigado. Espero que a nossa irmandade se possa prolongar por muitos mais anos e possamos continuar a caminhar juntos lado a lado.

Por fim, não poderia deixar de agradecera a toda a minha família. À minha avó, tios, padrinhos, pai, mãe e irmão, por todo o apoio e força que sempre me têm dado e que me continuam a dar. Obrigado por não deixarem de acreditar em mim e nas minhas capacidades. Amo-vos muito.

"Today is victory over yourself of yesterday; tomorrow is your victory over lesser men." (Miyamoto Musashi)

Abstract

With the increasing popularity of Blockchains supporting virtual cryptocurrencies it has become more important to have secure devices supporting operations in trustable cryptocurrency wallets. These wallets, currently implemented as mobile Apps or components of mobile Apps must be protected from possible intrusion attacks.

ARM TrustZone technology has made available an extension of the ARM processor architecture, allowing for the isolation of trusted and non-trusted execution environments. Critical components and their runtime support can be "booted" and loaded to run in the isolated execution environment, backed by the ARM processor. The ARM TrustZone solution provides the possible enforcement of security and privacy conditions for applications, ensuring the containment of sensitive software components and data-management facilities, isolating them from OS-level intrusion attacks. The idea is that sensitive components and managed data are executed with a trust computing base supported at hardware and firmware levels, not affected by intrusions against non-protected OS-level runtime components.

In this dissertation we propose TWallet: a solution designed as a generic model to support secure and trustable Mobile Client Wallets (implemented as mobile Apps), backed by the ARM TrustZone technology. The objective is to manage local sensitive stored data and processing components in a trust execution environment isolated from the Android OS. We believe that the proposed TWallet framework model can also inspire other specific solutions that can benefit from the isolation of sensitive components in mobile Android Apps.

As a proof-of-concept, we used the TWallet framework model to implement a trusted wallet application used as an Ethereum wallet, to operate with the Ethereum Blockchain. To achieve our goals, we also conducted different experimental observations to analyze and validate the solution, with the implemented wallet integrated, tested and validated with the Rinkeby Ethereum Test Network.

Keywords: Trusted Execution Environments (TEE); Hardware-Backed Isolation; ARM TrustZone; Trusted and Secure Wallets; Trust Computing Base (TCB); Hardware-Enabled Isolated TCB

Resumo

Com o aumento da popularidade de *Blockchains* e utilização de sistemas de criptomoedas, tornou-se cada vez mais importante a utilização de dispositivos seguros para suportar aplicações de carteiras móveis (vulgarmente conhecidas por *mobile wallets* ou *mobile cryptowallets*). Estas aplicações permitem aos utilizadores uma gestão local, cómoda, confiável e segura de dados e operações integradas com sistemas de *Blockchains*. Estas carteiras digitais, como aplicações móveis completas ou como componentes de outras aplicações, têm sido desenvolvidas de forma generalizada para diferentes sistemas operativos convencionais, nomeadamente para o sistema operativo Android e para diferentes sistemas de criptomoedas.

As *wallets* devem permitir processar e armazenar informação sensível associada ao controlo das operações realizadas, incluindo gestão e consulta de saldos de criptomoedas, realização e consultas de históricos de movimentos de transações ou consolidação do estado destas operações integradas com as *Blockchains* remotas. Devem também garantir o controlo seguro e confiável do processamento criptográfico envolvido, bem como a segurança das respetivas chaves criptográficas utilizadas.

A Tecnologia ARM TrustZone disponibiliza um conjunto de extensões para as arquiteturas de processadores ARM, possibilitando o isolamento e execução de código num ambiente de execução suportado ao nível do hardware do próprio processador ARM. Isto possibilita que componentes críticos de aplicações ou de sistemas operativos suportados em processadores ARM, possam executar em ambientes isolados com minimização propiciada pelo isolamento da sua Base de Computação Confiável (ou *Trusted Computing Base*). A execução em ambiente seguro suportado pela solução TrustZone pode oferecer assim um reforço adicional de propriedades de confiabilidade, segurança e privacidade. Isto possibilita isolar componentes e dados críticos de possíveis ataques ou intrusões ao nível do processamento e gestão de memória ou armazenamento suportados pelo sistema operativo ou bibliotecas *middleware*, como é usual no caso de aplicações móveis, executando em ambiente Android OS ou noutros sistemas operativos de dispositivos móveis.

Nesta dissertação propomos a solução TWallet, uma aproximação genérica para suporte de wallets utilizadas como aplicações móveis confiáveis em ambiente Android OS

e fortalecidas pela utilização da tecnologia ARM TrustZone. O objetivo é possibilitar o isolamento de dados e componentes sensíveis deste tipo de aplicações, tornando-as mais seguras e confiáveis. Acreditamos que o modelo de desenho e implementação da solução TWallet, visto como uma *framework* de referência, poderá também ser utilizada no desenvolvimento de outras aplicações móveis em que o isolamento e segurança de componentes e dados críticos são requisitos semelhantes aos endereçados. Este pode ser o caso de aplicações de pagamento móvel, aplicações bancárias na área de *mobile banking* ou aplicações de bilhética na área vulgarmente chamada como *mobile e-ticketing*, entre outras.

Como prova de conceito, utilizámos a TWallet framework para implementar um protótipo de uma *wallet* confiável, suportável em Android OS, para gestão de operações e criptomoedas na *Blockchain Ethereum*. A implementação foi integrada, testada e validada na rede *Rinkeby Test Network* - uma rede de desenvolvimento e testes utilizada como primeiro estágio de validação de aplicações e componentes para a rede Ethereum em operação real. Para validação da solução TWallet foi realizada uma avaliação experimental. Esta avaliação envolveu a observação de indicadores de operação com verificação e comparação de diferentes métricas de operação e desempenho, bem como de alocação de recursos da aplicação protegida no modelo TWallet, comparando esses mesmo indicadores com o caso da mesma aplicação sem essa proteção.

Palavras-chave: Ambiente de Execução Confiável (TEE); Isolação por Hardware; ARM TrustZone; Carteiras Móveis Seguras e Confiáveis; Base de Computação Confiável (TCB); TCB Isolada por Hardware.

Contents

List of Figures

List of Tables

Listings

Acronyms

AOSP Android Open Source Project 10, 46, 47, 72, 94

API Application Programming Interface 10, 12, 20, 21, 31, 33, 37, 43, 45, 49, 50, 52, 53, 54, 56, 57, 60, 86

HCE Host-Card Emulation 4, 12, 13, 19

IoT Internet-of-Things 3, 19

JNI Java Native Integration 57, 60, 74

NS Non-Secure 17

NW Normal World 11, 17, 33, 43, 46, 49, 50, 52, 53, 54, 56, 57

OS Operating System 1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 13, 14, 15, 17, 19, 20, 21, 22, 23, 26, 27, 28, 29, 31, 45, 63, 71, 72, 73, 83, 94

SE Secure Element 12, 13, 23

SGX Intel Guard Extension 3, 11, 13, 23

SMC Secure Monitor Call 17, 20, 21

SoC System on Chip 3, 10, 11, 15, 16, 18

SW Secure World 11, 14, 16, 17, 18, 19, 20, 22, 46, 65, 75

TA Trusted Application 10, 20, 21, 22, 32, 33, 34, 35, 36, 43, 44, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 59, 60, 65, 67, 70, 86

TCB Trusted Computing Base 2, 3, 5, 11, 14, 15, 19, 20, 23, 24, 25, 26, 27, 29, 34

TEE Trusted Execution Environment 2, 3, 5, 9, 10, 11, 12, 13, 16, 17, 19, 20, 21, 22, 23, 27, 29, 31, 32, 33, 34, 36, 37, 42, 43, 44, 45, 46, 47, 48, 49, 52, 53, 56, 59, 72, 74, 85, 86, 87

TPM Trusted Platform Module 2, 13, 24, 34

TZASC TrustZone Address Space Controller 18

TZMA TrustZone Memory Adapter 18

VM Virtual Machine 7, 8, 14, 22, 23, 31

1 Introduction

1.1 Context and Motivation

Mobile devices and apps. The proliferation and relevance of mobile devices, such as mobile smartphones, tablets, and applications (Apps), have been increasing throughout the years, as nowadays the great majority supporting rich functionalities, including web-browsing, social networking, messaging, gaming, media processing and different location-aware operations, among others. At the same time, a new variety of applications ranging from pure entertainment, sensing-based interaction and possible coupled actuation capabilities and new forms of personal mobility assistance support, are now used in the current digital transformation. Many of these applications are used to support and process more and more sensitive information, requiring the appropriate enforcement of security, privacy and trustability control conditions for the provided functionality. As examples, we use today applications for business-oriented management processes; mobile payments, mobile banking, e-ticketing, healthcare monitoring, applications for digital multi-factor authentication proofs, dematerialization of relevant documents such as citizen identity cards, driving licenses, medical records, among others. Cryptocurrency wallets for different types of cryptocurrencies and related blockchains is another vague of recent applications with a notable increasing use.

Mobile apps and security issues. With the increasing popularity of mobile and ubiquitous apps, the amount of critical personal data stored in these devices and related operation control also increases, making the security and trust on mobile devices more than ever an important requirement. This necessity is visible from different published data, showing the huge growth of threats, attacks and incidents targeting the use of mobile devices in the Internet exploiting vulnerabilities of different operating systems and a huge amount of affected applications [64, 65, 20, 52, 16, 63].

Mobile apps and security mechanisms. Several security mechanisms and tools have been developed for the device Operating System (OS), to improve data privacy controls and to mitigate some security risks. However, in general, these tools and mechanisms

run on the assumption that the device OS, as well as runtime libraries supporting applications, are together part of the Trusted Computing Base (TCB). Nonetheless, attacks to the OS components with intrusions and data exfiltrations have already happened in the past and are nowadays a concern in addressing trusted mobile systems and applications. Meltdown [41] and Spectre [35] are common examples of attacks on the OS, and similar attack vectors also affected different Operating Systems, such as Android OS, Apple IOS, Microsoft Windows Mobile OS or Linux-based OS Distributions. Unfortunately, protecting data on mobile devices is far from trivial. Typically, mobile apps rely many times on ad-hoc OS and application-level mechanisms or application-level support libraries to protect sensitive data and prevent data leaks. Moreover, the TCB code that mobile apps depend upon is very complex: popular mobile platforms based on Apple IOS, Android, or Windows 8 comprise a full-blown OS, local services, and system libraries, consisting of millions of lines of code (LOC). This exposes a large threat surface for possible intrusions and attacks vectors to install BOTs, viruses or malware, breaking the integrity of runnable code or having illicit access to processed and stored data. Therefore, because of the lack of fine-grain isolation on execution conditions, it is difficult to ensure apps' protection, opening the door for exploits that can be used to disable security checks. This facilitates a way to retrieve sensitive data that can occur directly (by the installation of unsecure and malware applications and components that are, unfortunately, downloadable from "poor scrutinized" mobile app stores). On the other hand, some of those applications (apparently inoffensive at a first glance) can be used as indirect attack vectors against other critical installed applications (called deputy escalation attacks), exploiting the isolation deficiencies in OS runtime services or the unawareness of users when executing such applications. As an industry answer, leveraging the research results for those problems, laptops, smartphones, and tablets are now increasingly incorporating trusted computing hardware. For example, Google's Chromebooks use Trusted Platform Modules (TPMs) to prevent firmware rollbacks and to store and attest users' data encryption keys. Windows 8 (on tablets and phones) offers BitLocker full-disk encryption and support for the storage of virtual smart cards, also using TPMs. Recent research leverages TPMs to build new trusted mobile services. However, TPMs are very limited in the sense that they only provide functionality for boot attestation and possible storage of encryption keys in related hardware chips, not providing a way to execute the code of application components.

Trusted Execution Environments. More recently, the research community has studied the design of trusted computing systems based on small TCBs usable as Trusted Execution Environment (TEE). In this case, TCB components can be isolated and executed with improved access and fine-grain isolation control. Such solutions allow application developers to execute parts of the application logic in a trusted controlled environment, isolated from the generic OS runtime. Because only a few basic services can be offered in such trusted environment, the minimal setup for these systems addresses TCBs on the order of thousands of lines of code. While this prior research has managed to explore the limits in shrinking the TCB of trusted computing systems, the functionality of these

systems may be also too restrictive for mobile applications or easy porting of current applications, as they are structured, designed, and implemented today. Moreover, mobile apps are typically written in high-level languages and compiled to intermediate code (e.g., Dalvik bytecodes in the case of Android Apps).

Hardware-Backed Trusted Execution Environments (TEEs). An interesting direction is the possibility for structuring applications with components running on TCBs providing a Trusted Execution Environment, isolated as extensions to hardware processor architectures. This is the case of the ARM TrustZone technology provided for ARM System on Chip (SoC) systems, covering the processor, memory, and I/O support for peripherals. In this direction, the research on how to support applications leveraged from ARM TrustZone is a current trend in the agenda of research and development communities. In the same direction, this dissertation is particularly focused on the use of hardware-enabled TEEs. In general, a TEE is an execution environment that runs alongside but isolated from the main operating system (considered as the Rich OS) [26].

By using Hardware-Backed TEEs to execute sensitive components of the OS itself or critical application components, we can enforce security mechanisms by providing: (i) an isolated and secure execution apart from the Rich OS and other applications; (ii) isolation of cryptographic operations, with required primitives and management of keys supported inside the TEE and never exposed outside (iii) cryptographic components for data authentication, integrity and confidentiality, as well as, (iv) logging, monitoring and firmware based attestation control mechanisms, for the verification of trustworthiness conditions of components in the application critical software stack, with measured (or attested) boots for integrity checks that can be verified when required. Because our main focus in this thesis are the hardware-based TEEs, we highlight the most commonly used hardware-based TEEs, the Intel Guard Extension (SGX) [13] and ARM TrustZone [70], with a particular focus on the former, as leverage technology to build a new generation of secure and trustable applications that can be targeted to ARM-based devices that can include laptops, smartphones, tablets, smart TVs or other candidate devices in the Internet-of-Things (IoT) ecosystems.

ARM TrustZone. The ARM TrustZone technology [70] is used on processors that follow an ARM architecture in current chipsets and firmware enablers. Processors with different ARM Cortex architectures are today commonly found in laptops, smartphones, tablets, Smart TVs, wearables, and other IoT devices [4]. The use of ARM TrustZone technology for such devices can mitigate software intrusion attacks and the security of applications can be interestingly enforced. Because of this technology possible applications, the scientific community has started many projects related to the research of this component [50, 75, 33, 17, 36, 40, 74, 10, 31, 23, 45, 57]. Some of the projects are focused on providing stronger security properties to our everyday applications, making them more resilient against possible attacks [36, 31, 57]. However, not all applications are being equally researched, and the explosion of recent mobile wallets for cryptocurrency blockchains are an example of current research and development.

Trustable and Secure Mobile Wallets. Mobile wallets are applications where sensitive information regarding the management of valuable assets, cryptographic keys, and state of secure transactions based on cryptographic primitives are processed, making the user's device susceptible to attacks. Moreover, the usual security conditions provided to such wallets are typically based on encryption schemes where the keys are derived from users' passwords and cryptographic libraries running at user-space, making it more susceptible to dictionary attacks or software attack vectors injected by malware, virus, or worms. Because the private keys define the ownership of cryptocurrency data, they must be very well protected and secured under sound trustability assumptions. Therefore, we considered that research surrounding the usage of ARM TrustZone technology can enforce security methods for popular mobile wallets, used during the management of transactions and storage of keys, an interesting topic. Furthermore, it is interesting to research generic solutions in a stack of trustable components for such applications that could be also used for other application types. In this we include, for example, mobile banking, mobile ticketing, and the so-called Host-Card Emulation (HCE) applications [61] used to dematerialize the use of smart cards or memory cards for payments, ticketing or value-vouchers, developed as HCE software apps ready to run in mobile devices.

1.2 Problem

Focusing on the use of ARM TrustZone technology as a relevant enabler for trusted execution environments, the problem of this dissertation can be defined as the research of answers to the following research questions:

- How to design, with generic modelling concerns, a solution for Hardware-Backed Trustable Components for more robust, secure and trustable apps, particularly supporting a new generation of trusted crypto-wallets for cryptocurrency blockchains?

- Is it possible to address a generic model for such applications that can also be used as a development Framework with reference components to develop others with similar requirements?

- How to offer, within the framework model, components to help programmers to design more robust and secure applications to run on rich operating systems, such as Android or Linux OSes, but using the minimal trust computing base assumptions in the provided hardware-backed trust computing model?

- How can we measure the advantages of such provided solutions and, how to analyse the possible trade-offs or drawbacks in the operational performance and required resources?

1.3 Goals

To find the answers for the above questions, the main goal of this thesis is to develop a trustable model and runtime system offering trust computing based components in a ARM TrustZone enabling software stack, to protect sensitive applications. Our approach is to address the goal by designing a trusted framework model and related runtime that can be applied to a variety of applications that can benefit from the proposed solution. However, to define a more focused proof-of-concept in our objective, we target the use of our trusted model to develop a cryptocurrency wallet application for the Ethereum blockchain.

Our solution, named TWallet, provides generic bootable components in the hardware-enabled TCB for ARM devices, that can be reused by other cryptocurrency wallets, and possibly for other apps that can also benefit from the components we provide. Moreover, we intend to offer our solution as an extensible base to ease the development of a new generation of more robust apps requiring isolated trustable execution guarantees and secure processing of sensitive information.

The objectives beyond the TWallet solution are:

- TWallet-based application must have a small TCB and reduced attack surface, by considering the TCB isolated in hardware by any ARM TrustZone chipset. With this requirements met, the application will be able to protect the data it contains, through isolation, against possible attacks to rich operating system kernels and related OS libraries;
- The solution provides an easy-to-use interface for developers, allowing it to be reused and integrated with other existent applications, with a minimal porting effort. With that, systems can make use of the security properties we intend to offer, avoiding the programmers to be faced with the burden and low-level programming abstractions offered at the level of the ARM TrustZone firmware support;
- Since ARM TrustZone does not provide an OS to directly execute applications, the usage of a bootable secure micro OS is a necessity. The TWallet model is designed for the implementation of its runtime components on top of a standard base TEE-API (as defined by the GlobalPlatform consortium), adopting a compliant TEE - namely OP-TEE [39], as the base leveraged solution

1.4 Contributions

To achieve our goal and objectives, this dissertation addresses the following contributions:

- Definition of the TWallet System Model and Architecture as a generic framework offering a software stack for the development of trusted applications enabled by ARM devices equipped with ARM TrustZone compliant chipsets;

- Implementation of the TWallet prototype and its share as an open solution for use to the research and development communities, supported by a reference ARM TrustZone chip-enabled development platform, the Hikey 960 board [2];
- Development of a Ethreum Crypto Wallet App (publicly available in a distribution repository[1]) as a proof of concept prototype following the TWallet design model and integrated as a remote wallet to interact with the Rinkeby Ethereum Test Network - an Ethereum test network that allows for blockchain development testing before deployment on Mainnet, the main Ethereum network in production;
- Validation of the TWallet solution, with an extensive experimental evaluation that aims to test its behaviour in a real-case scenario, with assessment metrics comparing security and trustability benefits, and the inherent performance drawbacks and resources' utilization. The experimental evaluation was conducted in a comparative analysis of the operations performed in the TWallet protected crypto wallet and a non-protected version of the same, by measuring latency and throughput conditions; profiling indicators, resource allocation requirements, such as storage, memory or boot-latency observations, and analysis of developed attestation mechanisms of the running components in the ARM Trust zone Trust-Computing Base.

1.5 Report Structure

The remaining document is be organized as follows: Chapter 2 presents a background on the state-of-art and provides some related work references considering the objective of this dissertation; Chapter 3 presents the system model and its assumptions, as the system architecture and its implementation considerations; Chapter 4 describes the implementation and architecture realization of our solution, which includes all developed secure components functionalities and implementation options; Chapter 5 details the experimental evaluation and validation done to our developed solution and related prototypes, as also a discussion regarding the obtained results; To conclude, Chapter 6 presents our final remarks about the project, the faced issues and future work.

[1] https://github.com/rafagameiro/Thesis_TWallet

2

Background and Related Work

Taking into account the background and different aspects related with the expected goals planned for this dissertation, in this chapter we present the study of related work references we considered needed to its development. Therefore, the chapter is organized as follows: the concept of isolated execution and containment is addressed in section 2.1; Trusted Execution Environments, their definition and the most used ones are presented in section 2.2; In section 2.3 we discuss development platforms and describe their features and specifications; ARM TrustZone is explored more in detail, in section 2.4, where we present its system model and architecture behind this component, as its limitations and issues; Some works related to ARM TrustZone and projects done to increase the security properties of applications and other components are mentioned at the end of this section. Finally, we summarize the main topics discussed throughout this chapter, and explain which of the concepts and technologies are used in our implementation.

2.1 Containment and Isolation

Virtualization refers to a technology that provides an abstraction of the computing resources used by some software, which runs in a simulated environment called a Virtual Machine (VM). There are multiple types of virtualization, however, most of them are out of the scope of this project and our focus will only be full Virtualization. This type consists in the simulation of physical hardware to allow the execution of multiple full operative system instances.

In this section, we synthesize some of the principles in William Stallings's, Computer Security: Principle and Practice [73] by describing the principal alternatives of virtualization, inside the full virtualization type, and some security concerns that must be taken into account when making use of these alternatives.

2.1.1 Virtualization Alternatives

One or more virtual machines can be run on the same physical host machine, where each virtual machine, executes its own programs, and runs its own OS, called guest OS,

independently from the others. While executed in the host OS, virtual machines run as a process in an application window, similar to any other application. The separation of resources between the different virtual machines is done by a software called hypervisor [32].

The hypervisor sits between the hardware and the VMs and acts as a resource broker. This means, it safely shares the physical resources of the host resources among the VMs, allowing for its safe execution and coexistence. With this technology, two types of virtualization, distinguished by whether there is an OS between the hypervisor and the host, can be used: the native virtualization, and the hosted virtualization.

Furthermore, different from the hypervisor vision that aims to emulate the physical resources, another type of virtualization, called container virtualization, belongs to the full virtualization group. All these alternatives are presented below.

2.1.1.1 Native Virtualization

Native Virtualization allows for a direct control over the physical resources of the host. Once the host OS is installed and configured, the system is capable of supporting hosted virtualization on another guests OS. This type of virtualization is arguably more secure, as it has fewer layers that require the appliance of security mechanisms, contrary to what happens in the hosted approach.

2.1.1.2 Hosted Virtualization

Hosted Virtualization exploits the resources and functions of the host OS, running the guests OS on top of it. This makes the host OS, responsible for the hardware interactions on the hypervisor behalf. This type of virtualization is particularly beneficial to developers, who need to run multiple environments during project development, deployment, and testing. The created VMs can be easily migrated between hypervisors, reducing the deployment time and increasing the accuracy of what is deployed.

2.1.1.3 Container Virtualization

Container, or Application, Virtualization runs its software, known as virtualization containers, on top of the host OS kernel and provides an isolated execution environment for applications. Different from the hypervisors approach, this type of virtualization makes all containerized applications share a common OS kernel. This greatly reduce the overhead needed to run separated OSes per application. However, it can potentially introduce greater security vulnerabilities when compared with the other virtualization types.

2.1.2 Virtualization Security Concerns

Despite the benefits presented by these virtualization techniques, a number of security concerns that result from its usage must be considered [58].

Regarding the guest OS, to prevent invalid accesses to memory regions that belong to other OSes or even the hypervisor itself, it should be isolated. This would ensure that the programs executing within the OS can only access its OS resources, and nothing more than that.

Regarding the hypervisor, since it has privileged access to the programs and data that execute in each of the guest OSes, it must be secured from possible attacks against it that could compromise the component and its guests. Therefore, the hypervisor should be secure using a process similar to the ones used to secure an operative system. That is, it should be placed inside an isolated environment, to minimize the number of vulnerabilities. A possible solution could be the isolation of the hypervisor, inside an hardware-backed TEE. This solution, will be explored in further detail on section 2.2.3.

2.2 Trusted Execution Environments

To prevent security vulnerabilities that might arise in applications, the support provided by TEEs is an option that has been taken into account by the research community. However, to fully provide all security guarantees these secure environments offer, a trustable hardware-backed environment is a necessity. In this section, we provide an overview over some TEEs, presenting its main features. Afterwards, we discuss some hardware-backed technologies that offer certain security properties, beneficial to the use of TEEs. To conclude, we present different configurations that can be used in a TEE to enforce the security properties during the virtualization process.

2.2.1 Trusted Operative Systems

A TEE, or Trusted OS, is a tamper resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states, and the confidentiality of its code, data and runtime states stored on a persistent memory [56]. This section is focused on presenting some TEEs used in a research context and its capabilities.

2.2.1.1 OP-TEE

OP-TEE is a open-source Trusted OS, designed as a companion to a non-secure OS kernel, running on ARM processors, using the TrustZone technology [67]. It was primarily designed to rely on TrustZone technology as the underlying hardware isolation mechanism, however, it has been structured to be compatible with any isolation technology as long as it is able to enforce the TEE concepts and goals. OP-TEE is nowadays maintained by the Linaro Consortium [37].

The main design goals for OP-TEE are :

- Portability, since this environments aims at being easily pluggable to different architectures and hardware;

- Isolation, provided between the non-secure OS and the Trusted Application (TA), executed on top of this OS;

- Small footprint, which is translated to its size as OP-TEE is small enough to reside in the SoC memory only.

2.2.1.2 SierraTEE

SierraTEE is a Trusted OS that executes inside the ARM TrustZone technology [59]. It provides a minimal secure kernel which can be run in parallel with a Rich OS on the same core. The communication between the secure kernel and the Rich OS is achieved through the use of Global Platforms' Application Programming Interface (API)s [25]. This allows for an easier development of portable applications, since these APIs are considered a standard in TEEs development.

SierraTEE uses the ARM TrustZone security extensions to protect the secure kernel and any secure peripherals from code running in the normal world. This means if an attacker obtains supervisor privileges on the Rich OS side, it could not possibly inflict the secure kernel, as it cannot gain access to the secure world.

For systems without the security extensions, an emulated version can be used to provide a software environment fully compatible with SierraTEE on systems with the ARM TrustZone security extensions. An multi-core implementation of this Trusted OS is also available, in case the current system uses a separate ARM processor for security-related matters.

2.2.1.3 Trusty OS

Trusty [8] is a Trusted OS, part of the Android Open Source Project (AOSP) [27], that provides a secure environment for the Android OS. Trusty is presented as a reliable and open-source alternative to other secure environments, mainly closed ones since Trusty's transparency level is an advantage.

Trusty OS runs in parallel with Android OS in the same processor. This means that Trusty can have access to the full potential of the device it runs, while isolated from the rest of the system. The system isolation is done by both hardware and software. This Execution environment is not limited to a single processor model, as it is able to run in both ARM and Intel processors. On ARM processors, Trusty makes use of the TrustZone technology to create a secure Trusted Execution Environment.

The main components of Trusty are, a small kernel derived from Little Kernel [34], a linux driver kernel, and a user-space library. The driver kernel is used to transfer data between the secure environment and the Android OS. The user-space library is a library that enables communication with the executed Trusted Applications through the driver kernel.

All Trusty applications are developed by a single party and packaged with the Trusty kernel image. Therefore, the development of third party applications is not supported at the moment [8]. Nonetheless, Trusty enables the development of new applications, but explicitly states that it must be done with extreme care, as each new application causes an increase of the TCB. That itself can increase the attack surface, leading to new threats to the system.

2.2.2 Hardware-backed TEEs

To ensure all security properties provided by a TEE, a secure and trustable hardware-backed environment, capable of running the TEE, is needed. This environment must be able to provide a set of properties, including integrity, authenticity, and confidentiality. In this section, we present multiple techniques used to guarantee these properties.

2.2.2.1 Intel SGX

Intel SGX consists in a set of extensions to the Intel architecture that aims to provide integrity and confidentiality guarantees to programs [14]. By making use of these guarantees, programs can safely perform computations even if privileged software like the OS, hypervisor, or BIOS, are compromised. Intel SGX can also provide protection against physical attacks up to a certain degree, assuming the CPU package is not breached [13].

The principal abstraction in the SGX architecture is the enclave, an isolated execution environment within the virtual address space of a process. The code and data contained and executed inside the enclave are stored in a region of protected physical memory called Enclave Page Cache (EPC) [11]. While in there, the enclave contents are guarded by CPU access controls. When the information is moved to and from memory, the information in the EPC must be encrypted and decrypted respectively, since the code and data that run inside the enclave must never leave the CPU boundary unencrypted. Enclave memory is also integrity protected, meaning that memory modifications and rollbacks can be detected. It's important to notice that non-enclave code cannot access enclave memory, although the contrary can happen [13, 71].

2.2.2.2 ARM TrustZone

The ARM TrustZone technology is a set of security extensions found in many ARM processors. This technology was developed by ARM that aims to provide sound trustability assumptions into any platform that uses its architecture, which is nowadays widely used, particularly in mobile devices. By taking advantage of the hardware inside the SoC along with software components, this technology can provide a secure environment where the confidentiality and integrity of the applications it runs are assured.

TrustZone relies on the concept of worlds separated by hardware: a non-secure Normal World (NW) and a Secure World (SW). The non-secure world is where the OS and

most applications run, while the secure world is an isolated space where secure software is executed. Both of these worlds execute in a time-sliced fashion and the system guarantees that no secure component can be accessed outside the trusted environment. A secure monitor allows for the programs and applications to context switch between worlds. This division in worlds is not only in the process but also in memory, software, interruptions, and other components virtually separated so that it is possible to create a trusted platform where applications can execute without being exposed to possible software or hardware attacks [9]. The executed applications can be implemented using platform-independent API, standardized by Global Platform [25].

2.2.2.3 Secure Elements

A Secure Element (SE) is a tamper-resistant platform capable of securely hosting applications and storing their confidential and cryptographic data [24]. This component is an evolution of the traditional chip that was built for use in contactless credit cards (e.g, smart cards) and provides the user with a level of security and identity management to assure the safe delivery of a specified service. There are different forms of SE: embedded and integrated SEs (in the case of smart cards), on the SIM or Universal Integrated Circuit Card (UICC) or on an SD card. The different form factors exist to address the requirements of different business implementations and market needs.

The information stored in this special chip is impossible to access by normal applications, and only through trusted applications it is possible to read and copy its data. Also, the SE communicates directly with end-applications without passing any data to the device operative system. Therefore, if the device was infected with malware, the SE would be intact and no information stored in it would be intercepted by the attackers.

Despite the advantages this component can bring, it also presents some drawbacks. One of them is the availability of this chip, as not all mobile devices have access to it. Also, the devices that possess a SE bring an extra cost to mobile builders and their users. Another drawback is its limited functionalities when compared with other hardware-backed TEE solutions, like the TrustZone Technology. That's a major reason why in our case, the use of a SE is not an option.

2.2.2.4 Other Solutions

In addition to the previously mentioned technologies, other alternatives have been devised to provide basic underlying primitives for the creation of secure and trustable environments. Given the considerable number and variety of existing technologies, a few of them are presented below to demonstrate each ones approach to provide security guarantees to the applications.

HCE is a contactless technology that offers the option for the NFC controller in mobile devices to route communications from the contactless reader to an HCE service, on the

device host CPU, instead of the SE default option [61]. This HCE service can be part of a mobile application with a user interface, such as a mobile wallet for payment.
When using a mobile device with a SE without HCE, the NFC controller routes all the messages from the NFC reader to the corresponding application residing in the SE. If the device uses an HCE service, the messages can be routed to an application running on the host CPU. The decision is up to the operative system [55].

Sanctum[15] stems from a research initiative targeting RISC-V processors. Similar to SGX, Sanctum enables the creation of enclaves at the user level. The isolation provided by these enclaves provably defends against known software side-channel attacks, including cache timing attacks and passive address translation attacks. Sanctum also shows that a strong isolation of software modules can be achieved with low overhead. Despite the benefits offered by this TEE, and unlike SGX, the enclave memory is not encrypted, which can make the system vulnerable to physical attacks on DRAM [50].

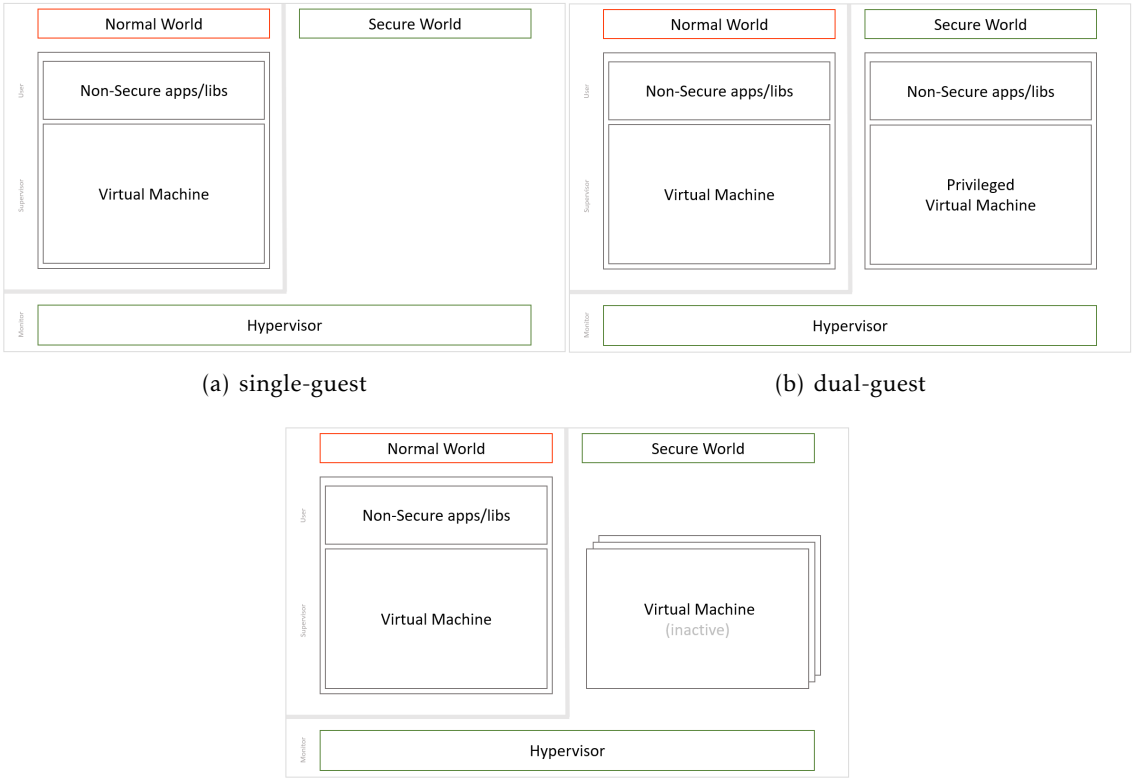
TPM, is a computer chip that can securely store artefacts used to authenticate the platform it is inserted[30]. Its primary purpose is to serve as a trustable source during the boot of the local platform, and securely store the cryptographic keys used from remote attestation, a process used to prove that a platform is trustworthy and has not been breached. However, this component cannot provide the means to execute security-sensitive code in isolation. Instead, it is used in tandem with trusted hypervisors or OSes that will then provide confidentiality and integrity protection to the applications it runs [50].

2.2.3 TEE-enabled Virtualization

As previously mentioned in section 2.1, to provide strong security guarantees, the hypervisor could be placed in an isolated environment. Given this thesis context, we will consider this isolated environment to be the ARM TrustZone.

The TrustZone technology, although implemented for security purposes, can be used as a form of system virtualization. Important to highlight that this type of virtualization is not considered either full virtualization or para-virtualization, because although the guest OSes can execute without changes in the non-secure world, they need to communicate between themselves to successfully manage the memory and address space usage.

Based on Sandro Pinto's work [50], regarding the TrustZone-assisted virtualization solutions, three configuration types are available: single-guest, dual-guest, and multi-guest. All configurations are represented in figure 2.1 and will be explained below.



(a) single-guest

(b) dual-guest

(c) multi-guest

Figure 2.1: TrustZone-assisted virtualization (based from [50])

2.2.3.1 Single-guest

As illustrated in Figure 2.1(a), the guest OS is executed under the non-secure perimeter, while the hypervisor runs in the secure world, in the monitor mode. This way, the hypervisor can visualize the entire system, while the guest OS has limited access to the system resources.

Using this configuration, the TCB becomes reduced to the code running in the SW, in this case being the hypervisor. All the resources used by the guest OS are considered non-secure as these are directly managed by the OS itself, while the remaining ones are under supervision of the hypervisor.

2.2.3.2 Dual-guest

This configuration is illustrated in figure 2.1(b), and consists in running each guest OS in its independent world, while the hypervisor, just like the single-guest, is executed in the monitor mode.

The dual-guest OS system is the most used configuration of the existing TrustZone-assisted virtualization options. The main motivation is because of the precise match between the number of consolidated VMs, and the number of virtual "worlds"existing

inside the processor. This configuration has been typically used for mixed-criticality systems. In these scenarios, a Real-Time Operative System (RTOS) runs in the secure world, while a normal OS runs in the normal world.

Since the privileged software runs in the secure world, the secure guest OS has a full view of the entire system, which means it is part of the TCB of the system. RTOSes typically have a reduced memory footprint, which makes them attractive candidates for such configurations.

2.2.3.3 Multi-guest

In the multi-guest configuration, the hypervisor executes in the monitor mode, while multiple guest OSes are encapsulated between the normal and secure worlds. During the execution of the system, the active guest OS runs in the normal world, while the context of the inactive guests is preserved in the secure world. Since guest OSes are able to run only on the normal world, the system's TCB is limited to the hypervisor size.

This configuration option is relatively new in comparison with the other two configurations, as for several years the TrustZone has been perceived as a limited and ill-guided virtualization mechanism [48]. However, recently, several works have been submitted [60, 49], hoping to further research and explore this new topic.

2.3 Development Platforms

ARM TrustZone main capabilities have been mentioned and our intent to make use of its functionalities were already expressed. However, this technology is not easily accessible. Most SoC hardware vendors do not provide access to their firmware and as a result, many developers and researchers are unable to find ways to deploy their systems or prototypes to TrustZone. In this section, we present some development platforms that will allow experimentation and evaluations on this technology.

2.3.1 TrustZone-enabled Platforms

To evaluate the development platforms we intend to study, table 2.1 presents a set of available alternatives by comparing them according to five dimensions: name of the platform, designation of the SoC, type of processor, number of cores, and the existence of publicly available TrustZone documentation.

As we can see, most presented boards have different SoCs, mainly because most of them belong to different manufacturers. Hikey 970 and 960 were both launched by Huawei, in collaboration with Linaro and Google respectively, using the HiSilicon processors, a subsidiary of Huawei, to make a board that offers their own based Kirin SoCs for both academia and industry [19, 62]. These boards were also presented as potential concurrence to the Raspberry PI board, one of the most used boards [76], either in a personal or research context. ODROID N2+ also emerges as another alternative to the

previous boards. Created by HardKernel, this family of development boards are mainly comprised of boards with ARM architecture in its SoCs, but also have some models with Intel processors [46].

Table 2.1: TrustZone-enabled Platforms.

Platform	SoC	Processor	Cores	Documentation
Hikey 970	HiSilicon Kirin 970	Cortex-A73/A53	octa-core	Yes
Hikey 960	HiSilicon Kirin 960	Cortex-A73/A53	octa-core	Yes
Raspberry PI 4	Broadcom BCM2711	Cortex-A72	quad-core	Yes
Raspberry PI 3 B+	Broadcom BCM2837B0	Cortex-A53	quad-core	Yes
ODROID N2+	Amlogic S922X	Cortex-A73/A53	hexa-core	No

Raspberry PI 4 presents a Cortex-A72, a less expensive alternative in comparison to the processors in the other manufacturer boards. Despite that, it is capable of yielding significant performance increases over its predecessor, the Raspberry Pi 3 [22].

Both Hikey and ODROID manufacturer groups decided to go with the same processor architecture and therefore, their boards have a Cortex-A73 paired up with an A53. However, the Hikey group seems to surpass the HardKernel board by having an A53 with four cores, making a total of eight cores. ODROID, while possess the same processor combination, it only has six cores, with the A53 being dual-core.

Considering the Hikey boards, the 970 model possesses a better SoC and a set of other specifications like internal storage and I/O peripheral in comparison with its predecessor, the Hikey 960. This is a natural occurrence, since the 970 model is a more recent board.

2.4 ARM TrustZone and Development Model

In section 2.2 we gave a brief explanation in what was the ARM TrustZone technology and its main qualities. This section aims to provide a deeper understanding on what is this component and what has been done to further experiment and understand its potentialities. With that in mind, we intend to present the TrustZone technology, when used to run trusted applications, with and without the need of a TEE. The TEE used in this section is the Linaro's OP-TEE [39]. In both cases the system model and architecture are presented, with an inside of each component and its contribution to the main system. Main issues and limitations of each configuration are discussed next, and which limitations must be taken into account during this thesis development.

2.4.1 Natively Supported Applications

Natively supported applications are applications predefined in mobile devices, whose subcomponent runs in the TrustZone secure world. Because an application in SW cannot

be executed without a TEE, manufacturers have implemented, proprietary or open-source, TEEs, to execute some applications subcomponents and make them safer. However, for practical effects the TEEs used are not publicly described and therefore in this section we consider that those TEEs do not exist.

Therefore, we intend to present the original model of ARM TrustZone and discuss its issues and limitations regarding this configuration mode. Also, given the research area surrounding this component, to develop applications with stronger security properties or simply enhance the security of already existent applications, we present some projects related to this theme.

2.4.1.1 Design Model and Architecture

The principal feature of ARM TrustZone consists in the introduction of two protection domains designated by the name of worlds: the SW and the NW. Figure 2.2 illustrates these concepts. The secure world consists of an isolated space where trusted software executed and critical data is managed. The common execution environment, or Rich OS, where the normal applications are executed, is located inside the normal world.

The processor can only operate in one of these worlds at a time, and the current world the processor executes is determined by the value of the 33rd processor bit, known as Non-Secure (NS) bit. The value of this bit can be read from the Secure Configuration Register (SCR), and it is sent throughout the system.

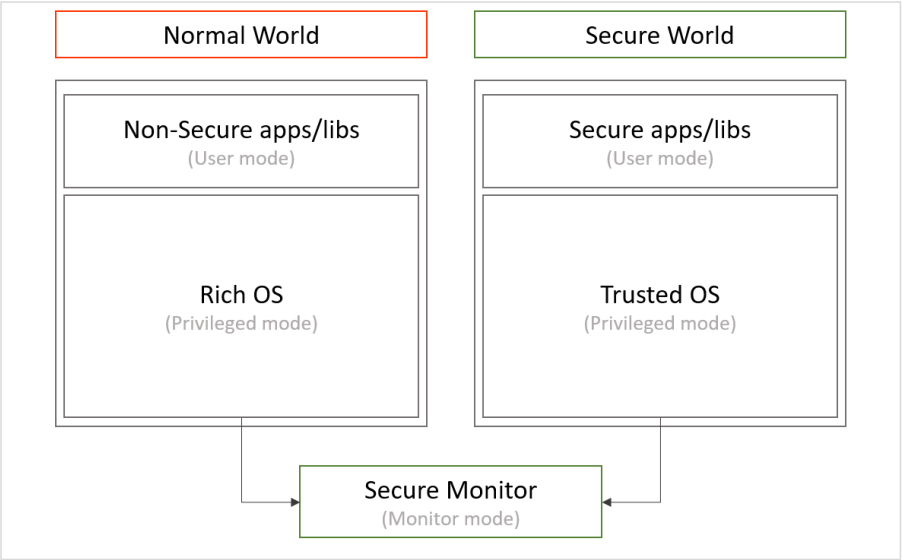


Figure 2.2: TrustZone System Architecture (based on [9])

TrustZone introduces a new processor mode, responsible for preserving the processor state when world transitions happens. This mode is nominated monitor mode, and acts as a bridge for placing the processor in the secure state, independently of the NS bit value. A new privileged instruction called Secure Monitor Call (SMC), is what allows the CPU to

enter in this new mode. Besides this instruction, it is possible to enter the monitor mode through exceptions, interruptions, and fast interruptions handled in the secure world [9].

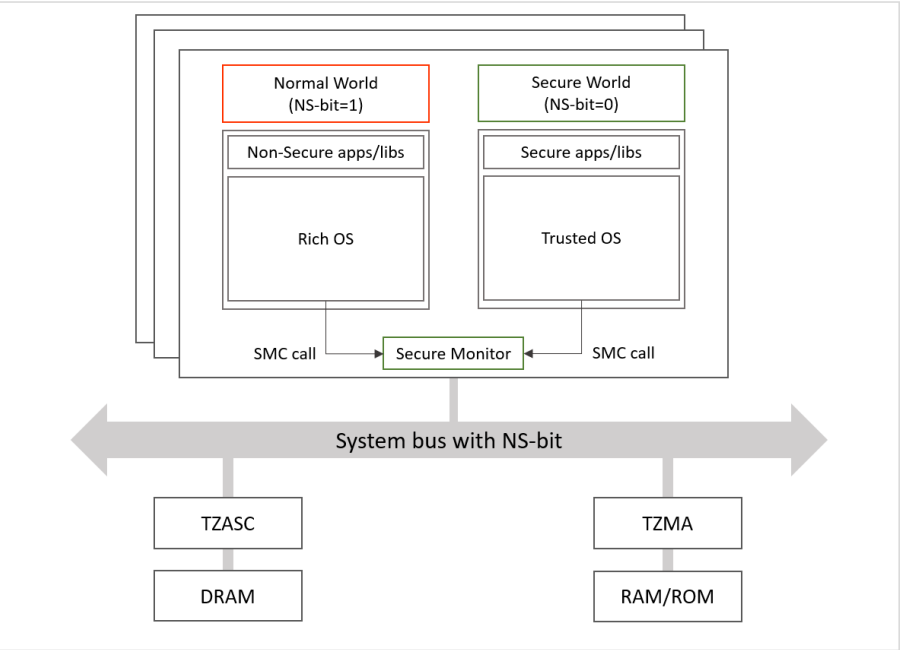


Figure 2.3: TrustZone System Architecture with additional components

Besides the concept of worlds and its context switch mechanisms, outside the CPU cores additional components can exists in the SoC implementation. These components are represented in figure 2.3, and are the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA), and they are responsible for extending TrustZone security features to the memory infrastructure. The TZASC can be used to configure specific memory regions as secure or non-secure, making applications running in the secure world able to access memory regions associated with the normal world, but not the contrary. A similar memory partitioning functionality is implemented by the TZMA, but targeting off-chip ROM or SRAM [50].

2.4.1.2 Issues and Limitations

Although TrustZone specification describes how the processor and memory are protected in the SW, and provides mechanisms to secure I/O devices, its remains silent to how many resources should be protected. This unreferenced specifications can become a issue during the development and implementation of this dissertation if not taken into account, since some of them could alter the design of our solution.

From the studied material [51], it is possible to identify some limitations in the TrustZone technology: No Trusted Storage, Lack of Secure Entropy and Persistent Counters, Lack of Secure Clock and other Peripherals, Lack of Virtualization, and Restrictive access. In our case we can consider these to be our main limitations:

- **Lack of Trusted Storage**, as there must exist enough secure storage to securely store all the necessary credential, keys, and data needed to secure our implementation.
- **Lack of Secure Entropy and Persistent Counters**, as the secure generation of keys needed to store the cryptocurrency or other necessary data require a secure source of entropy. Persistent counters might also be useful for cryptographic functions requiring a counter.
- **No System Environment**, needed to properly run and execute calls to our application. Therefore the usage of a Trusted OS (TEE) is needed. However, since this component is added by the programmer, other systems that might want to make use of our solution might also need to use the same TEE, as there is no direct translation between different TEEs.

2.4.1.3 TrustZone Enabled Applications

Several applications have been developed with the intent to make use of the security guarantees provided by ARM TrustZone.

Miraje Gentilal [23], just like our solution, studied how to increase the security properties of a cryptocurrency wallet by making use of TrustZone Technology. More precisely, this work consisted in analysing mobile and hardware wallets, and determine which functions should be implemented in the SW, increasing the security of its storage. Also, a study was done in how to process sensitive information in a more safely manner, using trustworthy and optimized cryptographic operations. The resultant wallet was more resistant to a series of attacks, namely dictionary and side-channel attacks, while enabling for a certain flexibility.

Nuno Santos et al. [57] designed and implemented a Trusted Language Runtime (TLR), a system for security-sensitive applications developed in .NET on mobiles devices. This system offer programming primitives that enable the execution of small application components inside the SW, isolated from the Rich OS and other applications. Furthermore, the integrity and confidentiality of the code and data are protected inside the TEE. Experimental evaluations done shown that this system achieved a significant reduction in the TCB of the studied applications, within an acceptable performance cost.

Alessandro Armando et al. [10] developed Trusted Host-based Card Emulation (THCE), an alternative to the existing card emulation solutions that relies on TEE, providing assets to the implementation of security critical applications. THCE proved to offer the same advantages of HCE systems, allowing developers to easily and freely implement NFC applications through this solution.

Le Guan et al. [31] presented TrustShadow, a designed runtime system that makes use of the TrustZone security guarantees to shield applications running on multi-programming IoT devices. With TrustShadow, there is no need to modify existing applications, as the security guarantees can still be granted. Moreover, security-critical applications on IoT

devices can be comprehensively protected even in the face of total OS compromise. Since the system imposes small overhead to these devices, the protection of its applications can be achieved in a lightweight manner.

Matthew Lentz et al. [36] developed a system called SeCloak, that uses a small TCB kernel allowing users to control peripherals on their mobiles devices. SeCloak runs in the SW of TrustZone and it can co-exists with the Rich OS without requiring any code modifications. The experimental evaluation demonstrated that mobile peripherals, like radios, cameras, and microphones, can be controlled in a secure and reliable way, with small performance overhead.

2.4.2 OS-Level TEE Assisted Applications

As previously mentioned in section 2.4.1 the natively supported applications executed in TrustZone secure world need the support of a TEE to properly execute. However, the manufacturers do not publicly describe the existence of a TEE in the mobile devices specifications. Moreover, it is not possible to run third-party applications or to reuse TEE components and services already provided by those manufacturers. As mentioned in section 2.2.1.1, OP-TEE is a open-source TEE, designed as a companion to non-secure OS kernel, running on ARM processor, using the TrustZone technology. Since TrustZone does not possess a system environment implemented to run its applications on top of an OS, and to run non-native applications a TEE is needed, in this section we intend to examine the TrustZone architecture behaviour, while deploying a TEE inside the secure world. Considering our thesis context, OP-TEE was chosen as the TEE of this architecture.

2.4.2.1 Design Model and Architecture using OP-TEE

In section 2.4.1.1 the TrustZone architecture was presented, along with its components and functionalities. In the secure world configuration, a trusted OS was needed to run any desired trusted application. In figure 2.4, we present the same TrustZone architecture, but with the Trusted OS component replaced by OP-TEE.

The OP-TEE Core, located inside the Trusted OS, controls the trusted OS as it allows the trusted applications to be executed. Also, it directly communicates with the secure monitor, sending and receiving messages between the normal and secure worlds. The secure monitor is a component that can be internal to the OP-TEE, as exemplified in the figure, in case the program responsible for the context switch between worlds is not identified during OP-TEE execution.

The TEE Internal API describes services that are provided to TAs. These services allow the applications to make use of the TEE internal libraries, as also to communicate with the OP-TEE Client through SMC calls. When the normal world communicates with the secure world, the normal world executes a SMC instruction. If the related service targets the Trusted OS, the monitor will switch to OP-TEE OS world execution. When the

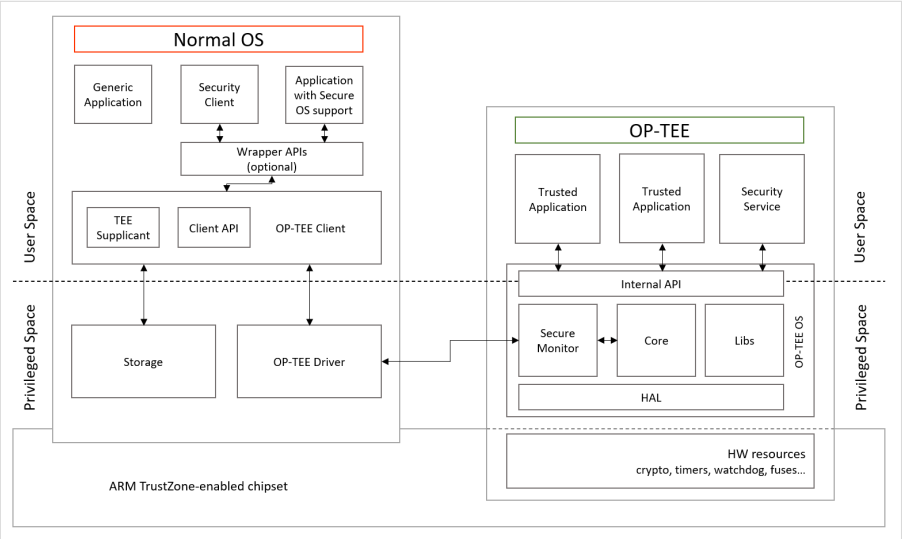


Figure 2.4: TrustZone System Architecture with OP-TEE (based on [38])

execution must return to normal world, OP-TEE OS executes a SMC that is caught by the monitor which switches back to the normal world.

Secure Storage in OP-TEE is implemented according to what has been defined in Global Platform's TEE Internal Core API. This specification mandates that it should be possible to store general-purpose data and key materials that guarantee confidentiality and integrity of the data stored and atomicity of operations. Two secure storage implementations are possible in OP-TEE, however, in this section only the default implementation will be described.

The default configuration states that the normal world file system is used as a secure storage space. When a TA calls a function to write data to a persistent object, a corresponding System Call through the Internal API is called, which will invoke a series of TEE file operations to store the data. The TEE file system will then encrypt the data and send to the Rich OS, commands and the encrypted data to the TEE supplicant, by a series of Remote Procedure Calls (RPC) messages. The TEE supplicant will receive the messages and store the encrypted data accordingly to the Linux file system. The reading process is handled in a similar manner. During the write and read process, the encryption and decryption of data is handled by a component called Key Manager.

The Key manager belongs to the TEE file system, and besides encryption/decryption of data, its also responsible for the management of sensitive key materials. There are three types of keys used by the key manager: the Secure Storage Key (SSK), which is a per-device key used to generate the second key; the second key is the Storage Key (TSK), a per-TA key used to encrypt and decrypt the third key; the File Encryption Key (FEK), which is the third and final key, is used to encrypt and decrypt the data stored in the secure storage space.

2.4.2.2 Issues and Limitations

Natively supported applications, to execute its components in TrustZone SW, need the use of a TEE. Because of that, manufacturers make use of proprietary or open-source TEE solutions to fully deploy those components. As the deployed TEEs only need some functionalities to run the desired applications, the manufacturers mitigate the overhead produced by the TEE inside the secure world, by deploying only the environment needed modules. Since OP-TEE is build as a TEE to allow execution of a wide range of different trusted applications, the overhead it brings in comparison to the TEE used by natively supported applications, it is bigger. Therefore, this overhead regarding resource allocation must be taken into account during performance testing.

As described in OP-TEE documentation, this TEE does not offer full support to all existent development boards, selecting only a few of them. Therefore, this limitation must be taken into account when setting up a development environment where OP-TEE is part of the development stack. However, even if OP-TEE does not offer direct support to a specific board, it might offer to its predecessor. Because of that, some minor alterations to the development board or the TEE might succeed in deploying the execution environment in the desired board.

OP-TEE can support virtualization. This is when OP-TEE runs a VM, using an hypervisor as an intermediary, and there executes TAs. With the virtualization support enabled, TAs are isolated and because of that, they won't be able to affect other TAs. At the moment of this thesis write, the virtualization support inside the OP-TEE is still experimental, and therefore, some limitations regarding the ARM architecture, hypervisor, and the number of VMs, should arise in case it exceeds the experimental limit.

Secure storage in OP-TEE consists, in the default configuration, in storing encrypted data in the normal world Rich OS's file system. The component responsible for the encryption and decryption of data is the Key Manager. This component is also responsible for the management of keys used throughout the encryption/decryption process. One of these generated keys, the Secure Storage Key, responsible for the generation of the other keys, requires an Hardware Unique Key (HUK) as a basis to derive this key. However, currently no OP-TEE platform supports the retrieval of this unique key, and as a replacement, a constant is being used. This results in no protection against decryption, or Secure Storage duplication to other devices, and should be taken into account when making use of this functionality.

2.5 Related Work

2.5.1 Summary

The main goal of this thesis is to develop an application capable of providing isolation and protection of sensitive data, managed through the use of the properties guaranteed by ARM TrustZone Technology. To achieve that, in this chapter we surveyed four themes we

considered essential to this project: Containment and Isolation methods, Trusted Execution Environments, Development platforms, and ARM TrustZone analysis and research.

We began by studying isolation and containment methods through virtualization techniques, in particular, full virtualization. This type allows to simulate physical hardware, and with that, execute multiple full operative systems at the same time. The full virtualization approach levels were explained, and with that, three alternatives were presented: native virtualization, hosted virtualization, and container virtualization. The security guarantees regarding these alternatives was also discussed.

Next, we studied the Trusted Execution Environments, in particular, the Trusted OS and the Hardware-backed options. The Trusted OS consists of an operative system that provide certain security guarantees to the applications it executes. The presented OSes were OP-TEE, SierraTEE, and Trusty TEE. This OSes can correctly enforce their security properties when inside an isolated environment that can guarantee integrity, authenticity, and confidentiality. Some of these environments were presented in section 2.2.2, highlighting Intel SGX, ARM TrustZone, and SEs. Furthermore, in the remainder of the section, TEE-enabled virtualization was presented, using the TrustZone technology as an example. It was discussed that by isolating a hypervisor and VMs it is possible to ensure inside the isolated environment, isolation of the executed applications and strengthen of security guarantees. With that in mind, three different configurations were presented: single-guest, dual-guest, and multi-guest.

Since the access to TEEs in mobile devices, in particular, the ARM TrustZone technology is still limited, we decided to research and analyse some development platforms used in industry or academic environments. This study was done with the objective to later setup our development environment. The boards were enumerated, presented, and its main specifications were compared among themselves to determine which could be a better alternative, considering our thesis goals. The Hikey 960 proved to be a better model, in terms of performance and efficiency, despite the 970 model being superior.

To conclude, the TrustZone technology was analysed in deeper detail, since this component is crucial for the development of our project. Its system architecture, as the principal issues and limitations that must be taken into account during the development phase, were presented. As a limitation, since the ARM TrustZone does not possess an embedded TEE, and considering our development stack for this project, the OP-TEE was used as an example to remove the component limitation. The architecture, main capabilities of the TEE were described and its issues and limitations were explained and discussed.

2.5.2 Critical Analysis

After analysis of the studied material ([23, 10, 17]) in deeper detail, and comparing with our proposed solution, some conclusions were reached.

Most of the studied solutions, [23, 17], intended to have a small TCB, mainly focused on components present in the secure world. Also, regarding verification methods of

these components and integrity checks, the same studied solutions addressed this point. Both these features, small TCB and code attestation, are also part of our proposed solution features. However, our solution goes one step forward by also providing attestation proofs measured at boot time, and scrutinised in runtime. This feature follows the measure and boot attestation functionality on TPM specifications, presented in [51].

We intend to develop a prototype using generic design principles so that its components could be reused by applications, like: ticketing, payment, or currency wallets. Some of the studied solutions, in particular [23] and [10], also addressed this goal with both solution offering its components to other applications that might request its services. Despite that, our solution goes further by providing more fine-grained generic components, like a logging service, a monitoring service, and an authentication service.

Regarding experimental evaluations and validation criteria in [23] and [17], the authors conducted an experimental analysis, discussing some results obtained through performed tests, as we also intend to do. However, we want to go further and provide a deeper analysis on our solution by measuring more parameters that we consider to be crucial to validate the wallet prototype. While the studied solutions measured secure storage performance and boot time, our intent is to not only observe the same criteria but also evaluate the percentage of allocated resources through fine-grained instrumentation and profiling observations, such as analysis of CPU workload, memory allocation, as well as operations' latency. For this purpose we must analyse the operation of each one of the trusted components in our trust computing base.

To conclude, despite the set of features these applications offer, and so does our solution, there is one feature in our project that stands out in comparison with the other features as this one is unique. Our proposed solution intends to offer a virtualization option to run our trusted secure components that the studied solutions did not mentioned. This virtualization options aims to isolate the secure components inside the secure world, so that the security between the different components is strengthen.

In the next chapter, we give an overview of our project, the TWallet System. We present the system architecture, adversary model considerations, and perform a deeper analysis over the components that make up our solution.

3

TWallet System Model and Architecture

In this chapter we provide a brief explanation with some guidelines and considerations about the developed system. To start, we present the system model and its architecture by discussing our adversary model assumptions, and explaining the system components that must be implemented for the proposed solution. After this, we start presenting our components in greater detail, where in section 3.2 we discuss the Secure Storage component restrictions and requirements, as also its desired architecture. In sequence, the next sections 3.3, 3.4, and 3.5 address the same aspects but focused on the Authentication Service, Logging Service, and Monitoring Service, respectively. Section 3.6 explains the TEE Adaptation and Isolation Layer functionalities, section 3.7 refers the Attestation Service, responsible for the attestation of our secure components before their usage, and section 3.8 presents the TWallet Framework model assumption regarding our solution. Finally we summarize our principal functionalities given a real world application context, concluding with a brief summary of the chapter.

3.1 System Model and Architecture Overview

When designing a secure system model and developing its architecture, it is important to take into account the adversary model assumptions, the target security properties, and the security mechanisms capabilities that try to protect the system. Given these concerns, we must discuss not only the advantages but also the constrains brought by the technology options of the desired systems. These trade-offs might, directly or indirectly, influence the design and implementation of systems, as well as its subsequent cases. In our context, given our objective to reduce the Trusted Computing base to the hardware-level through the execution of trusted environments, we must also consider its consequent constraints on computation resources, like memory and processing power. These restrictions must include not only the ones that are consequence of the use of our technology options, but also the resources spent that are out of our TCB. Throughout this chapter and subsequent

explanations regarding our system, all these constraints and requirements are taken into account.

For our design model assumption, we take as initial reference a popular OS used in most of the mobile devices, the Android OS. Regarding its current mobile ecosystem for Android, its stack of services and runtime libraries are layered as shown in Figure 3.1.

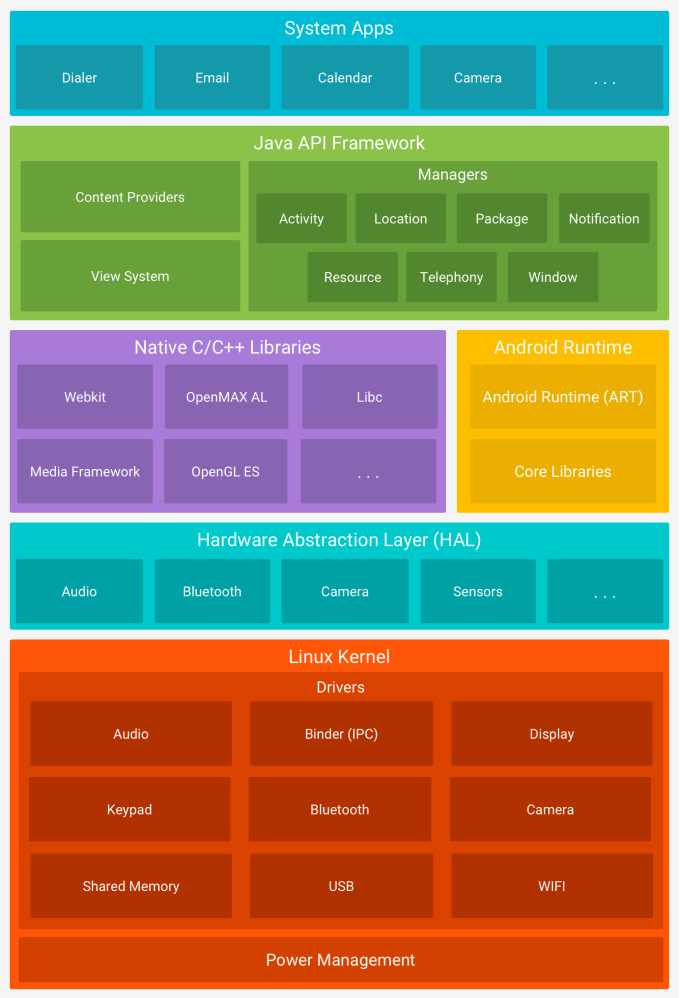


Figure 3.1: Android Architecture (extracted from [7])

As we can see in the figure, the application layer has its TCB provided by all the layers below it. This means that conventional applications consider all the layers and their components, from the Framework API layer to the Linux Kernel and Power Management layer, to be secure and trusted. However, we already expressed our concerns regarding the actual trust of these applications in the OS. Since attacks against it are already common [41, 35], we should make an effort to consider and remove the Android layers from the Application to the Kernel layer, from the TCB of all applications. Consequently, when considering all applications we take into special account our target type applications, the cryptocurrency wallets. This type of application might become vulnerable and insecure to certain attacks, by lacking the appropriate security and trustabilities mechanisms

needed to protect the managed data and its operations. In the referred vulnerabilities we highlight: (i) lack of trusted storage needed for store of wallet critical information, like access credentials and balance and transaction history information; (ii) lack of secure mechanisms to properly generate a log of activities, protected against tampering attacks; (iii) lack of trusted mechanisms, used to process the transactions on the client-side.

Even if we were to use cryptography to protect the managed data or to perform the wallet operations in an encrypted way, the processing of cryptographic operations, as well as the management of cryptographic keys, are also exposed to possible attacks, when these functions are executed in the vulnerable execution environment. The cryptographic keys itself are also exposed during the processed operations in memory, since these keys are located in the storage mechanisms provided by the Linux-Kernel Drivers and I/O functions related to storage services and storage devices.

To overcome the limitations imposed by the architecture and vulnerabilities presented above, our proposed architecture aims to reduce the current TCB, and consequently minimize the attack surface, to the hardware level. To effectively reduce the TCB to hardware-level, we decide to use a Trusted Execution Environment supported by the ARM TrustZone technology, with the security guarantees and aspects discussed in Chapter 2, Section 2.4. The TEE provides a set of guarantees and security related properties to the running applications, so that its operations and managed data can be protected from attacks against them. Despite this, the performed tests over this system setup must reflect the provided guarantees with an acceptable impact deviation in the use of resources, given the typical and real hardware platforms and their available resources.

With that in mind, we present a simple design model of our solution, that considers the Android OS and the TEE functions, executed by the ARM TrustZone, in Figure 3.2.

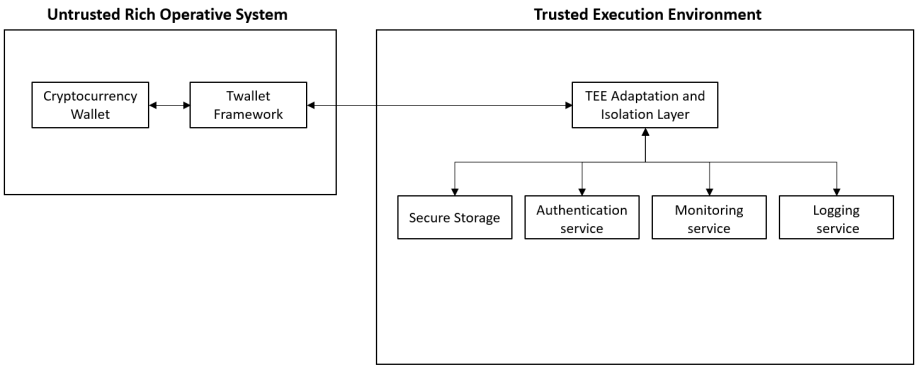


Figure 3.2: Design model of proposed solution

3.1.1 Adversary Model Assumptions

Based on the system model, we make the following adversary model assumptions, inspired by different threat models. Our defined adversarial considerations follows some of the adversary assumptions defined in previous works, like TPM [30] and TrustShadow [31]. Additionally, we also considered the ARM TrustZone threat model assumptions for a better specification and description of our solution.

Execution Environment Threat Model

This execution model describes possible attacks to the Operating System and the execution environment.

Root Attacker: We consider an attacker with root privileged, capable of having full control over the OS running in the normal world (Rich OS), to be the main source of untrustability, and potentially malicious. These privileges would allow an attacker, the ability to modify the system, compromising the OS and all related components, programs, and services. Furthermore, an attacker could mount multiple attacks against the secure world, its running programs, and consequently, our secure components. Some of those possible attacks are, execution of arbitrary code to tamper with memory and registers of a process, change secure world processes behaviour by hijacking system services, and request of invalid calls to TrustZone or not respond to requests coming from secure world.

Out-of-scope Assumptions

Regarding our Threat model assumptions, we considered the following threat-vectors to be out-of-scope:

- Physical, Side-channel attacks, or any other hardware related attack, and in particular any threat vector that tries to exploit vulnerabilities of ARM processors and its native functions;
- Exploits on software executed with the isolation guarantees inside the ARM Trust Zone Secure World Environment;
- Any attack vectors related to the issues and limitations present in the ARM TrustZone hardware and previously indicated in Chapter 2, Section 2.4.1.2

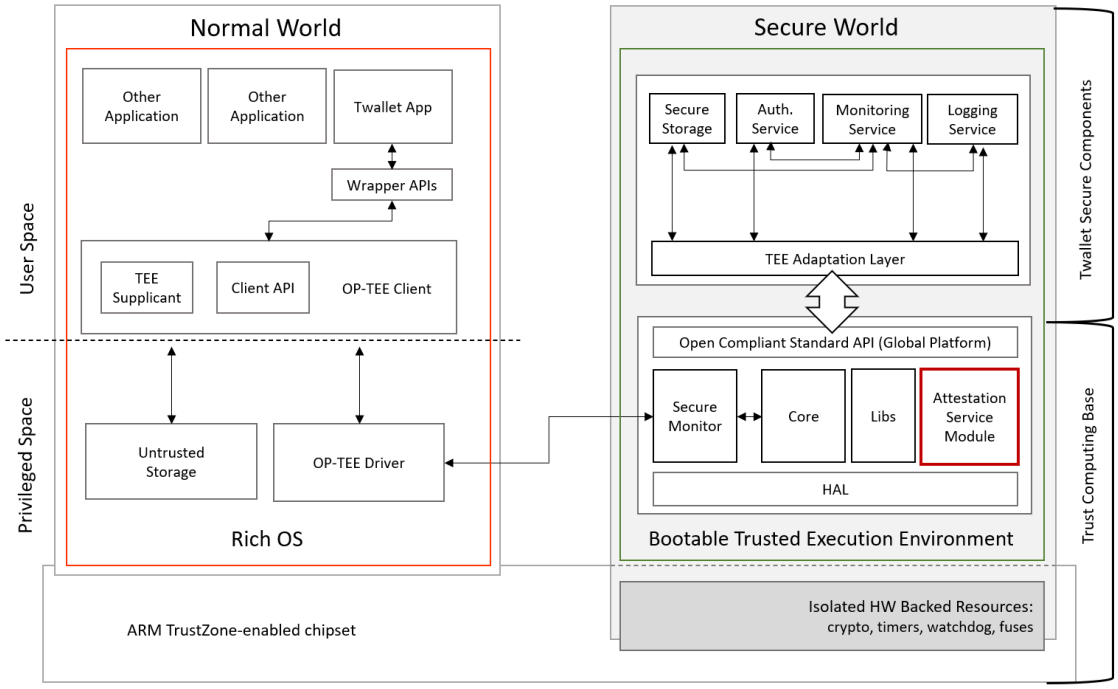
Furthermore, we consider that the hardware of the mobile device where our application is running, to be correct, as well as its libraries, highlighting the cryptographic ones.

3.1.2 Secure Architecture for Cryptocurrency Wallets

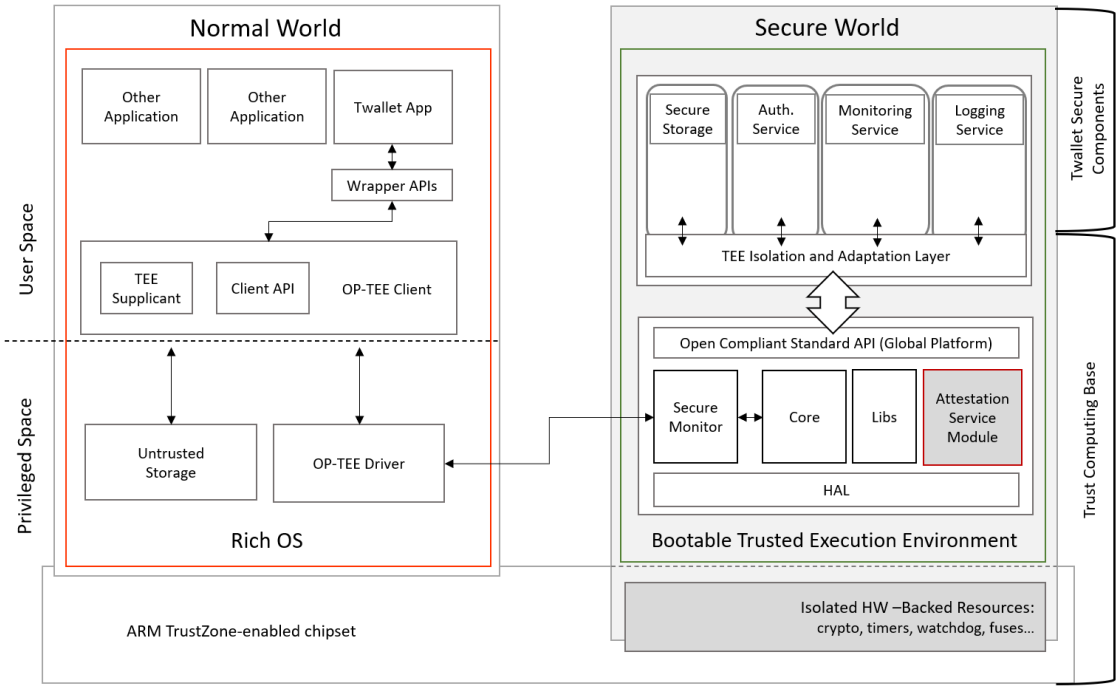
The proposed architecture addresses components executed under the trust guarantees provided by the ARM TrustZone isolation functionality, allowing for: (i) trusted storage of wallet credentials and data; (ii) secure generation and storage of a log of operations performed by these components; (iii) attestation of services provided, maintaining its trustability and security assumptions throughout its usage. The components intend to offer the necessary support for sensitive data management, under sound security and trustability assumptions.

As we previously explained, the Android OS is regarded as an untrusted Rich OS, and while considering the ARM TrustZone architecture it would be mapped in the Normal World, or Non-Secure World. However, our proposed solution's secure components will be mapped and executed inside the ARM TrustZone Secure World. Our intention regarding these architectural considerations, is to execute the secure and trusted components in a trustable execution environment, and with that, reduce the TCB and attack surface to shielded-hardware foundations. Therefore we can offer security services below the software layers for User Interaction Applications, Android Native Libraries, Hardware-Abstraction Layer and Base Linux Drivers and I/O support. We must remind that, as discussed in Chapter 2 Section 2.4.1, some native components on the Android System might also execute as components in the Trusted Execution Environment, mapped in the ARM TrustZone Secure World. However, this is not part of the dissertation scope that aims to provide support for cryptocurrency wallets by making use of our developed secure components in the TEE.

Summarizing our system model considerations, Figure 3.3(a) represents our final architecture when integrating our solution into OP-TEE system. The represented architecture consists of two main parts, the Normal World Component, TWallet, and the Secure World components.



(a) Simple Architecture



(b) TEE-Containerized Architecture

Figure 3.3: TWallet System Architecture

The TWallet library, located in Rich OS is the base application and it allows communication with the end-user by ordering operations and displaying the application data.

The secure components, namely the Secure Storage, Authentication Service, Monitoring Service, Logging Service, TEE Adaptation and Isolation Layer, and Attestation Service, will execute inside the ARM TrustZone Secure World, running on top of OP-TEE.

The Secure Storage is responsible for the storage of data regarding the logged wallet account. The Authentication Service provides a storage for the credentials, needed for wallet sign in. The Logging Services generates a log of the operations executed in the different secure components. The Monitoring Service is responsible for the monitoring and filtering of operations requested to the other secure components. The Attestation Service, based on [51], must provide integrity hash check regarding our secure components. The TEE Adaptation and Isolation layer is responsible for doing the conversion of calls displayed by OP-TEE Standard API, into calls that can be used by the secure components.

With the presented architecture in Figure 3.3(a), we can perceive that the data managed by the application is safe from the non-secure world and possible attackers that gain access to it. However, the secure components are not internally safe. This means that there is no security mechanism design to prevent threats from one of the components. To prevent possible threats regarding this case, and to further enhance the security and isolation methods, Figure 3.3(b) represents an alternative architecture to our system. The main difference of this architecture is the proposal of executing the secure components inside a containerized, isolated environment (e.g.VM). With the additional layer, although the component might suffer some limitations regarding the system resource availability (previously described in Section 2.2.3), each one will be isolated from the other components. This strengthens the secure components by providing an additional security layer at an internal level.

3.2 Secure Storage

The Secure Storage Component is responsible for the storage of any type of sensitive data, in a secure and trustable manner, and represents a database model capable of storing any type of data. Specifically in our context, the Secure Storage Component will be responsible for the secure storage of the wallet information, like balance and history of recent transactions. The design of this component must take into account the requirements: (i) a generic and flexible database; (ii) storage of any data content without following any specific schema; (iii) low memory footprint because of the TEE memory constraints; (iv) usage of safe and reliable methods of persistently storing the data this component manages, either by own implementation, or usage of the TEE own mechanisms.

Considering this requirements, the Secure Storage Component was initially designed as a simple key-value store, responsible for storing any data and having a footprint small enough to run successfully on top of the TEE Environment.

This application should follow the TA approach in order to guarantee the trustability and security of the data being processed. The TA approach can be defined as a methodology for performing all application processing exclusively in the Secure World side.

3.3 Authentication Service

The Authentication Service Component is responsible for the secure storage of any type of access credentials, and represents a database type model capable of storing any set of credentials. Specifically in our context, the Authentication Service will be responsible for the secure storage of the wallet credentials, like the wallet address and its password. The design of this component must take into account the requirements: (i) a generic and flexible database; (ii) storage of any type of access credentials; (iii) low memory footprint because of the TEE memory constraints; (iv) usage of safe and reliable methods of persistently storing the data this component manages, either by own implementation, or usage of the TEE own mechanisms.

Considering this requirements, the Authentication Service Component was initially designed as a component with a similar architecture as the Secure Storage Component, where the application has a simple key-value store, responsible for storing the credentials and having a footprint small enough to run successfully on top of the TEE.

This application should follow the TA or the Hybrid Trusted Application approach in order to guarantee the trustability and security of the data being processed. The Hybrid Trusted Application (HTA) approach consists in splitting the application processing between the Secure World and the Normal World.

3.4 Logging Service

The Logging Service Component is responsible for the secure management and storage of a log, describing the Secure Components' activity throughout its execution. Specifically in our context, the Logging Service will create an event log of the commands performed by each Secure Component, in order to produce an authentication proof of its activities. The design of this component must take into account the requirements: (i) registry of the activities performed by the different components; (ii) low memory footprint because of the TEE memory constraints; (iii) usage of safe and reliable methods of persistently storing the built log, either by own implementation, or by making use of TEE own storage mechanisms.

Considering this requirements, the Logging Service Component was initially designed as a simple log builder, where the application would simply insert new entries into the in-memory log, and in case of request, return the log built so far.

This application should follow the TA approach in order to guarantee the trustability, security, and authenticity of the generated log.

3.5 Monitoring Service

The Monitoring Service Component is responsible for the secure filtering and execution of the Secure Components commands, verifying which commands requested from the Normal World can be executed. The design of this component must consider the following constraints: (i) low memory footprint because of the TEE memory constraints; (ii) filtering of requests, by analysis of the command, secure component, and if its content is allowed to enter/leave the Secure World Environment.

Considering this requirements, the Monitoring Service Component was initially designed as a simple firewall, where the application would analyse the incoming requests and decide if these could be redirected and executed by the specific components, or if it should be refused its access. Additionally to executing this functionalities correctly, the component should have a footprint small enough to run successfully on top of the TEE Environment.

This application should follow the TA approach in order to guarantee the trustability and security of the filtering process.

3.6 TEE Adaptation and Isolation Layer

The TEE Adaptation and Isolation Layer is a component that serves as a middleware between the requests that come from the Normal World and the secure components API's endpoints. This component also serves, although indirectly, as a communication point between the secure components' different commands. This means that each command of the TEE Adaptation and Isolation Layer is in fact, a composition of different commands, from different secure components. That's why we mentioned that this component, besides its primary goal as a middleware, also serves a point of contact to the secure components.

In the design of this component we must take into account the following requirements: (i) low memory footprint because of the TEE memory constraints; (ii) interception of requests coming from the NW, as a way to redirect this requests to their respective secure components to be attended; (iii) point of contact between the different secure components commands.

Considering this requirements, the TEE Adaptation Layer was initially designed as a middleware component, where it would intercept the incoming requests, and redirect them to the desired secure components. To execute this functionalities correctly, the component should have a footprint small enough to run successfully on top of the TEE Environment.

Isolation Layer. In Chapter 2, Section 2.1, we studied isolation and containment methods through some virtualization techniques, like native virtualization, hosted virtualization, and container virtualization. In each of this alternatives, the advantages and security guarantees were discussed. The TEE Adaptation and Isolation Layer, aside from what

has already been mentioned, and as the name implies, also possesses an Isolation functionality, responsible for creating an virtualization environment for each of the secure components. The main motivation for this, is to further strengthen the security guarantees these components already have.

Therefore, taking into account this extra functionality, an additional requirement must be met: the capability of executing each of the secure components in an virtualized environment, preventing tempering from either, our own components, and our other running TAs.

Considering this new constraint, the TEE Adaptation and Isolation Layer must be designed as a virtualization environment's launcher where, when a command to a secure component is attended, the secure component itself must be executed inside an virtualization environment, and only after that, the request can served.

This application, with its full design considerations, should follow the TA approach in order to guarantee trustability, security and isolation over the executed secure applications, as its operations.

3.7 Attestation Service

The attestation process, following the TPM model, consists in the ability of requesting a certificate to cryptographically prove to a CA that the RSA key in the certificate request is protected by a TPM that the CA trusts [29]. With that in mind, our Attestation Service is responsible for performing the attestation process over our secure components, by providing a proof in how our components are considered secure.

Given that we consider the OP-TEE to be part of our TCB, the TEE will be trustworthy. Our objective is for OP-TEE to automatically launch our Attestation Service, making it also trustworthy and part of the TCB, as it was executed from a trustworthy source. With that in mind, every attestation proof generated by our service, of our secure components, will be trustworthy as it was made by a trustworthy component.

Taking into account the functionalities of the Attestation Service, we must consider the following design requirements: (i) low memory footprint, not only because of the TEE memory constraints, but also with the objective to not greatly increase the possible Attack Surface of our system; (ii) attestation of our secure components, by generation of a proof, considered trustworthy given the service software stack; (iii) Secure and trustworthy generation of keypairs, needed during the attestation process.

Considering this requirements, the Attestation Service was initially designed as an OP-TEE module, that each time a Normal World application would request an attestation proof of our secure components, the service would generate an attestation proof and send it to the requested application. By executing this functionalities correctly, the component should have a footprint small enough to run successfully on top of the TEE, and also to be part of it as a trustworthy component.

This application should follow the TA or the HTA approach in order to guarantee trustability, security and authentication over the generated attestation proof.

Attestation Protocol. The Attestation Service is responsible for authenticating the components of our solution in the OP-TEE environment, in successive stages, assuring that each component, as it is loaded, is a version that is approved for use, with a correct hash code for integrity purposes. As we previously mentioned, we consider the OP-TEE Boot process itself trustable and therefore included in our model assumptions. The setup of components during the attestation process can be illustrated in figure 3.4.

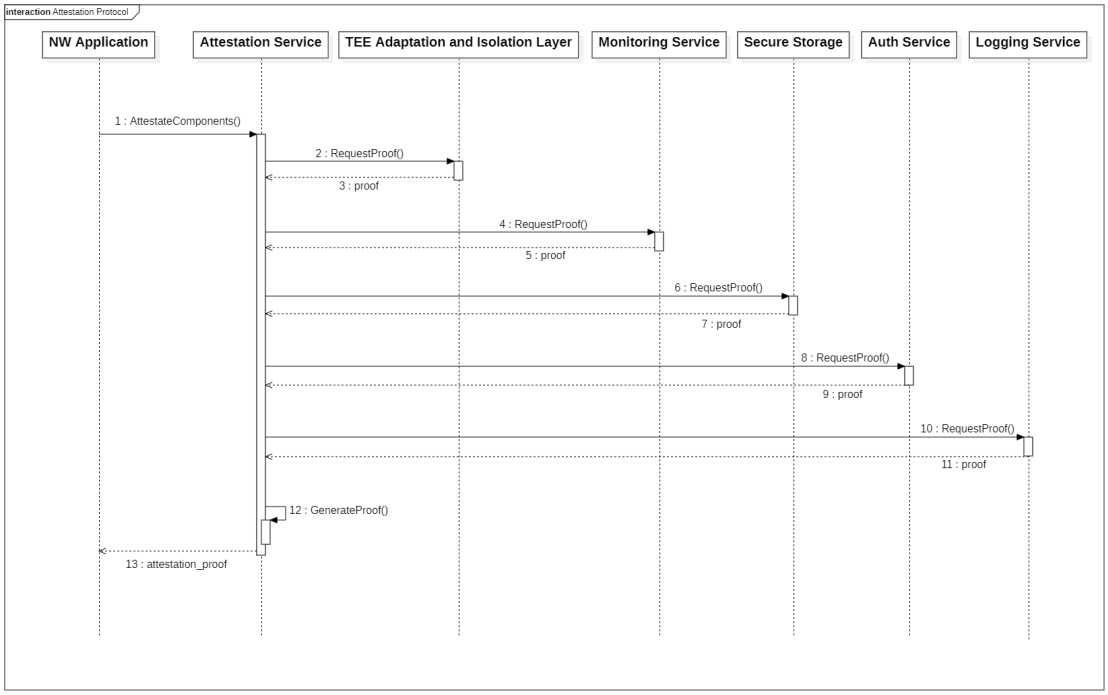


Figure 3.4: Attestion Protocol

As we can see, the process begins with the TEE Adaptation and Isolation Layer, as is the principal point of communication between the Normal World and the other secure components. Next, we have the Monitoring Service for the filtering functionality, needed for intercepting and restricting the requests send to the secure components. After that the Attestation service receives the hash of Secure Storage and Authentication Service, and finally, from the Logging Service. The final generated authentication proof is explained below:

$$\{ \text{cumulativeHash} \parallel \text{nonce} \}_{Kpriv}$$

- **cumulativeHash:** hash generated by the concatenation of all authenticated hashes retrieved from the secure components;
- **nonce:** a number sent by the Normal World application that must be appended to the proof before encryption. Used to prevent replay attacks.

The entire proof is then signed with the Attestation service private key so that we can prevent message forging from other TAs. After sending the proof to the Normal World, the application must verify the message signature and confirm that the attestation proof is deem trustworthy.

3.8 TWallet Framework

The TWallet Framework is the Normal World component, responsible for enabling communication between the secure components and all its services, located in the Secure World, and all interested apps in using our solution, in the Normal World. In this section we elaborate on the architecture the TWallet Framework is made of.

3.8.1 TWallet Framework and Library

The TWallet Framework as we already mentioned, is the service responsible for allowing applications to make use of our services. The basis of this service is the TWallet Library which is responsible for the communication between the service and the secure components located in the Secure World.

In the Secure world, the used TEE is responsible for the creation of operation requests and processing of the responses, receiving requests from the Normal World by normal applications using the TWallet Framework, and performing the requested operation on a specific secure component. This secure component is the TEE Adaptation and Isolation Layer, that will receive all operation requests that come from the Normal World, and then redirect them to the other secure components, that will perform the operation. The requests done by the Normal World should not contain any sensitive information, which mean, it should only contains information that when exposed does not compromise the security guarantees provided by the secure components. This also applies to their respective responses, which should not contain any sensitive plain information. To enforce this requirement, the Monitoring service is responsible for analysing the requests and respective responses, and guarantee that no critical information is leaked to the Normal World, and no threat against the secure components is attempted.

The normal procedure for performing and operation is: The user initiates an operation from the client application, which will be forwarded to the TEE in order to create the request. This request is redirected from the application to the TWallet Framework, which

by making use of the TWallet Library is able to send the request to the TEE to create it. The TEE after creating the secure request will send it to the TEE Adaptation and Isolation Layer, so that the Layer can forward the request to the secure component that will perform the operation. If needed, the Layer will first send the request to the Monitoring Service to analyse it, and only after that, redirect it to the specific secure application. The same procedure can be applied after performing the operation. After receiving the response, the TEE Adaptation and Isolation Layer will forward it to the client application so that it can be processed and do the required modifications. With this, the operation is now complete and the connection created between the Normal World application and the Secure World is shutdown.

In order to guarantee the trustability and security assumptions provided by the secure applications, this component should follow an Wrapper API approach, which offers those guarantees to the applications that make use of this component.

3.8.2 Supported Operations

As we previously mentioned, the TWallet Framework is responsible for requesting the execution of the operations that the secure components located in the Secure World have available. Therefore this component does not implement any operation whatsoever, but instead, it enables an API to request some operations to the secure components. Strictly speaking, we can say that the TWallet Framework has only two true operations: the create request operation which creates a request to be send to the TEE; and the process response operation that receives the response from the TEE. This response, depending on the request, must be processed and possibly demand modifications to the application state, either on an internal level, or on an interface level. In this section, we describe the TWallet Library functions that can be called from the Framework.

Store Credentials

Listing 3.1: Store Credentials

Pseudocode 1: Store Credentials

```
Input: id, address, password, id != NULL ∧ address != NULL ∧ password !=
  NULL
boolean result
result ← TEE_Invoke(STORE_CREDENTIALS, id, address, password)
if result then
  └ return result
else
  └ return ERROR
```

A new set of credentials is added to the Authentication Service with an address and password. The id can be generated in a varied number of ways, but we consider that the best method is to use an unique id, since typically only one account can be registered per

application. For the operation to succeed, it is important that both address and password are not empty (null).In case the operation was successful, a true value is returned. If the result value is false, the function returns an error since it could not conclude the operation.

Get Credentials

Listing 3.2: Get Credentials

Pseudocode 2: Get Credentials

```
Input: id, id != NULL
boolean result
result ← TEE_Invoke(GET_CREDENTIALS, id, address, hash_password)
if result then
    return [address, hash_password]
else
    return ERROR
```

The Get Credentials operation is listed in Listing 2. The operation retrieves the credentials from the Authentication Service storage, identified by its id. For the operation to succeed it is important that the id is not empty (null). In case the operation was successful, a set of [address, password] is returned. If the operation did not succeed, an error regarding the not found credentials is sent back.

Delete Credentials

Listing 3.3: Delete Credentials

Pseudocode 3: Delete Credentials

```
Input: id, id != NULL
boolean result
result ← TEE_Invoke(DELETE_CREDENTIALS, id)
if result then
    return result
else
    return ERROR
```

The Delete Credentials operation is listed in Listing 3. The operation deletes the credentials stored in the Authentication Service, identified by its id. For the operation to succeed it is important that the id is not empty (null). In case the operation was successful, a true value is returned. If the result value is false, the function returns an error since it could not conclude the operation.

Read Data

Listing 3.4: Read Data

Pseudocode 4: Read Data

```
Input: id, id != NULL
boolean result
string value
result ← TEE_Invoke(READ_DATA, id, value)
if result then
    return value
else
    return ERROR
```

The Read Data operation is listed in Listing 4. The operation retrieves the object from the Secure Storage's storage, identified by its id. For the operation to succeed it is important that the id is not empty (null). In case the operation was successful, the object value is returned. If the operation did not succeed, an error regarding the not found entry is forwarded.

Write Data

Listing 3.5: Write Data

Pseudocode 5: Write Data

```
Input: id, value, id != NULL
boolean result
result ← TEE_Invoke(WRITE_DATA, id, value)
if result then
    return result
else
    return ERROR
```

A new object with data is added to the Secure Storage Component with an id and value. The id can generated in a varied number of ways, but we consider that the best method is to use the address of the current logged in wallet, since typically only one object per wallet will be stored. For the operation to succeed, it is important that the id is not empty (null). In case the operation was successful, a true value is returned. If the result value is false, the function returns an error since it could not conclude the operation.

Delete Data

Listing 3.6: Delete Data

Pseudocode 6: Delete Data

```
Input: id, id != NULL
boolean result
result ← TEE_Invoke(DELETE_DATA, id)
if result then
    return result
else
    return ERROR
```

The Delete Data operation is listed in Listing 6. The operation deletes the object stored in the Secure Storage Component, identified by its id. For the operation to succeed it is important that the id is not empty (null). In case the operation was successful, a true value is returned. If the result value is false, the function returns an error since it could not conclude the operation.

Get Log

Listing 3.7: Get Log

Pseudocode 7: Get Log

```
boolean result
string log
result ← TEE_Invoke(GET_LOG, log)
if result then
    return log
else
    return ERROR
```

The Get Log operation is listed in Listing 7. The operation retrieves the log stored in the Secure World, that consists in a registry of all operations performed by the secure components. In case the operation was successful, the log object is returned. If the operation did not succeed, an error regarding the empty log is sent back.

Set Monitoring

Listing 3.8: Set Monitoring

Pseudocode 8: Set Monitoring

```
Input: value
boolean result
result ← TEE_Invoke(SET_MONITORING, value)
if result then
    return result
else
    return ERROR
```

The Set Monitoring operation is listed in Listing 8. The operation consists in defining if the Monitoring Service should filter or not the incoming requests from the Normal World. In case the operation was successful, a true value is returned. If the operation did not succeed, an error is forwarded.

Attest Components

Listing 3.9: Attest Components

Pseudocode 9: Attest Components

```
Input: nonce, nonce > 0
boolean result
string proof                          ▷ Attestation proof received from Secure World
string pub_key
result ← TEE_Invoke(ATTEST_COMPONENTS, nonce, proof, pub_key)
if result then
    if VerifyProof(proof, pub_key) then
        return true
    else
        return false
else
    return false
```

The Attest Components operation is listed in Listing 9. The operation consists in providing a proof that states that the secure components, currently keep their security and trustability guarantees intact. For the operation to succeed, it is important that the nonce value sent is bigger than 0. In case the operation that successful, the next step would be to validate the proof received from the Secure World. If the proof is correctly validated, the return result is true. If the proof is invalid, or the operation was unsuccessful, the result to return is false.

3.9 Real World Application Scenario

We consider now a real world case scenario, in which a user manages a cryptocurrency wallet backed by the TWallet solution. The user wants to keep track of its current balance, monitoring operations logged locally and performing transactions. The application must provide a set of features with security and trustability guarantees:

- The client is able to safely store its access credentials, so that they can later be reused to access the wallet information and perform other operations.
- The client is able to store the wallet information, such as balance and transaction history, in a secure manner. By making use of this, the client can protect this critical information against possible threats.
- The client is able to export and see a log of the secure actions performed throughout the application execution time.

The described scenario, can also be applied to other use cases, in which the used application is not a cryptocurrency wallet, but an application that possess a similar adversarial model, such as Ticketing apps, Mobile Banking, or even Mobile Payment apps.

3.10 Summary

In this chapter we discussed the system model assumptions for the proposed trusted runtime system and related architecture, in addressing the support for cryptocurrency wallets protected by a Trusted Execution Environment, and enabled by the ARM Trust Zone technology. Initially we presented an overview of the system model and its main architectural components, followed by some relevant considerations and assumptions that must be taken into account during the development of the proposed architecture. Then, we discussed our assumptions for the adversary model definition, explaining the considered threats and required countermeasures present in the proposed solution. In sequence, we addressed the design of the specific components of the architecture, namely the Secure Storage Component, the Authentication Service, the Logging Service, the Monitoring Service, the TEE Adaptation and Isolation Layer and the Attestation Service, as well as the TWallet Framework, used in user device applications, present in the Normal World.

Summarizing, the Secure Storage Component is responsible for the storage of any type of sensitive data, in a secure and trustable manner. Because of that, the component is designed as a simple key-value store, responsible for storing any data. The design of this component must consider the requirements: (i) a generic and flexible database; (ii) storage of any data content without following any specific schema; (iii) low memory footprint because of the TEE memory constraints; (iv) usage of safe and reliable methods of persistently storing the data this component manages, either by own implementation,

or usage of the TEE own mechanisms. With that in mind, this application should follow the TA approach in order to guarantee the trustability and security of the data being processed.

The Authentication Service is responsible for the secure storage of any type of access credentials, and represents a database type model capable of storing any type of credentials. This component must follow the requirements: (i) a generic and flexible database; (ii) storage of any type of access credentials; (iii) low memory footprint because of the TEE memory constraints; (iv) usage of safe and reliable methods of persistently storing the data this component manages, either by own implementation, or usage of the TEE own mechanisms. Considering all this, the Authentication Service is designed as a simple key-value store, with a structure similar to the Secure Storage Component. Therefore, it should follow the TA or the HTA approach in order to guarantee the trustability and security of the data being processed.

The Logging Service is responsible for the secure management and storage of a log, describing the Secure Components' activity throughout its execution. The component must comply with the constraints: (i) registry of the activities performed by the different components; (ii) low memory footprint because of the TEE memory constraints; (iii) usage of safe and reliable methods of persistently storing the built log, either by own implementation, or by making use of TEE own storage mechanisms. Considering its requirements, the Logging Service is designed as a simple log builder, where the application would insert new entries into the log, and return it if requested. The application should follow the TA approach, guaranteeing trustability, security, and authenticity of the generated log.

The Monitoring Service is responsible for the filtering requests sent to the Secure Components, verifying if the requests can be executed. The initial design of this application is as a simple firewall, where the application analyses the incoming requests and decide if these can be redirected and executed by the specific components, or if its access should be refused. The design of this component must consider the following constraints: (i) low memory footprint because of the TEE memory constraints; (ii) filtering of requests, by analysis of the command, secure component, and if its content is allowed to enter/leave the Secure World Environment. The application should follow the TA approach in order to guarantee the trustability and security of the filtering process.

The TEE Adaptation and Isolation Layer serves, not only as a middleware between the requests that come the Normal World, and the secure components API's endpoints, but also as a communication point between the secure components' different commands. Additionally, and as the name suggests, this component also possesses an Isolation functionality, responsible for creating an virtualization environment, for each of the secure components, preventing interactions between them. Considering all these functionalities, the design TEE Adaptation and Isolation Layer must follow the TA approach and its requirements are: (i) low memory footprint because of the TEE memory constraints; (ii) interception of requests coming from the NW, as a way to redirect this requests to their

respective secure components to be attended; (iii) point of contact between the different secure components commands; (iv) capability of executing each of the secure components in an virtualized environment, preventing tempering from either, our own components, and other running TAs.

The Attestation Service is responsible for performing the attestation process over our secure components, by providing a proof in how our components are considered secure. This component must perform an Attestation protocol, aiming for the authentication of each of the secure components in successive stages, assuring that each component, as it is loaded, is a version that is approved for use, with a correct hash code for integrity purposes. Therefore, the Attestation Service must follow the TA or the HTA approach, and its design constraints are: (i) low memory footprint, not only because of the TEE memory constraints, but also with the objective to not greatly increase the possible Attack Surface of our system; (ii) attestation of our secure components, by generation of a proof, considered trustworthy given the service software stack; (iii) Secure and trustworthy generation of keypairs, needed during the attestation process.

Finally we have the TWallet Framework architecture which combines the previous components, enabling for a secure and trusted processing of some cryptocurrency wallet operations. In the next chapter, considering the presented design requirements and constraints, we present a prototype of our solution and discuss its system model and architecture implementations choices.

4

Implementation

In this chapter, we describe the implementation and architecture realization of our presented solution. We start by defining our implementation environment in Section 4.1, explaining the underlying TEE system used, the development platform, and the setup required for our solution. After this, we start presenting our implementations. Section 4.2 presents the Secure Storage component, by describing the implementation choices made and how was designed, followed by its API. Next, we present the Authentication Service component through its implementation and API, in Section 4.3. Sections 4.4, 4.5, 4.6 and 4.7 follow the same structure as the previous sections but are directed to the Logging Service component, the Monitoring Service, the TEE Adaptation Layer, and the Attestation Service respectively. We finally conclude with the implementation and specification of our wrapper API, located in the Normal World, the TWallet Framework.

4.1 Implementation Environment

To properly develop, implement and test our solution, a development environment was needed. Based on the options displayed in the previous chapters, in this section, we present our choices made to ensure a good developing and testing ground for our solution.

4.1.1 Trusted Execution Environment

The TEE implementation chosen is the OP-TEE, previously described in Section 2.2.1.1. No further modifications were made to its kernel or OS code since the base version was more than enough to deploy our solution.

As previously described, this open-source TEE, currently maintained by the Linaro Consortium, was designed as a companion to a non-secure OS kernel running on an ARM processor. Our main motivation for choosing the TEE over the other studied ones, was due to its design principles, highlighting the portability feature. The portability of this TEE allows us to easily plug into different architectures and environments, making our proposed solution plug-gable to other architectures, different from the one used during this thesis development.

4.1.2 Development Platform

With the choice of TEE implementation selected, our options regarding the development platform became slimmer. As stated in the OP-TEE documentation [67], neither the Hikey 970, nor the Raspberry PI 4, both our desired development platforms, have support for the selected TEE. Despite that, some time was spent hoping to run OP-TEE on top of Raspberry PI 4, since at the time of this thesis writing, it has proven to be a more accessible board in comparison to Hikey 970. Unfortunately, even though the TEE did offer support to Raspberry PI 3, the great difference in the hardware configurations between the two boards made the port impossible.

Fortunately, we happen to have an Hikey 960 board at our disposition, which made us choose it as our development board for this project. The Hikey 960 board, not only has support for the OP-TEE, but also offer support for the Android Ecosystem through the AOSP, helping us in setting up our desired development environment: an Android OS running in the Normal World, and a TEE in the Secure World.

4.1.3 Development Platform Setup

To setup the development environment we followed a set of steps, described in Annex I. However, we took some additional procedures to properly execute our solution on top of the Android Ecosystem.

The reason was due to incompatibilities between the Hikey board and the available monitors to display the Android interface, mostly because of issues related to the connectivity of the Hikey board [1]. Additionally, to allow our app in the Normal World to communicate with the security components running in the SW, changes were issued to the Android permissions. The objective was to allow the Androids apps to communicate with the OP-TEE component located in the NW, the OP-TEE Client.

Therefore, to prevent those issues from happening, before compiling the build, we must configure a set of files. The additional configurations that need to be added are:

1. While in the <aosp root>/device/linaro/hikey/hikey960/, add the following to "BoardConfig.mk":

```
BOARD_KERNEL_CMDLINE += video=vfb:640x480-32@30 mode_option
   =640x480-32@30 vfb.videomemorysize=3145728
BOARD_KERNEL_CMDLINE += androidboot.selinux=permissive
```

2. Inside the "ueventd.common.rc", located at <aosp root>/device/linaro/hikey/, set:

```
/dev/tee0         0666   shell     shell
/dev/teepriv0     0666   shell     shell
```

3. In the <aosp root>/kernel/linaro/hisilicon-4.14 directory, execute the following set of commands:

```
$ git fetch https://android.googlesource.com/kernel/hikey-
   linaro refs/changes/96/889696/2 && git cherry-pick
   FETCH_HEAD
$ git fetch https://android.googlesource.com/kernel/hikey-
   linaro refs/changes/97/889697/3 && git cherry-pick
   FETCH_HEAD
$ curl https://github.com/vchong/linux/commit/64
   c9fd1c14b1c14b5b7510f18352ef33c2431002.patch | git am
```

After this, you can return to the project root and continue with the normal AOSP + OP-TEE build setup.

```
cd <aosp root>
rm -rf out
./build-p-hikey960.sh
```

4.1.4 Implementation Metrics

Regarding the development of our solution, its implementation was done using two programming languages, C and Java. The C language was mainly used in the development of the secure components that execute in the Secure World. The Java language was used in the development of the TWallet Framework since most of the Android applications are made in Java. Additionally in the Normal World, the C language was also used to make the connectors responsible for establishing communication channels between the TWallet Framework and the secure components themselves. Besides the TWallet Framework development, another application was created to launch our secure components as soon as the entire system booted. This application, called Components Init, was made using both Java and C languages.

Presenting some metrics obtained by sloccount [72], our project has a total of 53 C files and 3393 written lines of code. The Java language has a fewer number of lines and files, with only 20 and 331, respectively. All these files are available on a Github repository [1], so that other developers and researchers can make use of our implementation.

4.2 Secure Storage

The Secure Storage component follows a TA approach, having only the Trusted Application in the TEE, since all the processing is done by the TA, with no need for support from the Normal World.

[1] https://github.com/rafagameiro/TWallet_system

4.2.1 Implementation

Given its design requirements and characteristics, we decided to implement the Secure Storage service as a simple key-value store, based on hashtables where each key is associated with a set of values. This provides a more generic and flexible way of storing data, since the database does not follow a specific schema, and because of that, any type of data can be stored. Given our thesis theme, we decided that the key used to locate its associated data would be the wallet credentials, while the stored data would be an XML String. The XML would contain the balance of the associated wallet, and also a set of the most recent transactions sent and received in the wallet. Its implementation, considering the storage of wallet information, is represented in figure 4.1.

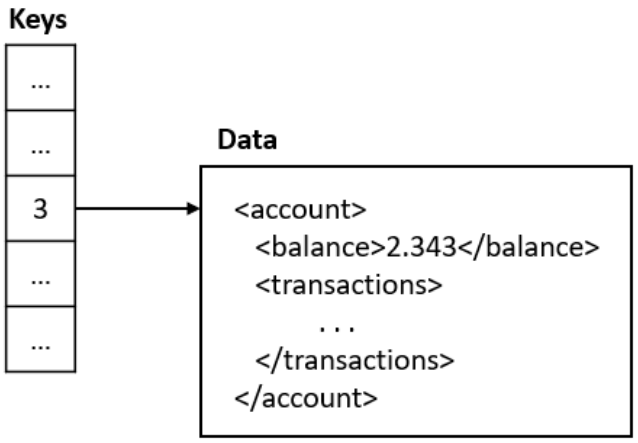


Figure 4.1: Secure Storage Implementation

The Secure Storage also supports secure data persistence, or "on-disk"secure storage, which is provided by OP-TEE Secure Storage [68] mechanism, that implements secure persistent storage in the TEE.

Each time a new entry is added to the hashtable, the same entry is not immediately stored in persistent memory. This operation only happens when the requesting caller closes the session with the TA, because of the serialization's data disposition, regarding the secure storage component and how its hashtable is organized.

Whenever a new session is created in the Secure Storage TA, all its data is loaded into memory before any request is attended. First, the TA retrieves the list of keys stored in the hashtable data structure, and then for each key, it will retrieve from the persistent memory its respective value. During this process, each key-value retrieval is concluded with its insertion into the hashtable, restoring the component to the same state it was before the TA session closure.

4.2.2 API

In this brief section we specify the Secure Storage TA API provided to client applications residing in the NW. All functions have its implementation logic explained.

- **TEE_Result create_item(char* id, char* data)** - Returns **TEE_SUCCESS** if the Secure Storage successfully creates an item.

  Performs a copy of the data stored in the arguments into local variables, and then creates an entry in the secure memory.

- **TEE_Result read_item(char* id, char* data)** - Returns **TEE_SUCCESS** if the Secure Storage successfully retrieves the data.

  Performs a search over the hashtable entries. If it finds a match that has the same id value as the one passed as an argument, copies the data to user-supplied buffer.

- **TEE_Result delete_item(char* id)** - Returns **TEE_SUCCESS** if the Secure Storage deletes the data.

  Performs a search over the hash table entries. If it finds a match that has the same id value as the one passed as an argument, deletes its data.

- **TEE_Result get_attestation_proof(void* proof)** - Returns **TEE_SUCCESS** if the Secure Storage successfully retrieves and sends a proof to the Attestation Service.

  The function retrieves an already generated proof regarding this service and copies it into the supplied buffer, proof.

4.3 Authentication Service

The Authentication Service component follows a TA approach, where only the Trusted Application located in the TEE does all the processing, with no aid from the Normal World. This approach was chosen since the implementation is rather simple and does not require any type of support from the Normal World.

4.3.1 Implementation

Considering the specifications and constraints for this component, the Authentication Service operates in a key-value store manner, where a key is associated with a specific wallet credentials. This method provides a way to store multiple credentials, where each one possesses a different key. In our case, this key is a UUID stored in the Normal World application, and the value, which corresponds to the wallet credentials, is stored as a string. Its implementation, considering the storage of a credential, is represented in figure 4.2.

Similar to the Secure Storage component, this TA makes direct use of the OP-TEE Secure Storage [68] mechanism, to store persistently the credentials of a wallet.

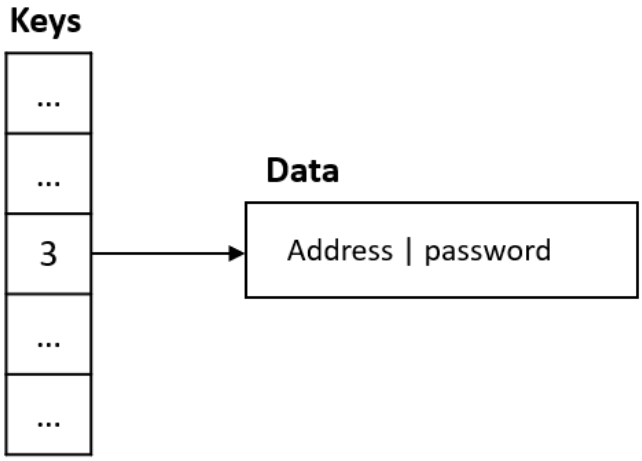


Figure 4.2: Authentication Service Implementation

Furthermore, and following the same logic as the Secure Storage TA, all new entries added to the component data structure, are only persistently stored in memory at the end TA session. The main motivation for this behaviour is due to the serialization's disposition and organization of data, inside the OP-TEE mechanism.

Regarding the data loading from persistent memory, the same procedure that happens in the Secure Storage TA occurs in this component. When the TA is instantiated, it will retrieve the list of keys stored persistently in memory. After this step, for each key, it will search in the OP-TEE Secure Storage and recover the associated value. During this process, each key-value retrieval is concluded with its insertion into the TA data structure, restoring the component to the same state as it was before the last TA termination.

4.3.2 API

In this brief section, we specify the Authentication Service TA API provided to client applications residing in the NW. All functions have their implementation logic explained.

- **TEE_Result store_credentials(char* id, char* data)** - Returns **TEE_SUCCESS** if the Authentication Service stores the credentials.

  Performs a copy of the credential data stored in the arguments into local variables, and then creates an entry in the hashtable.

- **TEE_Result load_credentials(char* id, char* data)** - Returns **TEE_SUCCESS** if the Authentication Service successfully retrieves the stored credentials.

  Loads the credentials stored in the hashtable, and write them into the supplied buffer, data. If no entry with the same id value was found, the function returns with an error and the buffer is sent empty.

- **TEE_Result delete_credentials(char* id)** - Returns **TEE_SUCCESS** if the Authentication Service deletes the credentials.

  Performs a deletion of the credential stored in the hashtable that has the same id value. If no entry was found, the function returns an error.

- **TEE_Result get_attestation_proof(void* proof)** - Returns **TEE_SUCCESS** if the Authentication Service successfully retrieves and sends a proof to the Attestation Service.

  The function retrieves an already generated proof regarding this service and copies it into the supplied buffer, proof.

4.4 Logging Service

The Logging Service component follows a TA approach, in which the Trusted Application located in the TEE does all the computation required for the component functioning. Since no type of support is required from the Normal World, we decided to go with this implementation strategy.

4.4.1 Implementation

Given the constraints and requirements, we decided to implement the Logging Service as a dataset of objects called log, where each object would consist of a log entry. For possible memory limitation issues, we decided to restrict the log to the most recent 100 operations performed by the secure components. Each time this limit would be reached, the service would automatically remove the 10 oldest entries to make space for the next entries that would come. Each entry can be resumed in a set of values statically defined: the name of the component, the operation executed, and the time the operation was requested. An implementation regarding the representation of a log entry is presented in figure 4.3.

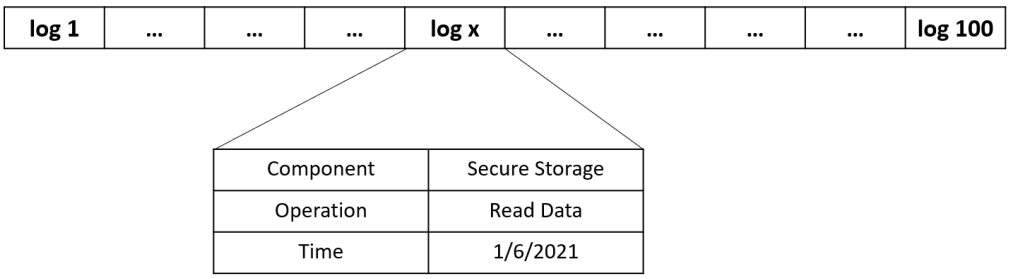


Figure 4.3: Logging Service Implementation

Equally to the Secure Storage and Authentication Services, the Logging service makes use of OP-TEE Secure Storage [68] mechanism to securely store the log entries into non-volatile memory.

Each time a new entry is added to the log, the same entry is not stored in persistent memory until the requesting caller closes the session with the TA. The motive is not only due to optimization purposes but also due to the store process, which consists in serializing and saving the entire log dataset as a single object.

Whenever a new session is created with the Logging Service TA, the log will be loaded into memory before the request is attended. The process consists in retrieving the entire object from the OP-TEE's Secure Storage and de-serializing and process each log entry until the service successfully reconstructs the log state before the TA termination.

4.4.2 API

In this brief section, we specify the Logging Service TA API provided to client applications residing in the NW. Note that this TA has some functions restricted to Secure World only. In all functions, its implementation logic is explained.

- **TEE_Result log_new_entry(char* service, char* command)** - Returns **TEE_SUCCESS** if the Logging Service successfully stores the information.

  Creates a new entry in the log, and fills it with the information passed by the arguments. If the log has reached its max entries limits, it will first remove the oldest entries.

- **TEE_Result log_read_data(char* data)** - Returns **TEE_SUCCESS** if the Logging Service retrieves the logging stored.

  Performs conversion of the log data stored in memory into a string, and copies it into the supplied buffer, data.

- **TEE_Result get_attestation_proof(void* proof)** - Returns **TEE_SUCCESS** if the Logging Service successfully retrieves and sends a proof to the Attestation Service.

  The function retrieves an already generated proof regarding this service and copies it into the supplied buffer, proof.

4.5 Monitoring Service

The Monitoring Service component follows a TA approach, where all the processing is solely done by the Trusted Application running on top of the TEE. We considered this the best approach regarding our implementation since no aid is needed from the Normal World.

4.5.1 Implementation

Regarding this component requirements and characteristics, the Monitoring Service was treated as a rather simple TA. This means that the component contains a set of volatile

data, needed to monitor the other secure components activity running in the Secure World. Whenever a new session is created with the Monitoring Service, the TA would initialize the dataset with the information required for a well functioning of the component.

We considered that the information needed to effectively filter and monitor any request coming from the Normal World should be summarized as the component whose request was being directed to and the list of available requests that can be done from an application located in the NW. This way, in case an application would request an operation, not available from the Normal World perspective, the Monitoring Service would intercept the request and discard it.

Furthermore, in case an application wishes to completely disable the Monitoring Service functionalities, the TA provides an operation that allows skipping any validation done to incoming requests done to the secure components.

4.5.2 API

In this brief section, we specify the Monitoring Service TA API provided to client applications residing in the NW. Please notice that, although this API has some functions restricted to Secure World only, all operations have their implementation logic is explained.

- **TEE_Result trigger_filter(bool filter)** - Returns **TEE_SUCCESS** if the Monitoring Service successfully sets the filter.

  Sets the value stored in the filter variable to the one stored in the argument filter.

- **TEE_Result filter_op(int service, int command)** - Returns **TEE_SUCCESS** if the Monitoring Service allows for the operation to be performed.

  Verifies if the operation in the specified service can be performed. To do it, the function checks if the specified service and operation, both passed as arguments, are accessible from the Normal World.

- **TEE_Result get_attestation_proof(void* proof)** - Returns **TEE_SUCCESS** if the Monitoring Service successfully retrieves and sends a proof to the Attestation Service.

  The function retrieves an already generated proof regarding this service and copies it into the supplied buffer, proof.

4.6 TEE Adaptation Layer

The TEE Adaptation Layer component follows a TA approach, where all its processing is done only by the Trusted Application running on top of the TEE. Since this TEE does not require any support from the Normal World, we considered this the best approach.

4.6.1 Implementation

Considering this component constraints and characteristics, the Adaptation Layer was treated as a rather simple TA. The principal function of this component is to intercept the requests coming from the Normal World, and properly attend to them. The correct procedure to treat these requests consists in verifying if the request can be done by the Normal World, calling the respective TA function, and if it is a request that must have its activity registered, register its activity. This procedure allows separating the execution of each component, preventing the possibility of intertwining operations between components.

To verify the correctness of a request, the TEE Adaptation Layer calls the Monitoring Service to guarantee the requester from the NW has the authorization to call the function associated with an incoming request. If authorization is conceded, the request can be forwarded to the respective component, but if not, an unauthorized access error e sent to the NW application.

To register the activity of a certain set of requests, the Adaptation Layer call for a Logging Service function, `new_log_entry(char* service, char* command)`, to record the request made to the respective secure component.

4.6.2 API

In this brief section, we specify the Adaptation Layer TA API provided to client applications residing in the NW. In all functions, its implementation logic is explained.

- **TEE_Result store_credentials(char* id, char* credentials)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully stores the credentials.

  The function starts by requesting permission to execute the Authentication Service's store credentials, by calling the Monitoring Service's filter operation. If The operation is allowed, the function will execute it, otherwise, the TA sends back an unauthorized access error. Additionally, after the operation is successfully executed, calls the Logging Service new log entry, to register a new entry, regarding the executed operation, into the log.

- **TEE_Result load_credentials(char* id, char* credentials)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully loads the credentials.

  The function starts by requesting permission to execute the Authentication Service's load credentials, by calling the Monitoring Service's filter operation. If The operation is allowed, the function will execute it, otherwise, the TA sends back an unauthorized access error. Additionally, after the operation is successfully executed, calls the Logging Service new log entry, to register a new entry, regarding the executed operation, into the log.

- **TEE_Result delete_credentials(char* id)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully deletes the credentials.

  The function starts by requesting permission to execute the Authentication Service's delete credentials, by calling the Monitoring Service's filter operation. If The operation is allowed, the function will execute it, otherwise, the TA sends back an unauthorized access error. Additionally, after the operation is successfully executed, calls the Logging Service new log entry, to register a new entry, regarding the executed operation, into the log.

- **TEE_Result storage_read_data(char* id, char* data)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully reads the data.

  The function starts by requesting permission to execute the Secure Storage's read item, by calling the Monitoring Service's filter operation. If The operation is allowed, the function will execute it, otherwise, the TA sends back an unauthorized access error. Additionally, after the operation is successfully executed, calls the Logging Service new log entry, to register a new entry, regarding the executed operation, into the log.

- **TEE_Result storage_write_data(char* id, char* data)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully stores the data.

  The function starts by requesting permission to execute the Secure Storage's create item, by calling the Monitoring Service's filter operation. If The operation is allowed, the function will execute it, otherwise, the TA sends back an unauthorized access error. Additionally, after the operation is successfully executed, calls the Logging Service new log entry, to register a new entry, regarding the executed operation, into the log.

- **TEE_Result storage_delete_data(char* id)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully deletes the data.

  The function starts by requesting permission to execute the Secure Storage's delete item, by calling the Monitoring Service's filter operation. If The operation is allowed, the function will execute it, otherwise, the TA sends back an unauthorized access error. Additionally, after the operation is successfully executed, calls the Logging Service new log entry, to register a new entry, regarding the executed operation, into the log.

- **TEE_Result log_read_data(char* data)** - Returns **TEE_SUCCESS** if the Adaptation Layer retrieves the logging stored.

  The function starts by requesting permission to execute the Logging Service's read log, by calling the Monitoring Service's filter operation. If The operation is allowed, the function will execute it, otherwise, the TA sends back an unauthorized access error.

- **TEE_Result trigger_monitoring(bool filter)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully sets the filter.

  Sets the value stored in the filter variable to the one stored in the argument filter.

- **TEE_Result get_attestation_proof(void* proof)** - Returns **TEE_SUCCESS** if the Adaptation Layer successfully retrieves and sends a proof to the Attestation Service.

  The function retrieves an already generated proof regarding this service and copies it into the supplied buffer, proof.

4.7 Attestation Service

The Attestation Service component follows an HTA approach, meaning some of its processing is done by the Trusted Application that runs on top of the TEE, while the remaining processing is done on the Normal World. We considered this the best approach since this component does not require any additional support from the Normal World.

4.7.1 Implementation

Considering this component constraints and characteristics, the Attestation Service was treated as a hybrid TA. The principal function of this component is to attest and generate proof stating that all secure components, at boot time, are indeed secure.

The procedure the Attestation Service we considered to be ideal was already previously described in Chapter 3, Section 3.7. The procedure, given the constraints and requirements of this component, consists in initially requesting a generated proof to each secure component. After receiving all proofs, the service would join all proofs into a single one and append to it the nonce initially passed as an argument, forming the digest. Finally, the digest would need to be signed, using an RSA keypair, resulting in the attestation proof. For simplicity purposes, the first time the service starts, a new RSA keypair is generated a stored in persistent memory so that it can be later used.

The Attestation service should then send back the newly generated proof, along with the essential parameters to reconstruct the public key in the Normal World side. These parameters would be needed, to verify the attestation proof and validate that the service sent valid proof, confirming that all components are secure.

4.7.2 API

In this brief section we specify the Attestation Service TA API provided to client applications residing in the NW. All functions have its implementation logic explained.

- **TEE_Result get_attestation_proof(int nonce, void* digest, void* proof, void* exp, void* mod)** - Returns **TEE_SUCCESS** if the proof is successfully sent to the Normal World.

The functions start by requesting from each secure component an authentication proof. If any of the components send an invalid proof, the function immediately stops operating and alerts the NW of an abnormality. After retrieving all proofs, it concatenates them, appends the nonce passed as arguments, and generates the attestation proof. This proof is then sent to the application that requested it through the supplied buffer, proof. Additionally, for the Normal World application to accept the attestation proof it needs to verify the proof, and to do that, the function also sends back: (i) the cumulative hash originated from the secure components appended with the received nonce; (ii) the exponent and modulus to generate the public key, needed for the decryption process.

4.8 TWallet Integration Support

The TWallet Service follows a Wrapper API approach, where the available functions made public by this service will call other functions that directly communicate with the APIs of each developed secure component. We considered this the best approach since the objective of this service is to create a channel that enables the communication between applications in the Normal World, and the secure components located in the Secure World.

4.8.1 Implementation

Given the default programming language used for Android applications development is different from the one used in Trusted Applications development, this service makes use of an integration library to link both languages. The integration library is called Java Native Integration (JNI) and is used to write Java native methods and embed the Java virtual machine into native applications [47]. This way, the android applications, typically developed in Java, can communicate with the running TAs, developed in C language.

4.8.2 API

In this brief section, we specify the TWallet Service API provided to Java Applications that wish to make use of our secure components. All functions have their implementation logic explained. Note that some functions refer to a created type called AccountInfo. This type, as the name suggests, consists of an object that contains the information regarding the current account in use, such as balance and set of transactions.

- **boolean attestateComponents(int nonce)** - Returns true if the Attestation Service successfully attests all components.

The function will pass an argument to the Secure World's Attestation Service, the generated attestation proof, and its public key. After receiving the proof, the functions must verify the proof signature and confirm if the nonce initially sent is part of its body.

- **boolean trigger(boolean filter)** - Returns true if the Monitoring Service successfully changes the monitoring filtering approach.

  Sets the Monitoring Service filter to allow access, or not, to the secure components' methods.

- **boolean writeData(String id, AccountInfo info)** - Returns true if the Secure Storage successfully writes a new data into memory.

  The function will initially convert the information stored in the AccountInfo object into a string, and then send it to the Secure Storage. The information will be stored, and consequently, the function will return true. In case there is already an entry with the same id, the function returns false.

- **AccountInfo readData(String id)** - Returns an object AccountInfo if the object identified by its id is successfully retrieved.

  The function will search through the objects registered in the Secure World's Secure Storage for an entry with the same id. In case a match is found, the stored object is retrieved and it will be converted into an AccountInfo object. If nothing is found regardless, the function will still return an AccountInfo object, but empty.

- **boolean deleteData(String id)** - Returns true if the Secure Storage successfully deletes the object identified by its id.

  The function will search through the objects registered in the Secure World's Secure Storage for an entry with the same id. If the object is found, it will be permanently deleted, and it will return true. If an object is not found the function will simply return false.

- **boolean storeCredentials(String id, String cId, String cPwd)** - Returns true if the Authentication Service successfully stores a new credential into memory.

  The function will create try to create a new entry in the Authentication Service, using the id and credentials (cId, cPwd). In case an object identified by the same id already exists, the function will cancel the operation and return false.

- **String[] loadCredentials(String id)** - Returns an array of strings with the credentials if the Authentication Service successfully finds an object with the respective id.

  The function will search through the objects registered in the Secure World's Authentication Service for an entry with the same id. If a match is found, the object

containing the credentials is retrieved. If nothing is found, the function returns false.

- **boolean deleteCredentials(String id)** - Returns true if the Authentication Service successfully deletes a set of credentials identified by its id.

  The function will search through the objects registered in the Authentication Service for an entry with the same id. If an object is found, it will be permanently deleted, and the function returns true. If no match is found, the function returns false.

- **String getLogging()** - Returns a string that contains the event log built so far by the Logging Service.

  The functions will always retrieve the event log object, either a log with entries or an empty one.

4.9 Summary

In this chapter, given the already discussed design requirements and constraints, we presented a prototype of our solution and explained its system model and architecture implementations. Initially, we presented our chosen development environment, describing the TEE platform, as well as the steps we took to set it up. In sequence, we addressed the implementation considerations of the developed components that are part of our architecture. These components are the Secure Storage Component, the Authentication Service, the Logging Service, the Monitoring Service, the TEE Adaptation and Isolation Layer, the Attestation Service, and the TWallet Framework.

The Secure Storage Component is designed as a simple key-value store where each key is associated with a single value. The underlying data structure is used as a hashtable, in which the key would immediately point to the assigned value. To persistently store the information managed by this component, the TA made use of the OP-TEE's Secure Storage mechanism.

Next, we have the Authentication Service, and exactly like the Secure Storage, this component is designed as a key-value store. The main difference between these components consists in the underlying information stored. Whereas the Secure Storage registered in our scenario the wallet's information, the Authentication Service is focused on storing the access credentials of registered wallets.

Then we present the Logging Service, which was designed to generate an event log of all operations used by the secure components throughout its execution time. The underlying structure consisted of a simple list of fixed entries, wherein in case the limit was reached, the oldest entries would be removed. To persist the log throughout the TA sessions, the Logging Service used the OP-TEE Secure Storage mechanism to store the log information "in-disk".

The Monitoring Service is the next to be explained. This component principal function is to intercept the incoming requests from NW and validate if those requests could be called without posing any threat to the secure components. The Monitoring Service underlying structure consisted is a simple fixed-size list, where each entry would describe the information of each secure component. This information would then be used during the operation filtering.

Next, we present the TEE Adaptation Layer, which implemented a simple middleware between the NW applications and the secure components. This application would intercept the requests, verify if they could be executed, and if needed register its activity in the event log. To correctly execute its functions the Adaptation Layer made use of the Monitoring Service and Logging Service API functions.

Then we have the Attestation service, whose design consisted in generating an authentication proof that would confirm that the secure components, at attestation time, would be secure. The component implementation consisted in requesting a proof from each of the secure components, joining them, appending a nonce obtained from the NW application, and signing everything. By sending back this proof, along with the verification key, the requested application can determine if the components at the moment can be considered secure or not.

Finally, the TWallet provides the implementation of a Wrapper API architecture. This application would be responsible to make the services provided by the secure components available to any application that desires to use it. The implementation made use of the JNI integration library to intertwine the java language with the C language. In our context, this was essential because, typically, android applications are developed in Java, while the applications developed to communicate with TAs, are written in C.

The next chapter is directed to the experimental evaluation phase, where we describe the performed tests and our analysis, given the obtained results.

5

Experimental Evaluation

To validate the designed solution and related prototype, and to study the performance of supported operations and implemented components, we performed a series of different experiments using an experimental test bench environment, while considering the development environment described in Chapter 4. We also observed some practical indicators for software engineering metrics, related to resource allocation throughout the execution of prototype components. In this chapter, we intend to present the conducted experiments and their respective results. In Section 5.1 we explain the evaluation environment and all considered evaluations. In the following sections, we present the experimental evaluations done to analyze our developed solution and discuss the obtained results and related conclusions. Section 5.2 discusses the TWallet System overall performance, Section 5.3 discusses Profiling metrics, such as storage cost, system and application boot times. Section 5.4 discusses the spent system resource to operate on an app without our version vs. one with our solution, and finally, Section 5.5 evaluates the Attestation Process performance. To conclude, in Section 5.6 we summarize the main observation of all evaluations and the more relevant validation arguments of our work.

5.1 Testbench and Evaluation Methodology

5.1.1 Testbench Environment

For the Evaluation testbench, we used the same environment that was used during the development phase. The environment is summarized in the following table, describing the complete set of characteristics of all components, either from a software and hardware perspective.

Table 5.1: Testbench Environment Characteristics

Tested board [3]	Hikey 960, Kirin 960, Cortex-A72 2.3GHz & Cortex-A53 1.8GHz (ARMv8) 64-bit SoC
OP-TEE [69]	3.12.0
Android Version [18]	9.0

Furthermore, some of the operations we intend to evaluate require some communication with an Ethereum Blockchain. To enable this desired communication, we make use of Hikey WiFi adapter, with support for 2.4 and 5GHz dual-band with two antennas, and a network with 50 Mbps of bandwidth. The process between our prototype and the blockchain can be illustrated in figure 5.1.

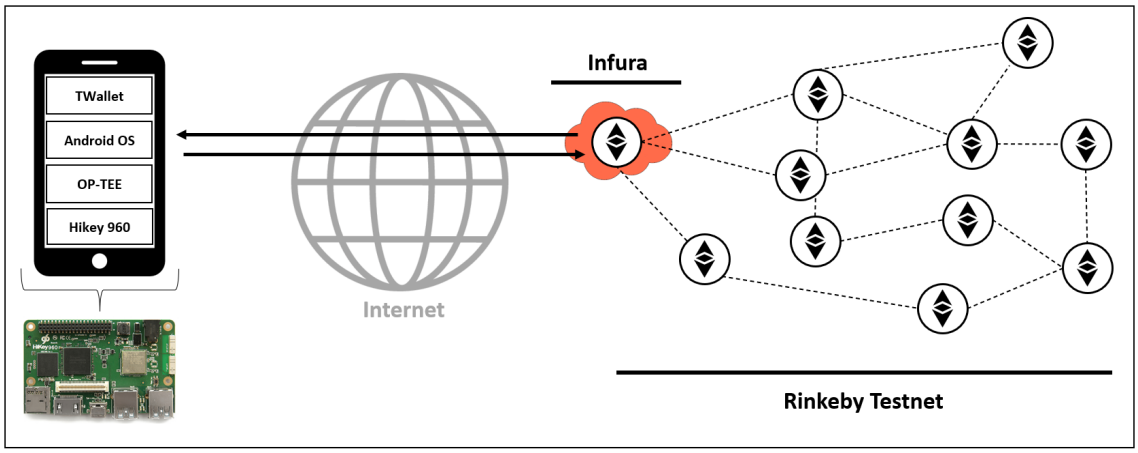


Figure 5.1: Communication between Prototype and Ethereum Blockchain

Since in this thesis we are not able to use the real Ethereum Blockchain to manage Ether cryptocurrencies, we decided to use the Rinkeby testnet [53] to fully perform the operations that demanded communication with the Blockchain. Rinkeby is an Ethereum test network that allows for blockchain development testing before deployment on the main Ethereum network [77]. To connect to this testnet we used a node service provider called Infura [12] that managed the node we used to perform our requests to the network. These requests were done using a remote procedure call protocol over HTTPS, named JSON-RPC [28].

5.1.2 Evaluation Methodology

For the validation of the proposed solution, we conducted a sequence of experimental evaluations covering the issues: (i) solution performance and overhead, including comparative evaluation of operations performed on a normal cryptocurrency wallet and on a wallet that uses our solution, as also performance measurement of our components operations; (ii) profiling of runtime components, including the analysis of storage cost and observation of the setup latency; (iii) resource allocation analysis, and (iv) performance of the attestation Protocol and its impact when using different ciphersuites and cyptographic key sizes. This allows us to gather the required data to perform our planned experimental evaluation of our prototype as described in Chapter 1.

The comparative evaluation of operations between a normal cryptocurrency wallet and a wallet backed by the TWallet was performed using a benchmark android application we developed that performs the normal operations and the operations that include

the calls to our secure components. To measure the time spent to do a complete operation we used the Java function `System.getTimeMillis()`, however, some of these tests aimed to measure the performance of the secure components commands and therefore to register their times we instead used the OP-TEE function `TEE_GetREETime()`. Each performed test was executed 70 times since we considered this value to be large enough to reduce the standard deviation of the analyzed operations to an acceptable margin of error.

The evaluations regarding profiling of runtime components were done by measuring the System and Application Boot times, as registering the additional storage cost. For the time registration tests, we decided to analyze the output of the Boot process through a UART component, and register its presented value when the boot reached its finish state. To obtain the storage cost we measured the storage space required by developed Apps and TWallet Trusted Components, as well as, the storage required by OP-TEE. For tests that measure time, we used 10 a sample size for repetitions, while for the storage cost tests we used the obtained results as these values would not change.

Regarding the calculation of extra allocated resources, we used the same developed android app during the comparative analysis to run the tests but, the measurement of the resources was done through the Android Studio Profiler tool [6]. For each of the performed tests, we did 10 repetitions.

In the last experiments about the attestation protocol performance, we used the developed benchmark to perform the tests. Due to the nature of some tests, our objective was to register the time the attestation protocol spent only inside the Secure World and because of that, we used the OP-TEE function `TEE_GetREETime()` to provide a more precise value aligned with the Rich OS time system. Each test was executed 70 times following the same principles that were described before on the comparative analysis tests.

5.1.3 Summary of Evaluation Metrics

Based on the above evaluation methodology aspects and experimental observations to assess and validate the developed TWallet prototype, the following table 5.2 summarizes the metrics observed in each experiment.

Table 5.2: Evaluation metrics

TWallet Comparative Performance	
Observation	Observation Criteria
Wallet Operations Performance	Latency (ms) Throughput (ops/s)
Secure Components' Operations	Latency (ms) Throughput (ops/s)
Components Internal Operations	Latency (ms) Throughput (ops/s)
TWallet Profilling	
Observation	Observation Criteria
Solution Required Space	Space (KBytes)
System Boot	Time (s)
Application Boot with TWallet	Time (s)
TWallet Allocated Resources	
Observation	Observation Criteria
CPU usage	CPU load (%)
Memory usage	Space (MBytes)
Network usage	Latency (ms)
Attestation Protocol Performance	
Observation	Observation Criteria
Attestation Protocol (different ciphersuites)	Latency (ms)
Attestation Protocol (different ciphersuites) incl. Key Generation	Latency (ms)
Key Generation Process	Latency (ms)

In the following sections, our experimental observations are reported and discussed.

5.2 TWallet System Performance

We conducted the Twallet System evaluation by starting with measuring the operations that can be performed in the user device by the end-user, where some of them need to communicate with an entity external to the user device. The measurement considered an application without our solution, and the same application but with our solution included. After that, we decided to isolate the functions called by our secure components that are part of the end-user available operations and evaluate its performance.

Regarding the operations that demand some communication with an external entity, which given this thesis context the referred entity is the Ethereum Blockchain, we consider that all communication establishment procedures were already done before the operations performance measurement.

5.2.1 Operations Performance

To measure the overall performance we decided to evaluate the operations that can be performed on the user device by the end-user. The objective is to obtain a first general analysis of the impact on the user device by making use of our solution. The following plots, illustrated in figure 5.2 presents a comparison of some cryptocurrency wallet operations, performed by trusted and untrusted applications. Considering the registered results in figure 5.2(a) and 5.2(d), we think that a presentation regarding those operations' throughput would not add any particular information to this evaluation, since most of the measured operations have an execution time larger than 1 second, and therefore the produced throughput would be inferior to 1. However, the remaining operation had their throughput calculated and can be observed in tables table 5.3 and 5.4.

Table 5.3: Performance of Normal Wallet Operations

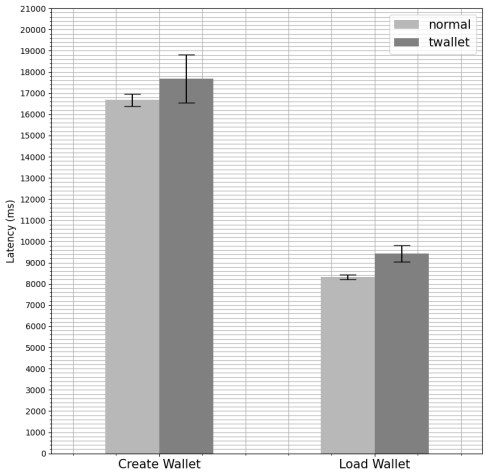
Operation	Throughput (ops/s)
Balance	7
Delete Credentials	621
History of Transactions	4

Table 5.4: Performance of TWallet Operations

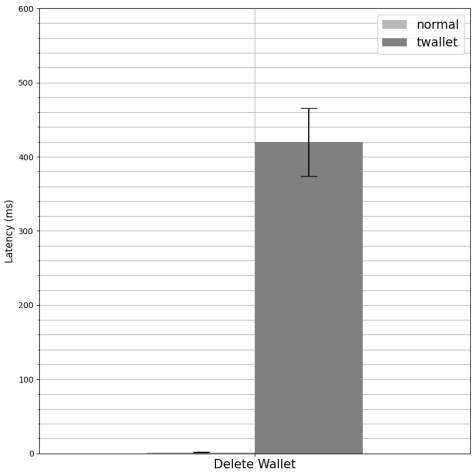
Operation	Throughput (ops/s)
Balance	2
Delete Credentials	2
History of Transactions	1

Interpreting the observed results, we can easily deduce that the additional instructions done by our solution are not expensive enough to dominate the operation execution time. However, and as expected, there is an inherent overhead caused by the extra security provided to each of the evaluated operations. The origin of this overhead is mainly due to:

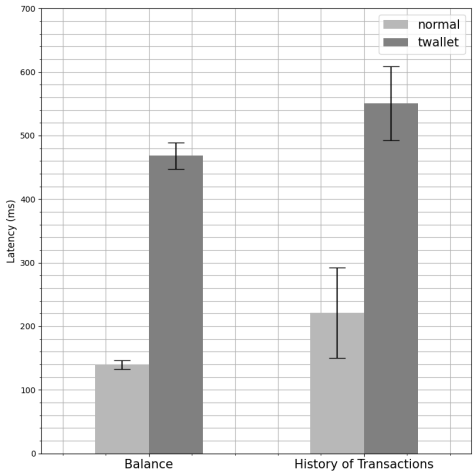
- Context Switch - The Context Switching mechanism of ARM TrustZone provides isolation guarantees between the trusted and untrusted execution environments, and the operations implemented on the TA itself when compared to the untrusted application.
- Trusted Application Switching - Most of our solution operations are subdivided into other distinct applications - Monitoring Service, Logging Service and Component itself, each with their respective Trusted Applications in the SW. This is because the Normal World does not communicate directly with the TA it intends to communicate, but with the TEE Adaptation Layer, as explained in Chapter 3 Section 3.6. After the TEE Adaptation Layer receives the incoming request, it will sequentially send a request to each of the involved Trusted Applications so that



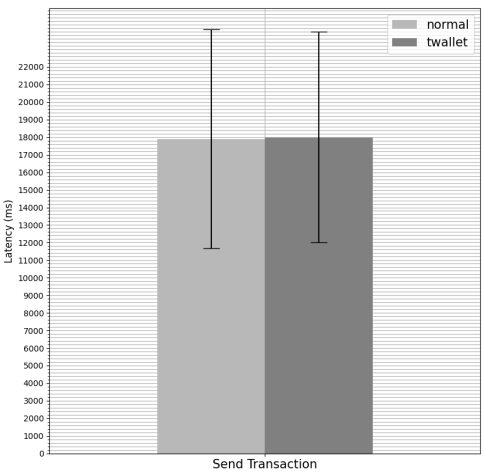
(a) Operations: Create Wallet, Load Wallet



(b) Operations: Delete Wallet



(c) Operations: Balance, History of Transactions



(d) Operations: Send Transaction

Figure 5.2: Wallet Operations Latency Comparison

the request can be attended. When a TA calls another function from another TA, the OP-TEE changes its execution context from one application to another, similar to when a user application performs a system call that triggers an interrupt. The interruption consequently changes the current context of execution between the user-space and the privileged kernel-space, which handles the interruption. All this process involves an additional cost of switching between Trusted Applications that must be taken into account when measuring its overall performance.

Without the existence of these factors, we consider that the execution time of the operations of the Trusted Application would be identical to its untrusted counterpart.

An analysis of each operation's performance is detailed below:

- The **Delete** operation is the fastest of them all. The operation consists in deleting a wallet file and all its associated information stored in the secure world. The wallet file deletion, considering the app without our solution, simply deletes the wallet file, and by observing the results we can see the operation itself is almost ignored. When taking into account an app with our solution we can see that there is an increase in the execution time due to the 2 delete operations that must be done in the Secure World. This is a significant decrease in the operation cost considering the other operations since all others perform write operations over the Authentication Service or Secure Storage, which have a bigger impact than a deletion command in comparison to the total operation cost.

- The **Balance** operation is the second fastest of them all. The operation requests to the Ethereum Blockchain the balance associated with the wallet that performed the request, and then stores the result into the secure world. The operation normal process consists in retrieving the value from the Blockchain, which made the operation the second fastest. By adding our solution, the operation must additionally perform a write operation to the Secure Storage, increasing by more than 2 times the operation normal execution time.

- The **Send Transaction** is the most expensive operation. This operation consists in sending a transaction to the Ethereum Blockchain (testnet), and until the Blockchain confirms that the transaction was concluded and will be properly processed in the future, the operation is interrupted while awaiting a response. After receiving confirmation the operation was concluded, the transaction information is written into the Secure Storage. Since the time spent to confirm that the transaction was processed can greatly vary due to the time needed to successfully close a block in the Blockchain, the operation completion time is directly influenced which is expressed in the standard deviation values we can see in the plot.

- We consider that the **Balance** and **History of Transactions** operations could be similar regarding its execution time. However, due to the already explained History

of Transactions communication segment, the values ended up being unstable. The History of Transactions performs a request to the Ethereum Blockchain to obtain the history of past transactions done by the requested wallet, and after receiving the response stores its results in the Secure World. Similar to the Balance operation, the retrieved response is stored in the Secure Storage, which causes the operation to take more than 2 times a normal execution would take.

- The **Create Credentials**, **Load Credentials** and **Send Transaction** are the most expensive operations even without the use of our solution. All of them require expensive computations to properly perform their functions, and by adding our solution to the function processing, its execution time is slightly increased due to the write operations requested to the Authentication and Secure Storage Services.

Standard Deviation Analysis. Observing the registered standard deviation values we can notice that the operation Send Transaction presents an abnormal high standard deviation value. The Send Transaction is a call done to the library Web3J [43] and consists in sending a transaction synchronously to the Ethereum Blockchain.

The operation sends the transaction with the specified information to the Blockchain environment (e.g., rinkeby test network), with the confirmation that it was inserted in a block. Observing the parameters in the Blockchain operation [21] it is expectable that the mean completion time of the Proof of Stake to close the blocks is around 13 to 14 seconds. However, there are cases that given the number of transactions to close, as also the current gas value, the completion time can largely increase to 30 or even 40 seconds. This explains a considerable variation as observed in the evaluated standard deviation illustrated in the plot.

Besides the operation Send Transaction, we can see that the History of Transactions also displays a high standard deviation value. The History of Transactions consists in retrieving from the Ethereum Blockchain a list with the history of past transactions the requested account already did. This operation can be divided into two segments, the communication segment, responsible for sending the request and retrieving the response, and the processing segment, which processes the response information and stores it in a data structure so that, in case the operation was to be done in a real application, it could be presented to the user.

To further understand the standard deviation abnormal values, we decided to perform an evaluation in which we re-run the operation, but this time considering only the communication segment. The obtained results, as also the previously registered values, are all presented in table 5.5.

As we can see by the data presented in the table, the abnormal standard deviation value seems to persist while only considering the communication segment. The reason why this happens might be due to the endpoint where the request is being done since this is the only operation we tested that sends a request to a different endpoint. Some of the other operations, although they also possess a communication segment, they seem to

Table 5.5: History of Transaction Segment Comparison

Segment	Latency (ms)	Throughput (ops/s)	Standard Deviation (%)
Complete Operation	220.96	4.5	32.20
Communication Segment Only	198.86	5.0	38.75

present a standard deviation significantly lower. These operations are Balance and Send Transaction.

With the data obtained by performing these two evaluations, we can conclude that the communication segment is the dominant part of the complete operation since the registered standard deviation value seems to pass on to the entire operation. Therefore, the presented standard deviation value of the History of Transaction is caused by the communication segment, which is out of this thesis scope, and because of that, we decided to ignore it.

5.2.2 Secure Components Performance

After evaluating the available operations performed by the end-user, we decided to measure the overall performance of some operations executed by our secure components. These sub-operations are called when the operations measured in Section 5.2.1 are called. The sub-operations belong to the Authentication Service and Secure Storage. The Authentication Service manages the creation, load and deletion of user credentials needed for the wallet app, while the Secure Storage manages some information regarding the wallet account, such as balance and log of transactions.

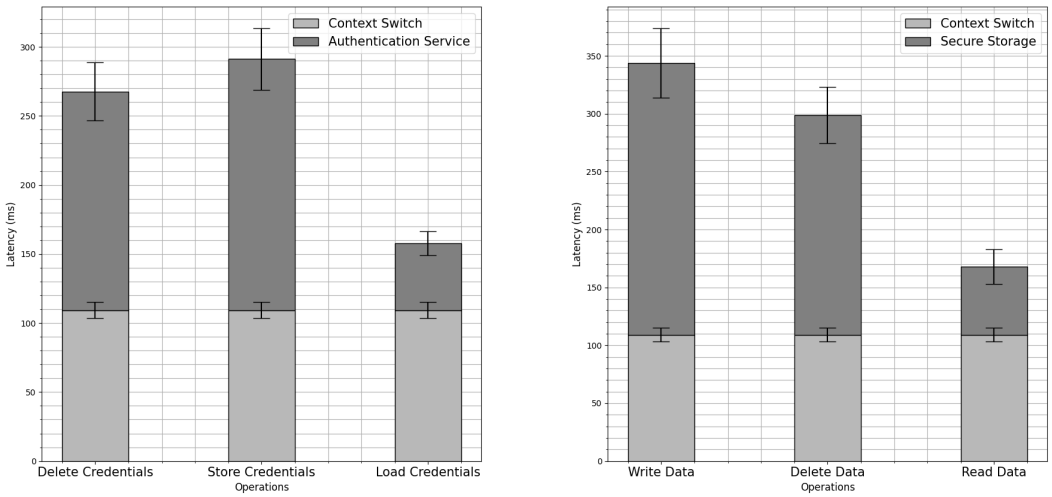
Figures 5.3(a) and 5.3(b) illustrate a chart that presents the latency of sub-operations, performed by Authentication Service and Secure Storage, respectively. Tables 5.6 and 5.7 represent the operations per second performed by the same Trusted Applications.

Table 5.6: Performance of Authentication Service Operations

Operations	Throughput (ops/s)
Store Credentials	3
Load Credentials	6
Delete Credentials	3

Table 5.7: Performance of Secure Storage Operations

Operation	Throughput (ops/s)
Write Data	2
Read Data	5
Delete Data	3



(a) (b)

Figure 5.3: Secure Components Main Operations Latency

As observed in the figure, the execution time of these operations is significantly small, producing a good throughput of operations performed per second. Despite the listed execution time, we can state that a certain percentage does not belong to the processing of the request itself. In these operations, a part of the execution belongs to the Context Switch and Trusted Application Switching, which is responsible for switching executions between the Normal World and the Secure World, and switching execution context between TAs, respectively.

5.2.3 Internal components Performance

Despite the time presented by the execution of the operations of the secure components, as we mentioned in Chapter 3 Section 3.1, these operations are a composite of operations performed by different secure components, all connected through the TEE Adaptation Layer. The TEE Adaptation Layer is responsible for intercepting the incoming requests from the Normal World, and if needed, performing a set of auxiliary operations before redirecting the request to the secure component that will attend it. The auxiliary operations consist in: verifying if the operation can be performed from a Normal World perspective, by the Monitoring Service; and registering the operation activity in a log that contains all operations performed by the secure components, in the Logging Service.

Both these operations performed by these Secure components were measured and its latency and number of operations per second are respectively presented in figure 5.4, and table 5.8.

As we previously mentioned, the measured run time is divided across different secure

Table 5.8: Performance of Internal Operations

Operation	Throughput (ops/s)
Log new Entry	13
Filter Operation	34

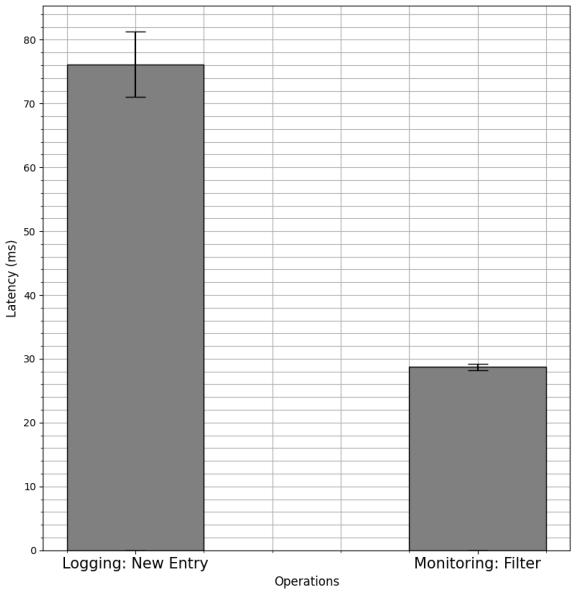


Figure 5.4: TWallet Internal Component Operations Latency

components, with these operations being part of them. However, as we can observe the registered times we can deduce that, as expected, these operations are dominated by the main secure component that is responsible for attending the request.

With all observations registered, we can conclude that as expected, there is an overhead caused by the provided extra security from our solution to the operations performed in a cryptocurrency wallet. However, this overhead seems to not dominate the entire execution of the operation and taking into account its percentage considering the entire execution time, and the additional security properties, we think of this to be beneficial to our solution.

5.3 Profiling

To successfully monitor and evaluate the profiling indicators regarding our system solution, we decided to measure two essential aspects: first, we measure the boot time of the system and the tested applications, to determine the relationship between the usage of our solution, and the overhead time spent to fully boot the user device OS and a normal application; second, we intend to determine the cost of using our solution, by analyzing

the allocated space and comparing it with a base solution, without our system additional security guarantees.

5.3.1 Boot Execution Time

To analyze the time our solution's system setup takes to boot, in an incremental way, we decided to measure the system boot times. To achieve it, we examined the logs produced by OP-TEE and computed the timestamp between the moment the Operating System starts booting up and the moment all background processes are concluded, and both Trusted OS and Rich OS are ready to execute their applications. Our thought process consisted of: firstly, registering the TEE Boot time, followed by the TEE + Rich OS Boot time; and then measuring the entire Boot process from the TEE boot, until the boot of our secure components. For this evaluation the number of repetitions was significantly small, being only 10. The table 5.9 describes each of the processes registered times.

Table 5.9: System Boot Times

System	Time (s)
OP-TEE (TEE)	3.66
OP-TEE + AOSP (TEE + Rich OS)	17.25
OP-TEE + AOSP + Secure Components	18.26

By observing the results, we can conclude that the boot process time spent is mainly due to the Rich OS boot. Seeing the boot full process (OP-TEE + AOSP + Secure Component) and comparing it with the OP-TEE + AOSP times, the secure components boot time is not large enough to dominate the system setup, which results in the fact that the OP-TEE + AOSP boot time is the part that takes more time to boot, whereas the Secure Components boot time is the smallest. Additionally, since knowing the OP-TEE boot times and comparing with the OP-TEE + AOSP times, we see that the percentage corresponding to the OP-TEE boot is smaller than 30% of the total OSes boot time, meaning that the AOSP boot is the big dominant during the entire boot process.

Because the Rich OS is the main process that defines the system boot time, we can conclude that our solution overall boot would not largely affect the normal boot process of a mobile device. Since the trade-off between the boot process additional time and, the enforcement of security properties that can be used by some applications is beneficial, we think of this as a point in favour of our solution.

5.3.2 Application Boot

Now aiming to evaluate the boot of our tested applications, we decided to analyze their boot times (or boot latency). To measure the boot time, we decided to inspect the logs produced by OP-TEE and computed the timestamp between the moment the user starts the application, and the moment the application finishes executing all its background processes and displays a window requesting for user input to proceed. In this experiment,

the number of repetitions was only 10, and the obtained results can be consulted in table 5.10.

Table 5.10: Application Boot Times

System	Time (s)
Base Application	0.716
Application + TWallet	0.881

Analyzing these results, we can see that the boot time of a normal application is naturally faster than an application that uses our solution. The overhead cost considering the tested applications is slightly more than 20% boot time of the normal application. The differences between using or not our solution during the application boot process can be summarized in checking if the Authentication Service contains some stored credentials, which will determine if the application must or not request a new set of credentials from the user. Additional to the Authentication Service request, another command is issued while using our solution during the boot: the attestation process execution. Although the cost for executing this process is not included in these recorded times, the performance analysis of the attestation process will be done in section 4.7.

5.3.3 Storage Cost

To evaluate the cost of a normal cryptocurrency wallet against a wallet incorporated with our solution, we decided to consider the storage cost, as the allocated space measured as soon as the apps were installed as system apps in a user device. With that in mind, table 5.11 presents the registered values.

Besides comparing between the occupied space of a wallet with and without our solution, from the Normal World, we desire to measure the allocated space of our solution. This includes not only the app installed in the Rich OS in the Normal World but also the space needed for our secure components present in the Secure World, to properly execute its functions.

Table 5.11: Storage Cost Values

System	Cost (kB)
Base Application	4.13
TWallet	360
Secure Components	350
OP-TEE	938
Total	1652

Analyzing the table we can observe that as expected, our solution, either with or without considering only the TWallet system storage cost, is bigger than the base solution. In the case where we included the OP-TEE occupied space, our solution overall prove to be largely superior in storage cost terms, when compared to the base application. However,

our decision to include the TEE in our allocated space measure calculation is not because it is part of our solution, but because without the TEE our solution would not execute.

Regarding the remaining cases, we considered the major cause for the storage increase to be the JNI library. The JNI library intertwines the processing of the Android base app, developed in Java, and the C functions used to communicate with the secure components in the Secure World.

There is an increased allocated storage space when comparing the TWallet-protected App with the Non-Protected App. However, we can conclude that we can have gains in security, with a reasonable aggravation on the required resources, regarding typical mobile Apps and the resources of mobile devices.

5.4 System Resources

To properly observe the resource allocation behaviour when a system uses our solution, we conducted the same performance operations evaluated in Section 5.2.1, since these operations are directly used by the end-user. The resource allocation tests were performed considering the CPU utilization, Memory Cost, and Network. The number of repetitions done in these observations was only 10.

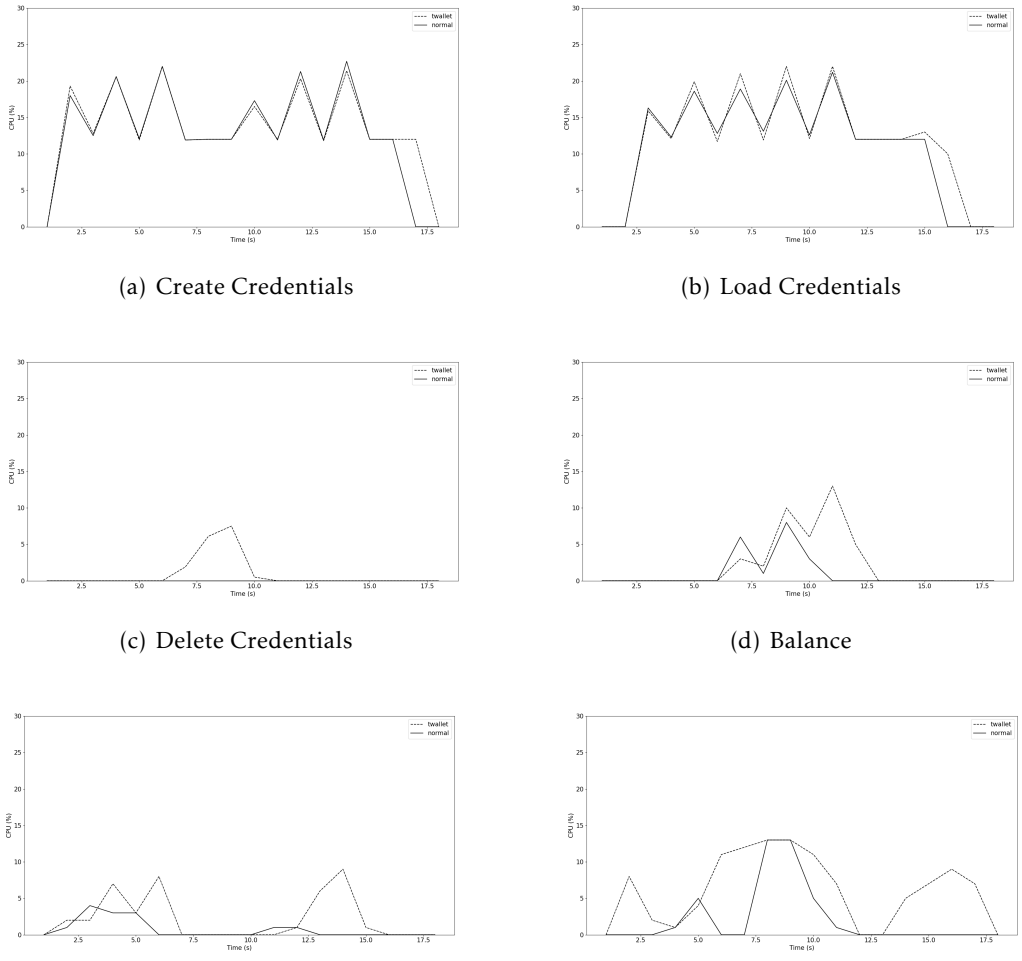
5.4.1 CPU Utilization

To measure the CPU utilization occupied by the operations previously observed, we decided to re-run those operations, while using an app without our solution, and compare its obtained results with the same application, but now using the TWallet System. The registered values can be visualized in figure 5.5.

By observing the obtained results, we can see that Create Credentials, Load Credentials and Balance registered a clear increase of the CPU resources, on the horizontal plane. This means the operations did not require more CPU resources than the already used but did spend more time with the resources allocated. Such behaviour is due to the additional execution of the requests to our solution components, as they demand the use of CPU utilization because of write and read operations.

The delete operation displayed the greatest increase in the CPU spent resources. This operation originally did not require any CPU resources, since it only consisted in deleting the wallet information stored in a local file. However, by using our solution, the operation did a request to our secure components, namely a delete request, which requires allocating some CPU utilization to delete the information from memory.

The Send Transaction and History of Transactions, as we already mentioned in Section 5.2.1, expressed large variations over the registered values, mainly due to the communication segment. This segment influence seems to also be expressed on the CPU resources as the registered values do not seem to have the same pattern, as the one observed in the Create Credentials, for example. Despite the disparity between registered values,



(a) Create Credentials

(b) Load Credentials

(c) Delete Credentials

(d) Balance

(e) Send Transaction

(f) History of Transactions

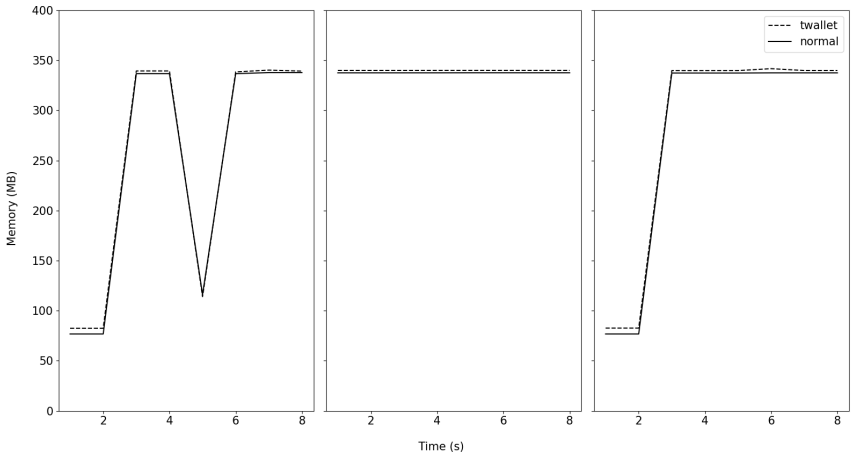
Figure 5.5: CPU Utilization Comparison. The light green represents the wallet with the TWallet System, and the other green the wallet without our solution.

we can see a point in common in both operations, which is the extent of the operation execution time and an increase of the CPU allocated resources at its end. This is due to the requests done to our secure components located in SW, as both commands request write operations to the Secure Storage component.

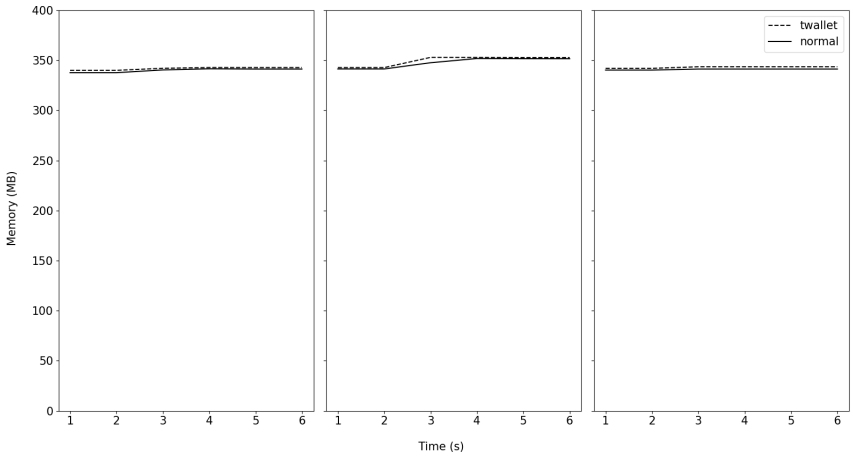
5.4.2 Memory Cost

While aiming to evaluate the memory cost of the performed operations, on a cryptocurrency wallet without our solution, and on a similar application but with our solution incorporated, we analyzed the same operations observed in the previous Section. The registered results are expressed in figure 5.6.

Through the registered results, we can notice that both operations, Create Credentials and Load Credentials seem to greatly increase their memory cost. This is due to the



(a)



(b)

Figure 5.6: Memory Cost Comparison

library used by the tested application, Web3J [43]. The operations when called will use the respective functions from this library to create a new wallet or load existing wallet information into memory, which requires generating and loading a significant amount of information, producing a large increase in memory cost, which we can observe in the plots. As we can see, using our solution does not invoke an increase greater than the one caused by the library functions, therefore the memory used by the operations of our solution are mitigated by a large amount of memory spent on the library functions.

Regarding the other evaluated operations, the amount of memory spent is uniform and unaltered. As we already explained, a large amount of memory used is due to the Web3J functions and this reason goes across operations like Balance, Send Transaction, and History of Transactions because these operations are operations that can only be done when wallet information is loaded on memory.

Considering our statement regarding the large increase in memory cost being caused by the library functions, the Delete Credentials would technically perform the inverse phenomenon. However, the reason why this is not expressed in the plot is since Web3J does not have a function responsible for deleting wallet information, and therefore the library cannot free the information loaded on memory.

With these observations, we can deduce that our solution does not cause a significant increase over the memory cost, since the dominant segment belongs to the library Web3J allocated variables. Therefore, considering the tested applications, we can argument in favour of our solution, since it provides a relevant security enforcement to critical operations with an acceptable increase in the memory requirements, particularly considering the typical memory resources we find in commonly used mobile devices.

5.4.3 Network

To analyze the network connectivity, we observed the performance of the previously mentioned operations, namely Balance, Send Transaction and History of Transactions. For this observation we used the Android Studio Profiler tool to obtain the latency of each operation and related data exchange between the local device and the blockchain, comparing the values for the application with and without the support of the TWallet protection. The obtained results are expressed in figure 5.7.

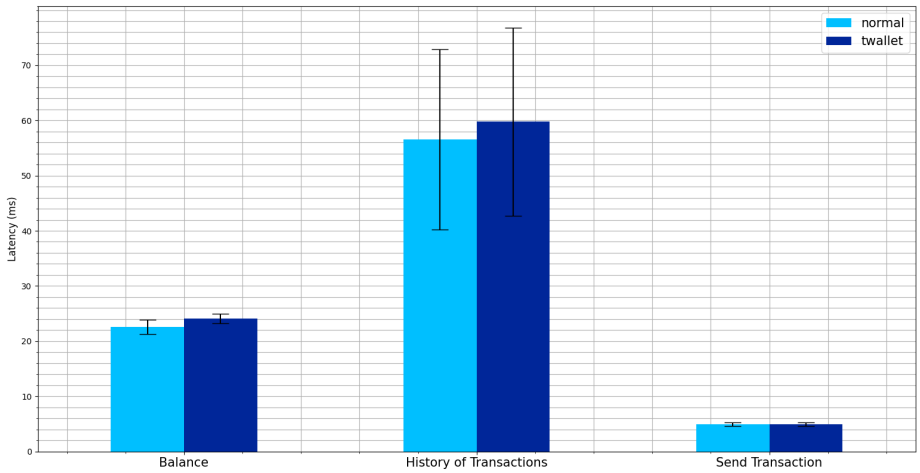


Figure 5.7: Network Resources Comparison

In this plot we must notice that for the case of the Send Transaction, the observed latency doesn't include the finalization of the transaction in the Blockchain e.g., the latency in the Blockchain side to achieve the finality of the block in which the transaction will be included, leading to the registered values.

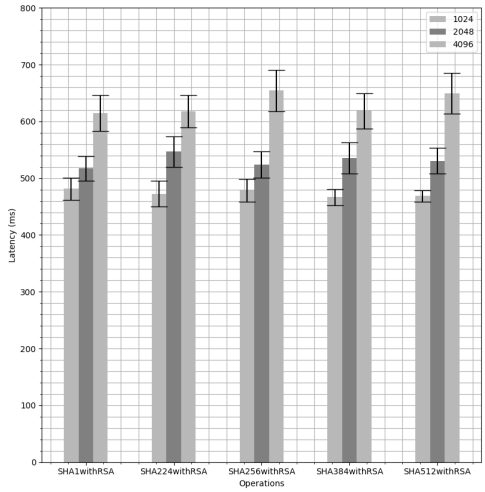
As we can observe, the obtained values are very similar, even when considering the two different scenarios. This is due to the fact that our solution does not enforce any additional security properties over the communications with any external entity. With all results considered, we conclude that the communication segment of each operation, as its performance, is out-of-scope of our solution and does not suffer any change.

5.5 Attestation Service

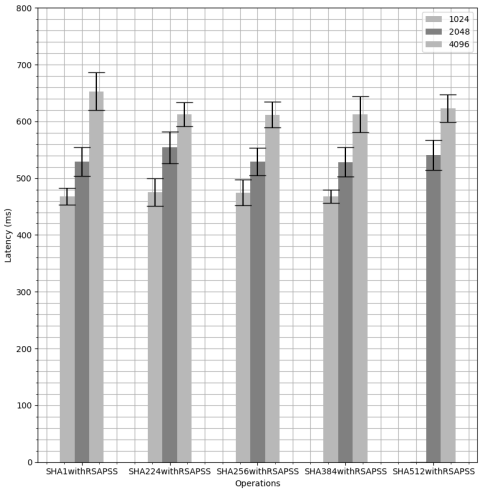
To evaluate the Attestation Service performance, we decided to benchmark its attestation protocol when used in a client application context. Regarding the attestation process itself, we considered it to be out-of-scope, the generation of the hash proof by each secure component as that process is performed during the system boot, and the key generation process since it only happens the first time the protocol is executed.

To measure the attestation process performance, we did variations to the ciphersuites used during the signing process, used to authenticate the proof sent to the client application in the Normal World. The ciphersuite variations take into account different algorithms, RSA and ECDSA, different padding strategies, PKCS and PSS, different hash functions related to the signing process, and different key sizes. All these permutations were properly measured and evaluated, and their results can be presented in figure 5.8.

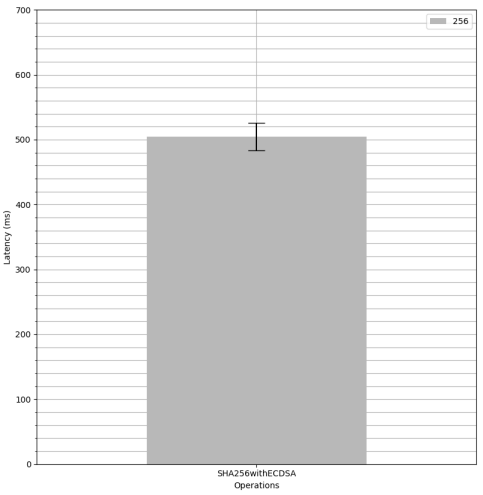
An initial observation is that some ciphersuites, namely the SHA512 with RSA and



(a) RSA Signature with PKCS1.5 Padding with 1024, 2048 and 4096 bit key length



(b) RSA Signature with PSS Padding with 1024, 2048 and 4096 bit key length



(c) ECDSA Signature with 256 bit key length, using NIST curve P-256

Figure 5.8: Attestation Process Latency Comparison

padding PSS, have only two registered times because of the used key size. The reason for this phenomenon results from a condition of PSS padding that states that the key/modulus size must be at least the sum of the salt and hash sizes plus 9 bits [66, 44]. This causes the RSA ciphersuites that use PSS padding to have a minimum key size of 1024, 1024, and 2048 for SHA256, 384 and 512 respectively. The other studied hash functions must even use smaller key sizes, but since we didn't consider them during this evaluation they were not analyzed.

Visualizing the obtained results, we notice that despite using different ciphersuites, the presented results are in the same interval. We can deduce that the attestation process is independent of any ciphersuite, if and only if, the key generation or any other additional procedure related to the used ciphersuite are not executed during the protocol.

We can however see that by using different key sizes, we obtain different time intervals. This is mostly due to the signing process itself, as by using bigger key sizes the sign process time gradually increases. Another reason for the time increase between the used key sizes can be due to the key load operation from storage. All used keys are stored on disk and to use them the protocol needs to load them into memory. Since the object needs to load, the bigger the object the more time will consume to completely retrieve it and load in memory, therefore the load key process does also influence the protocol execution time.

During these measurements, we considered out-of-scope the key generation process. However, as previously mentioned, this process also occurs during the attestation protocol only once, on the first time the protocol is executed. Considering that first time, we re-executed the tests and included the key generation process in the protocol procedure. The obtained results can be visualized in figure 5.9.

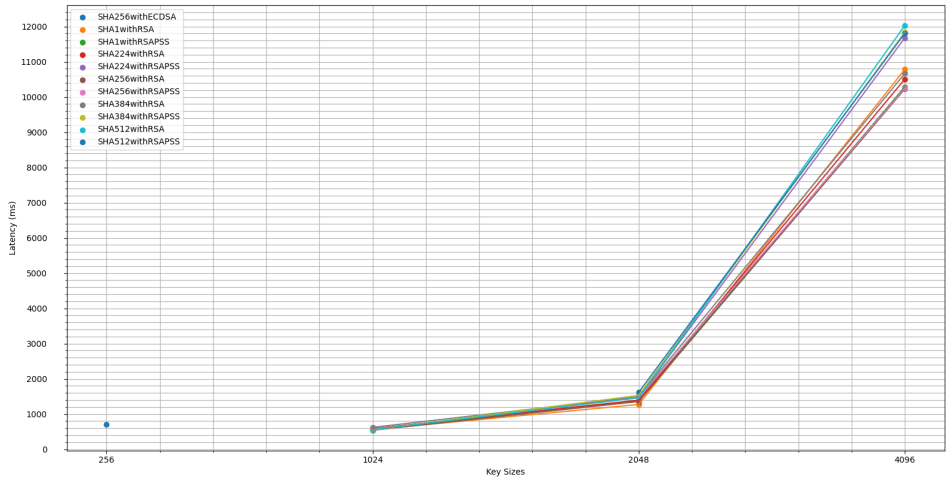


Figure 5.9: Latency of Attestation Process including key generation process

As a first analysis, we can observe that there is a direct relationship between the time

spent to generate and send the proof to the Normal World, and the generated key size used to sign the proof. This is because a larger key size needs more time to be generated so that it can produce a stronger key pair than another pair that has a smaller key size.

Despite the presented values, we noticed that throughout this experimental evaluation, the standard deviation percentage was steadily increasing as we increased the key size. The registered standard deviation values are presented below, in table 5.12.

Table 5.12: Attestation Process, Standard Deviation per Ciphersuite and Key Size

Ciphersuites	256 bits	1024 bits	2048 bits	4096 bits
RSA-PKCS1.5 SHA1	-	7.32 %	24.36 %	52.61 %
RSA-PKCS1.5 SHA224	-	7.75 %	30.92 %	45.23 %
RSA-PKCS1.5 SHA256	-	6.88 %	28.42 %	45.47 %
RSA-PKCS1.5 SHA384	-	8.89 %	31.58 %	43.67 %
RSA-PKCS1.5 SHA512	-	7.16 %	28.37 %	50.39 %
RSA-PSS SHA1	-	8.00 %	30.33 %	45.11 %
RSA-PSS SHA224	-	7.66 %	31.82 %	45.41 %
RSA-PSS SHA256	-	8.45 %	30.35 %	56.02 %
RSA-PSS SHA384	-	8.20 %	28.12 %	47.46 %
RSA-PSS SHA512	-	-	26.1 %5	55.51 %
ECDSAp256	5.92 %	-	-	-

A clear observation we can make is the fact that all ciphersuites that used a RSA key pair registered a high standard deviation percentage. Because of this common point, we decided to measure the time spent to solely generate a RSA key pair, varying the key size used. The obtained values are presented in figure 5.10. In this plot, we decided to set the key side for ECDSA and RSA to 256 and 2048 respectively.

With the acquired results we can deduce that the high standard deviation registered during the attestation process comes from the key pair generation function. To generate a strong RSA key pair, the function needs two very large pseudo-random prime numbers that will be later used to form the private and public keys [54]. Therefore, due to the randomness associated with the prime numbers choice process, the time it takes to effectively generate a key can vary, producing the presented standard deviation.

Despite the variation caused by the RSA key generation, the same phenomenon did not happen during the ECDSA key generation. This is mostly since the generated key pair has a very small size (256-bit length) in comparison with the other generated keys. Nonetheless, ECDSA is increasing its popularity as being a more secure and faster cryptographic function [5] and that is the reason why we decided to evaluate its performance in our solution.

During this experimental evaluation, our objective was to measure the time spent to generate an attestation proof, capable of authenticating our components and ensure that those same components are considered safe during the system boot. With the obtained results and analysis we concluded that the process can obtain similar results, even if used with different ciphersuites and different key sizes. If we consider the key generation

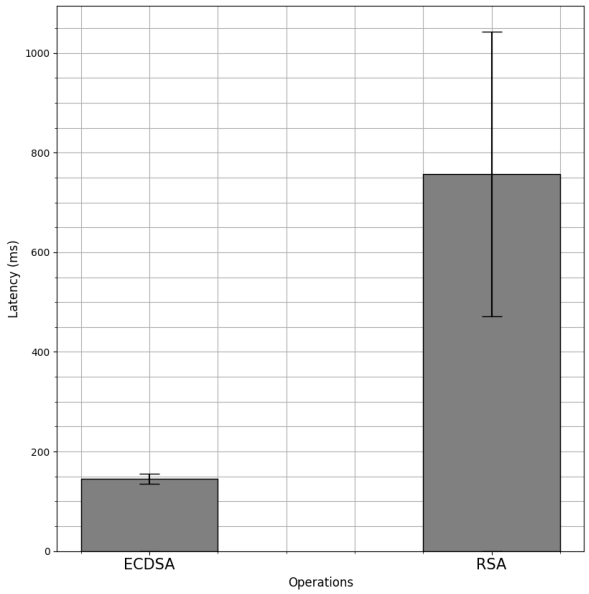


Figure 5.10: Key Generation Process Latency

function to be part of the attestation process, significant variations in the measured times are bound to occur due to the used key size.

5.6 Summary

To validate and analyze the developed solution, we conducted a set of experimental evaluations that: (i) examinated the solution performance and overhead, including comparative evaluation of operations, as also performance measurement of operations executed by the components; (ii) analyzed profiling indicators of Runtime components; (iii) recorded resource allocation activity spent during the execution of operations between a normal wallet vs. a wallet with our solution; and (iv) assessment of the attestation Protocol performance latency.

TWallet System Performance. Considering the experiments to observe the performance of the TWallet System in our prototype, the obtained results show that Delete and Balance operations are faster than the other operations. We must notice that these operations were originally the fastest ones. The History of Transaction could be as fast as the Balance operation, however, due to the great standard deviation values caused by the communication between the application and the Ethereum Blockchain, its execution time is greater. The Create Credentials, Load Credentials and Send Transaction proved to be the slowest operations since they originally required a lot of resources to operate, with the Send Transaction being the slowest of them all, since it must wait until the blockchain

confirms that the sent transaction was concluded and will be confirmed in the future. The operations, despite adding calls to our solution functions seemed to not suffer a significant increase in their execution time, since our solution is not part of the dominant percentage.

Profiling. Regarding the profiling evaluations, we analyzed the storage cost of our solution and measured its system and application boot time. The Storage Cost results showed us that despite the increase in the application size, it was in the order of kB, not being significant to increase the application storage requirements. With the registered boot times, we concluded that our solution, even considering the OP-TEE System Boot, was not the dominant part of the boot process, with that role belonging to the Rich OS boot. Regarding the application boot and considering our tested applications, our solution proved to be slower than the normal application boot time by more than 100%.

System Resources. Considering the CPU utilization operations such as Create Credentials, Load Credentials and Balance it was possible to observe an increase in the allocated time of the CPU resources, but not an increase of the CPU spent percentage. The Delete Credentials was the operation that expressed the greatest change since this operation normally would not require any allocation of resources from the CPU, and with our solution, the allocation was needed. The Send Transaction and History of the transaction, despite its irregular values we saw a point in common in both operations, which is the extension and increase of the CPU allocated resources due to the write requests done to our secure components.

Attestation Service. Concerning the Attestation Process performance measurement, we decided to vary the used ciphersuite and key size during the key generation process. We observed that despite using different ciphersuites and key sizes, the presented results were in the same interval, mostly because the key generation process was excluded from the attestation protocol, which we consider to be a great entropy factor. We then decided to include the key generation function and were able to deduce that the larger the key size the larger the time spent to generate the key. Furthermore, we noticed an increase in the standard deviation values, caused by the randomness of the key generation process. With the observed results, we concluded that considering the key generation process part of the protocol, the Attestation Process can vary depending on the used key size.

In the end, we are satisfied with our solution since it was able to provide more security over a set of operations of an existing crypto wallet, where some of those operations connected to a real Ethereum network, in this case, the Rinkeby testnet. From a performance perspective, the solution proved to be slightly slower in comparison with the "unprotected application", wherein in some cases, the operation took more than 2 times the initial time to finish its execution. In this case, we can verify that security and trustability always come with some performance drawbacks. However, the application backed by the TWallet, compared with a base application, did not cause an overhead significant enough to diminish the overall user experience.

6

Conclusion and Final Remarks

The main objective and goal of the work developed in this dissertation was to design, develop and validate a trustable model and runtime system that offers trust computing-based components in an ARM TrustZone enabling software stack, to protect sensitive applications. Our main focus was to make use of this solution to strengthen the functionalities of cryptocurrency wallets. However, this same solution can be used for a specific family of applications, namely mobile banking, mobile payment, and mobile ticketing, where sensitive operations and data management, and processing capabilities are involved. To address the problem, we designed a generic architecture for cryptocurrency wallet applications leveraged by the ARM TrustZone Technology capabilities. The solution consisted of a set of secure components executed inside an isolated and trustable execution environment where each component would be responsible for reinforcing a specific functionality. Our developed components allowed for the secure storage and access of wallet credentials, secure storage of wallet information regarding its balance and recent transaction history, and generation of a log relative to the operations requested to these same secure components. Comparatively, with pre-existent solutions, we tried to provide additional functionalities, like an attestation service of our secure components and a monitoring service responsible for filtering incoming requests, that would make us a better option when trying to reinforce the referred family of applications with some security guarantees. For this purpose, we implemented our solution under shielded hardware trust assumptions, provided by the ARM TrustZone trusted execution environment.

6.1 Results and Contributions

Considering our planned objective, the dissertation proposes the design of a trusted framework model and related runtime system that offers a set of trusted computing-based components, executed in an ARM TrustZone enabling software stack to protect sensitive applications. In our journey throughout this document we achieved the expected contributions, around the following aspects in the proposed solution:

- Definition of the TWallet System Model and Architecture as a generic framework offering a software stack for the development of trusted applications enabled by ARM devices equipped with ARM TrustZone compliant chipsets.

- Implementation of the TWallet framework, by providing a set of functionalities, all backed by a shielded hardware trusted execution environment, guaranteeing authentication, confidentiality, and integrity to the performed operations. This set of functionalities are executed through a group of trusted applications, where each application is responsible for a specific set of tasks. The TWallet framework can provide support to a family of applications, namely Mobile Banking, Ticketing apps, and Mobile Payment, and highlighting given our thesis objective, cryptocurrency wallets.

- Development of an Ethereum Crypto Wallet App as a proof-of-concept prototype of the designed solution. This prototype following the TWallet model was integrated as a remote wallet to interact with the Rinkeby Ethereum Test Network - a test network used for blockchain development testing before deployment on the Ethereum main network.

- With the developed prototype, we conducted a set of experimental evaluations and validations. In this test bench, we analyzed: (i) comparative evaluation of operations performed on a cryptocurrency wallet, using or not our solution by measuring the latency and throughput; (ii) profiling of Runtime components, such as analysis of space occupancy, and the boot time of solution system setup; (iii) observation of System Resources spent while performing wallet operation, with and without our solution; (iv) analysis of Attestation Protocol performance latency.

With all points considered, we were able to provide a solution capable of enforcing relevant security properties for android applications managing sensitive data and operations, in a TEE backed by the ARM TrustZone isolation guarantees. Our developed solution runs on top of OP-TEE, a TEE that offers a set of functionalities that allowed our project to run in the ARM TrustZone Technology Secure World. Without disregarding the many functionalities our developed components have, these components are additionally protected by an attestation process that runs before they start attending requests so that we can ensure that the components are indeed secure and trustable. The TWallet System is therefore a trustable and safe solution, available to all interested researchers and developers, not only for study purposes but also for possible future enhancements.

6.2 Developed Experience and Knowledge Consolidation

Following the planned objective and expected contribution, we were able to achieve interesting results in the validation of the proposed solution. Furthermore, the experience

acquired throughout the design of our solution allowed for consolidation on the knowledge and hands-on experience in designing trusted mobile applications, supported in the trusted execution environments enabled by ARM TrustZone Technology. We believe that both these theoretical and practical developed skills will be a valuable knowledge base for future initiatives in the relevant challenging fields of trusted computing, mobile security systems, and related emerging technology.

We present some highlights from the acquired experience in the dissertation topics:

- Experience in the use of real-world Software/Firmware stack on the development of trusted computing systems, namely on ARM-enabled hardware platforms. This includes the learning and challenging parts of using TEEs and TrustZone Technology.
- Knowledge of low-level abstraction mechanisms, such as Context Switch, hardware, and software data structures relationships, and a deeper understanding of operating systems.
- Experience in developing Android apps for Android 9.0 and above, and a better understanding of methods used to intertwine different programming languages, namely given our thesis development, Java and C.

The approach used while designing TAs to be executed in the OP-TEE, and consequently the TrustZone environment, posed many challenges. With focus, perseverance, and help from the community, it was possible to overcome the many obstacles and conclude our solution. The express the required dedication we emphasize the following topics:

- Plenty of online information, but with no stable methodology while designing applications.
- The documentation online properly specifies the operations and functions provided by OP-TEE. However, most of the acquired knowledge and how to make use of those functions was through community forums and contact with people that faced the same problems or had a higher level of expertise regarding the matter.
- OP-TEE is currently an ongoing project and it is used as a trusted execution environment to develop trusted applications, with the aid of Global Platform's API. Because of the constant changes this execution environment suffers, it can hinder development. One must take this factor into account when trying to develop trusted applications.
- Some of the difficulties were encountered while trying to set up the OP-TEE in different development boards. Given our objectives and constraints, and considering someone not familiar with the firmware, kernel, and hardware-level experience, the process of setting up an environment to properly develop and the test was slowed.

Despite all these presented issues, we consider they have enriched our background and will be relevant in future technologies with trustability requirements since these technologies are dependent on the TEE's own capabilities and architectures, as also on the supported platforms.

6.3 Future Work

Given our final solution and characteristics, there are interesting remarks that can be taken while thinking about future improvements. We summarize those remarks:

- Our principal remark would be to conclude our initial design and architectural considerations given our solution. For that, the isolation capability, provided by the TEE Adaptation and Isolation Layer should be properly implemented and tested, so that each of the secure components could have an extra security layer over the ones already guaranteed.
- A second direction would be the overall upgrade of our development environment. Despite the Hikey 960 board being a fairly recent model with good specifications in comparison with other boards, it is clearly different when compared with mobile devices, in particular mobile phones. To shorten the specification difference, we propose an upgrade to the more recent Hikey board currently available in the market, which at the moment of this thesis writing would be the Hikey 970. Furthermore, we consider that a software update would be advisable. In our case, the Android version we used could be updated to a more recent one, namely Android 11, and the OP-TEE version could also be upgraded to 3.14.
- An interesting goal could be to reuse our design model and develop solutions in the context of developing trustable real-world usable applications, such as Mobile Ticketing, Mobile Banking, Mobile Payment Apps, or Cryptocurrency Wallets. However, due to the difficulty of having open-based ARM TrustZone solutions on commercially available mobile smartphones in the market, it would be necessary to target a development kit, in a collaboration or partnership with some specific vendor.
- Finally, we believe that our work effort could be relevant for future developers and researchers in this field, and would be interesting to create a guide for a methodology development process for trusted applications running on OP-TEE and ARM TrustZone. However, this objective seems fairly difficult to achieve as it would be needed to have a main procedure, responsible to adapt the development process according to the used hardware platform.

Bibliography

[1] 96Boards. *Hikey960 Screen Problem*. 2019. URL: https://discuss.96boards.org/t/hikey960-screen-problem/7136 (cit. on p. 46).

[2] 96Boards. *HiKey970*. 2021. URL: https://www.96boards.org/product/hikey970/ (cit. on p. 6).

[3] 96boards. *HiKey960 Development Board User Manual*. 2021. URL: https://www.96boards.org/documentation/consumer/hikey/hikey960/hardware-docs/hardware-user-manual.md.html (cit. on p. 61).

[4] T. Alsop. *Arm - statistics & facts*. Oct. 2020. URL: https://www.statista.com/topics/7087/arm/ (cit. on p. 3).

[5] M. Amara and A. Siad. "Elliptic Curve Cryptography and its applications". In: *International Workshop on Systems, Signal Processing and their Applications, WOSSPA*. 2011, pp. 247–250. DOI: 10.1109/WOSSPA.2011.5931464 (cit. on p. 81).

[6] Android. *Android Profiler*. 2021. URL: https://developer.android.com/studio/profile/android-profiler (cit. on p. 63).

[7] Android. *Platform Architecture*. 2021. URL: https://developer.android.com/guide/platform/ (cit. on p. 26).

[8] Android. *Trusty TEE*. 2020. URL: https://source.android.com/security/trusty (cit. on pp. 10, 11).

[9] ARM. *ARM security technology. Building a secure system using trustzone technology*. Tech. rep. ARM Ltd., 2009. URL: https://documentation-service.arm.com/static/5f1ffa25bb903e39c84d7e98?token= (cit. on pp. 12, 17, 18).

[10] A. Armando, A. Merlo, and L. Verderame. "Trusted host-based card emulation". In: *2015 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, July 2015. DOI: 10.1109/hpcsim.2015.7237043 (cit. on pp. 3, 19, 23, 24).

[11] S. Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *OSDI*. 2016 (cit. on p. 11).

[12] Consensys. *Infura: Ethereum API, IPFS API & Gateway, ETH Nodes as a Service*. 2022. URL: https://infura.io/ (cit. on p. 62).

[13] I. Corporation. *Software Guard Extensions Programming Reference*. Tech. rep. Intel Corporation, 2014. URL: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf (cit. on pp. 3, 11).

[14] V. Costan and S. Devadas. "Intel SGX Explained". In: *IACR Cryptol. ePrint Arch.* 2016 (2016), p. 86 (cit. on p. 11).

[15] V. Costan, I. Lebedev, and S. Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". In: *USENIX Security Symposium*. 2016 (cit. on p. 13).

[16] Cybertalk.org. *Mobile Security Report 2021*. Research rep. Cybertalk.org, 2021. URL: https://www.cybertalk.org/wp-content/uploads/2021/04/mobile-security-report-2021.pdf (cit. on p. 1).

[17] W. Dai et al. "SBLWT: A Secure Blockchain Lightweight Wallet Based on Trustzone". In: *IEEE Access* 6 (2018), pp. 40638–40648. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2856864 (cit. on pp. 3, 23, 24).

[18] A. Developers. *Android 9 Pie*. Jan. 2022. URL: https://developer.android.com/about/versions/pie (cit. on p. 61).

[19] X. Developers. "Huawei and Linaro launch the HiKey 970 development board with the Kirin 970 SoC". In: (2018). URL: https://www.xda-developers.com/huawei-linaro-hikey-970-development-board-kirin-970-soc/ (cit. on p. 15).

[20] ENISA. *ENISA Threat Landscape 2021 Report*. Tech. rep. ENISA, Oct. 2021. URL: https://www.enisa.europa.eu/ (cit. on p. 1).

[21] Ethestats. *Ethereum Blockchain Statistics*. 2021. URL: https://ethstats.net/ (cit. on p. 68).

[22] R. P. Foundation. "Raspberry Pi Blog". In: (2019). URL: https://www.raspberrypi.org/blog/raspberry-pi-4-on-sale-now-from-35/ (cit. on p. 16).

[23] M. Gentilal, P. Martins, and L. Sousa. "TrustZone-backed bitcoin wallet". In: *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems - CS2 '17*. ACM Press, 2017. DOI: 10.1145/3031836.3031841 (cit. on pp. 3, 19, 23, 24).

[24] GlobalPlatform. *Introduction to Secure Elements*. Tech. rep. 2018. URL: https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Secure-Element-15May2018.pdf (cit. on p. 12).

[25] I. GlobalPlatform. *TEE Internal Core API Specification Version 1.2*. Tech. rep. GlobalPlatform Technology, 2019. URL: https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/ (cit. on pp. 10, 12).

[26] I. GlobalPlatform. *TEE System Architecture v1.2*. Tech. rep. GlobalPlatform Technology, 2018. URL: https://globalplatform.org/specs-library/tee-system-architecture-v1-2/ (cit. on p. 3).

[27] Google. "Android Open Source Project". In: (2017). URL: https://source.android.com/ (cit. on p. 10).

[28] J.-R. W. Group. *JSON-RPC 2.0 Specification*. 2010. URL: https://www.jsonrpc.org/specification (cit. on p. 62).

[29] T. C. Group. *TPM 2.0 Architecture*. Tech. rep. 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf (cit. on p. 34).

[30] T. C. Group. *TPM Main: Part 1 Design Principles, Version 1.2*. Tech. rep. TCG, 2011. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf (cit. on pp. 13, 28).

[31] L. Guan et al. "TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone". In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, June 2017. DOI: 10.1145/3081333.3081349 (cit. on pp. 3, 19, 28).

[32] R. Hat. "Virtualization: What is a virtual machine (VM)?" In: (). URL: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine (cit. on p. 8).

[33] W. Huzaini et al. "Mobile Ticketing System Employing TrustZone Technology". In: *International Conference on Mobile Business (ICMB'05)*. IEEE. DOI: 10.1109/icmb.2005.71 (cit. on p. 3).

[34] L. Kernel. "Little Kernel". In: (2016). URL: https://github.com/littlekernel/lk (cit. on p. 10).

[35] P. Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. 2019 (cit. on pp. 2, 26).

[36] M. Lentz et al. "SeCloak". In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, June 2018. DOI: 10.1145/3210240.3210334 (cit. on pp. 3, 20).

[37] L. Limited. "Linaro". In: 2020. URL: https://www.linaro.org/ (cit. on p. 9).

[38] Linaro. "HKG15-311: OP-TEE for Beginners and Porting Review". In: (2015). URL: https://pt.slideshare.net/linaroorg/hkg15311-optee-for-beginners-and-porting-review (cit. on p. 21).

[39] Linaro. *OP-TEE: Open Portable Trusted Execution Environment*. 2020. URL: https://www.op-tee.org/ (cit. on pp. 5, 16).

[40] J. Lind et al. "Teechain". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, Oct. 2019. DOI: 10.1145/3341301.3359627 (cit. on p. 3).

[41] M. Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018 (cit. on pp. 2, 26).

[42] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[43] W. L. Ltd. *Web3j*. 2019. URL: https://docs.web3j.io (cit. on pp. 68, 77).

[44] e. a. Moriarty. "RFC 8017". In: *IETF* (2016). URL: https://datatracker.ietf.org/doc/html/rfc8017#section-9.1.1 (cit. on p. 80).

[45] C. Müller et al. "TZ4Fabric: Executing Smart Contracts with ARM TrustZone". In: *ArXiv* abs/2008.11601 (2020) (cit. on p. 3).

[46] ODROID. "ODROID Wiki". In: (2020). URL: https://wiki.odroid.com/ (cit. on p. 16).

[47] Oracle. "Java Native Interface (JNI)". In: (2021). URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/ (cit. on p. 57).

[48] S. Pinto et al. "LTZVisor: TrustZone is the Key". In: *ECRTS*. 2017 (cit. on p. 15).

[49] S. Pinto et al. "Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems". In: *IEEE Computer Architecture Letters* 16 (2017), pp. 158–161 (cit. on p. 15).

[50] S. Pinto and N. Santos. "Demystifying Arm TrustZone". In: *ACM Computing Surveys* 51.6 (Feb. 2019), pp. 1–36. DOI: 10.1145/3291047 (cit. on pp. 3, 13, 14, 18).

[51] H. Raj et al. "fTPM: A Software-Only Implementation of a TPM Chip". In: *USENIX Security Symposium*. 2016 (cit. on pp. 18, 24, 31).

[52] C. Report. *Mobile Security Report 2021: Insights on Emerging Mobile Threats*. Tech. rep. Checkpoint Report, 2021. URL: https://pages.checkpoint.com/mobile-security-report-2021.html (cit. on p. 1).

[53] Rinkeby. *Rinkeby testnet*. 2021. URL: https://www.rinkeby.io/#stats (cit. on p. 62).

[54] R. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Commun. ACM* 21 (1978), pp. 120–126 (cit. on p. 81).

[55] I. Rocha. "A Mobile Secure Bluetooth-Enabled Cryptographic Provider". MA thesis. Faculdade de Ciencias e Tecnologias, NOVA University, 2019 (cit. on p. 13).

[56] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *TrustCom 2015*. 2015 (cit. on p. 9).

[57] N. Santos et al. "Using ARM trustzone to build a trusted language runtime for mobile applications". In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems - ASPLOS '14*. ACM Press, 2014. DOI: 10.1145/2541940.2541949 (cit. on pp. 3, 19).

[58] K. Scarfone, M. P. Souppaya, and P. Hoffman. "Guide to Security for Full Virtualization Technologies". In: 2011 (cit. on p. 8).

[59] Sierraware. "SierraTEE for ARM® TrustZone® and MIPS". In: (2016). URL: https://www.sierraware.com/open-source-ARM-TrustZone.html (cit. on p. 10).

[60] Sierraware. "SierraVisor: Embedded Virtualization". In: (2012). URL: https://www.sierraware.com/SierraVisor_Embedded_Hypervisor_Datasheet.pdf (cit. on p. 15).

[61] M.
bibinitperiod N. C. Smart Card Alliance. *Host Card Emulation (HCE) 101*. Tech. rep. 2014. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.568.6523&rep=rep1&type=pdf (cit. on pp. 4, 13).

[62] G. 3. Staff. *Huawei Launches HiKey 960, a Super-Powered Raspberry Pi-Style Development Board*. Apr. 2017. URL: https://gadgets.ndtv.com/laptops/news/huawei-launches-hikey-960-a-super-powered-raspberry-pi-style-development-board-1687262 (cit. on p. 15).

[63] Statista.com. "Statista Statistics: Number of detected malicious installation packages on mobile devicesworldwide (2015-2021)". In: (2021). URL: https://www.statista.com/statistics/653680/volume-of-detected-mobile-malware-packages/ (cit. on p. 1).

[64] Symantec/Broadcom. *Internet Security Threat Report*. Tech. rep. 24. Symantec/Broadcom, Feb. 2019. URL: https://docs.broadcom.com/doc/istr-24-2019-en (cit. on p. 1).

[65] Symantec/Broadcom. *Internet Security Threat Report*. Tech. rep. 23. Symantec/Broadcom, Feb. 2028. URL: https://docs.broadcom.com/doc/istr-23-executive-summary-en (cit. on p. 1).

[66] Topaco. *Signing data is throwing exception for RSA SHA512 algo with key size as 512 and 1024*. Mar. 2018. URL: https://stackoverflow.com/questions/66673428/signing-data-is-throwing-exception-for-rsa-sha512-algo-with-key-size-as-512-and (cit. on p. 80).

[67] TrustedFirmware. "OP-TEE documentation". In: 2020. URL: https://optee.readthedocs.io/en/latest/ (cit. on pp. 9, 46).

[68] TrustedFirmware. "OP-TEE Secure Storage". In: 2020. URL: https://optee.readthedocs.io/en/latest/architecture/secure_storage.html (cit. on pp. 48, 49, 51).

[69] TrustedFirmware.org. *OP-TEE 3.12.0 Documentation*. Jan. 2021. URL: https://optee.readthedocs.io/%5C_/downloads/en/3.12.0/pdf/ (cit. on p. 61).

[70] TrustZone. *ARM TrustZone*. Tech. rep. ARM Ltd., 2017. URL: https://www.arm.com/products/silicon-ip-security (cit. on p. 3).

[71] C.-c. Tsai, D. Porter, and M. Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *USENIX Annual Technical Conference*. 2017 (cit. on p. 11).

[72] D. A. Wheeler. "SLOCCount". In: (2017). URL: https://dwheeler.com/sloccount/ (cit. on p. 47).

[73] L. B. William Stallings. *Computer Security: Principles and Practice, Global Edition*. Pearson, 2018. 800 pp. ISBN: 1292220619. URL: https://www.ebook.de/de/product/31543227/william_stallings_lawrie_brown_computer_security_principles_and_practice_global_edition.html (cit. on p. 7).

[74] J. Winter. "Trusted computing building blocks for embedded linux-based ARM trustzone platforms". In: *Proceedings of the 3rd ACM workshop on Scalable trusted computing - STC '08*. ACM Press, 2008. DOI: 10.1145/1456455.1456460 (cit. on p. 3).

[75] S. D. Yalew. "Mobile Device Security with ARM TrustZone". In: 2018 (cit. on p. 3).

[76] ZDnet. "Raspberry Pi has now sold 30 million tiny single-board computers". In: (2019). URL: https://www.zdnet.com/article/raspberry-pi-now-weve-sold-30-million/ (cit. on p. 15).

[77] F. Zwanzger. *Rinkeby: Main Ethereum Proof-of-Authority Testnet*. 2021. URL: https://www.anyblockanalytics.com/networks/ethereum/rinkeby/ (cit. on p. 62).

I

Hikey960 AOSP+OP-TEE Setup

The following instructions directly were taken from the official OP-TEE documentation (I.4.1) and adapted to our setup.

I.1 Prerequisites

Before proceeding to the build phase, the user must met a set of prerequisites, needed for a correct build process. Those are:

1. The computer OS where you are going to build the system must be a Linux Ubuntu 18.08 LTS. At the time of writing this thesis, the most recent supported Ubuntu version for the build process is this only.

2. You should be able to build AOSP for Hikey according to the official instructions (I.4.2). Please note that the AOSP build and this build are **COMPLETELY SEPARATED** from one another. This prerequisite is only used to verify and make sure that your system has everything needed to build AOSP without any issue.

Additionally, a set of packages, needed for the OP-TEE and AOSP build have to be installed. In order to correctly install the desired packages, the user must enable the installation of i386 architecture packages and update the package managers database.

```
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
```

After that, you can proceed to the installation of the needed packages:

```
$ sudo apt-get install android-tools-adb android-tools-fastboot \
    autoconf automake bc bison build-essential ccache cscope curl \
    device-tree-compiler expect flex ftp-upload gdisk iasl \
    libattr1-dev libcap-dev libfdt-dev libftdi-dev libglib2.0-dev \
    libgmp-dev libhidapi-dev libmpc-dev libncurses5-dev libpixman-1-dev \
    libssl-dev libtool make mtools netcat ninja-build python-crypto \
```

```
python3-crypto python-pyelftools python3-pycryptodome \
python3-pyelftools python-serial python3-serial rsync unzip \
uuid-dev xdg-utils xterm xz-utils zlib1g-dev default-jre \
```

Besides the presented packages, the Repo tool must be installed in your computer. The steps to achieve it are:

1. Make sure you have a bin/directory in your home directory and that it is included in your path:

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
```

2. Download the Repo tool and ensure that it is executable:

```
$ curl https://storage.googleapis.com/git-repo-downloads/repo \
       > ~/bin/repo
$ chmod a+x ~/bin/repo
```

I.2 Build Instructions

The build process is fairly simple. First, clone the repository containing the files needed for the build process.

```
$ git clone https://github.com/linaro-swg/optee_android_manifest [-b <release_tag>]
# release tags come in the form of X.Y.Z, e.g. 3.8.0
$ cd optee_android_manifest
```

and then run the following commands:

```
$ ./sync-p-hikey960.sh
$ ./build-p-hikey960.sh
```

Both steps **MUST** finish with no errors. For `sync*.sh` scripts, that means there must be no errors prior to the `Sync done!` console output. For `build*.sh` scripts, that means there must be a `build completed successfully (MM:SS (mm:ss))` console output! If there are errors, then there is no point in trying to flash the device.

I.3 Flashing the Image

For the flash process to succeed, in the Hikey board, the switches 1 and 2 must be up, while the 3rd switch is down. This corresponds to the board Recovery mode. More details related to this and other modes can be found in `device/linaro/hikey/installer/hikey960/README`. After that, invoke:

```
$ cp -a out/target/product/hikey960/*.img device/linaro/hikey/installer/hikey960/
$ sudo ./device/linaro/hikey/installer/hikey960/flash-all.sh /dev/ttyUSBn
```

where the `n` in `/dev/ttyUSBn` corresponds to a number. Note that the device only remains in this mode for about 90 seconds. If you take too long to run the flash commands, it will need to be reconnected again.

I.3.1 Warning

There might be some cases that, while running the `flash-all.sh` script, the progress gets stuck in `<Waiting for any device>`. In this case, the user must turn off the board, disconnect it from the pc and power supply, and change the switches to the fastboot mode (switches 1 and 3 up, 2 down). After that, reconnect the board, turn it on and execute the fastboot commands present at the end of the `flash-all.sh` script.

I.4 References

I.4.1 OP-TEE Documentation

https://optee.readthedocs.io/en/latest/index.html

I.4.2 AOSP instructions

https://source.android.com/setup/build/devices#960userspace