Secure migration of WebAssembly-based mobile agents between secure enclaves

Master of Science in Technology Thesis
University of Turku
Faculty of Technology
Security of Networked Systems
October 2021
Vasile Adrian Bogdan Pop

Supervisors:
Seppo Virtanen (University of Turku)
Petri Sainio (University of Turku)
Arto Niemi (Huawei Technologies Oy)

Acknowledgements

I would like to thank my supervisors Seppo Virtanen and Petri Sainio from the University of Turku for their guidance in writing this work, and my supervisor Arto Niemi from the Huawei Technologies Oy Helsinki Research & Development Center for not only advising me in the process of writing this work but also during my stay at the R&D Center. Further, I would also like to thank the remaining members of the team I worked with, Jan-Erik Ekberg, Valentin Manea, and Antti Rusanen, for their technical support in carrying out the various tasks during my internship.

Finally, I would like to thank my parents, Vasile Pop and Irina Pop, for all the support they have given me and which has made me reach this achievement in my life, and Sara Desiati and my friends for their moral support throughout this experience.

UNIVERSITY OF TURKU
Faculty of Technology

Vasile Adrian Bogdan Pop: Secure migration of WebAssembly-based mobile agents between secure enclaves

Master of Science in Technology Thesis, 98 p.
Security of Networked Systems
October 2021

Cryptography and security protocols are today commonly used to protect data at-rest and in-transit. In contrast, protecting data in-use has seen only limited adoption. Secure data transfer methods employed today rarely provide guarantees regarding the trustworthiness of the software and hardware at the communication endpoints.

The field of study that addresses these issues is called Trusted or Confidential Computing and relies on the use of hardware-based techniques. These techniques aim to isolate critical data and its processing from the rest of the system. More specifically, it investigates the use of hardware isolated Secure Execution Environments (SEEs) where applications cannot be tampered with during operation. Over the past few decades, several implementations of SEEs have been introduced, each based on a different hardware architecture. However, lately, the trend is to move towards architecture-independent SEEs.

As part of this, Huawei research project is developing a secure enclave framework that enables secure execution and migration of applications (mobile agents), regardless of the underlying architecture. This thesis contributes to the development of the framework by participating in the design and implementation of a secure migration scheme for the mobile agents. The goal is a scheme wherein it is possible to transfer the mobile agent without compromising the security guarantees provided by SEEs. Further, the thesis also provides performance measurements of the migration scheme implemented in a proof of concept of the framework.

Keywords: Trusted Computing, Secure Enclaves, Secure Migration, Trusted channel, Mobile Agents

Contents

1 Introduction **1**

1.1 Project Overview 5

1.1.1 Personal Contributions 7

2 Background **9**

2.1 Information Security Objectives 9

2.1.1 Confidentiality 9

2.1.2 Integrity 10

2.1.3 Availability 10

2.1.4 Authenticity 11

2.1.5 Non-Repudiation 11

2.1.6 Trust & trustworthiness 11

2.2 Cryptographic background 12

2.2.1 Cryptographic Primitives 12

2.2.2 Public Key Infrastructure 20

2.2.3 Remote Attestation 21

2.2.4 Secure Channel Protocols 23

2.3 Terminology 26

2.3.1 Trusted Computing 26

2.3.2 Trusted Computing Base 27

2.3.3 Secure/Trusted Execution Environment 27
2.3.4 Secure Enclave 28
2.4 Secure Technologies 29
2.4.1 Rust 29
2.4.2 WebAssembly 30

3 Prior Art **31**
3.1 Distributed Computing 31
3.1.1 Mobile Code 32
3.1.2 Mobile Agents 33
3.2 Secure Execution Environments & Secure Enclaves 34
3.2.1 External Security Co-Processors 34
3.2.2 Embedded Security Co-Processors 37
3.2.3 Processor Secure Environments 39
3.2.4 Existing Attacks 45

4 Architecture **48**
4.1 Host application 49
4.1.1 Host application logic 50
4.1.2 Host Migrator 50
4.1.3 Enclave driver 51
4.2 Secure enclave 51
4.2.1 Mobile Agent 52
4.2.2 Enclave runtime 52
4.3 Enclave harness 56
4.3.1 Platform attester 56

5 Migration design **58**
5.1 Requirements 58

5.1.1 Security Requirements 58
5.2 Migration phases 60
5.2.1 Snapshotting & Resumption 61
5.2.2 Serialization & Deserialization 62
5.2.3 Attestation 64
5.2.4 Transmission 73
5.3 Migration flow 75

6 Implementation 78
6.1 Host application 80
6.1.1 Host application logic 80
6.1.2 Enclave driver 80
6.1.3 Host migrator 81
6.2 Secure enclave 82
6.2.1 Mobile Agent 83
6.2.2 Enclave runtime 83
6.3 Enclave harness 86
6.3.1 Platform attester 87

7 Benchmarks & Analysis 88
7.1 Benchmarks setup 88
7.2 Benchmarks methodology & Results 89
7.3 Security analysis 94

8 Conclusions 97

References 99

List of Figures

2.1 Remote attestation procedure 22
2.2 Transport Layer Security Handshake 25

3.1 Architectures of Secure Execution Environments 35

4.1 Secure enclave framework architecture. 48

5.1 Attestation claims employed in the migration process 67
5.2 Trust chain employed in the migration process 70
5.3 Secure migration scheme 76

6.1 Technologies employed in the Proof-of-Concept 79

7.1 Distribution of the migration time 92
7.2 Mobile Agent size vs. Migration time 93

List of Tables

7.1 Migration time measurements 91

List of acronyms

MEC Mobile Edge Computing

AEAD Authenticated encryption with associated data

AE Authenticated Encryption

AES Advanced Encryption Standard

AKE Authenticated Key Exchange

CA Certificate Authority

CBC Cipher Block Chaining

CCA Confidential Compute Architecture

CC Cloud Computing

CCC Confidential Computing Consortium

CIA Confidentiality Integrity Availability

COD Code on Demand

DER Distinguished Encoding Rules

DH Diffie-Hellman

DoS Denial of Service

ECB Electronic Code Book

EPC Enclave Page Cache

FFI Foreign Function Interface

GCM Galois/Counter Mode

HSM Hardware Security Module

IoT Internet of Things

JADE Java Agent DEvelopment Framework

JVM Java Virtual Machine

KVM Kernel-based Virtual Machine

MAC Message Authentication Code

MCC Mobile Cloud Computing

MITM Man-in-the-middle

MMU Memory Management Unit

NIST National Institute of Standards and Technology

OEM Original Equipment Manufacturer

PKI Public Key Infrastructure

PMP Physical Memory Protection

PoC Proof-of-Concept

PSE Processor secure environment

PSP Platform Security Processor

REE Rich Execution Environment

REV Remote Evaluation

RMI Realm Management Interface

RMM Realm Management Monitor

RSA Rivest Shamir Adleman

RSI Realm Service Interface

RTOS Real-Time Operating System

SDK Software Development Kit

SEE Secure Execution Environment

SEP Security Enclave Processor

SEV Secure Encrypted Virtualization

SoC System on Chip

TA Trusted Application

TCB Trusted Computing Base

TLS Transport Layer Security

TPM Trusted Platform Module

TSL Trusted Sockets Layer

WASI WebAssembly Interpreter

Wasm WebAssembly

1 Introduction

In the last decades, digitalization has had a huge effect on industries and us as individuals. Industries have been able to cut costs, increase their efficiency and productivity, and cope with the growing demand by digitalizing their services. At the same time, we as individuals have been offered better products that have increased the comfort of our daily life. Among all the different digital facilities offered to us, mobile devices are the ones that have the most influence on our lives. Their constant connectivity and mobility allow users to access the desired services at any time from everywhere and sometimes are even strictly required to access certain products. For instance, smartphones not only are connecting people all over the world but they are becoming essential due to the different services they can offer, such as handling banking accounts, payments, tickets, digital identities, etc. In addition to smartphones, also IoT devices are able to add value to individuals and industries. In fact, they can be employed in a great range of applications, from monitoring production processes and infrastructures to augmenting the comfort of our homes.

Despite the performance enhancements that mobile devices and IoT have experienced, their resources are still limited and cannot always satisfy the computational intensive applications that the end-user may need to use. Hence, computation offloading techniques such as Mobile Cloud Computing (MCC) started to be used as a mean to overcome the various limitations imposed by restricted processing capa-

bilities, battery lifetime, and storage capacity [1]. Furthermore, to have an even better user experience, new technologies like Mobile Edge Computing (MEC) have been standardized [2]. These are addressing the latency issues of traditional Cloud Computing (CC) techniques by moving the computational capabilities to the edge of the network. As in Cloud Computing, MEC is based on virtualized platforms able to allocate physical resources that will be used to accomplish the tasks required by IoT and mobile devices. The hosting platforms are able to support both, (i) virtualization technologies, such as *virtual machines* (VMs) and *containers*, and (ii) execution environments, like the ones required by *mobile agents*. According to Guo et al. [3], among these technologies, mobile agents seem to be more suitable for IoT and MEC/MCC due to their smaller size and higher flexibility in heterogeneous environments, since they only need a runtime environment to be executed. A mobile agent is interpreted code that can migrate between different hosts, together with its state, to perform specific tasks. One such example is the *JAVA Agent Development Framework* (JADE) [4] based on the Java runtime environment, or the newly introduced WebAssembly runtime that is able to ensure good security guarantees and platform independence.

Mobile devices and IoT together with the MEC and MCC technologies, provide us with many facilities and services but at the expense of our privacy. That is because they need to collect and process huge amounts of data about our activities or our persona, such as sensitive data, and personal information. To address this issue, multiple cryptographic algorithms and protocols are used. These can protect the data at rest (e.g. with AES-XTS based full-disk encryption [5]) and in transit (e.g. the TLS protocol [6]) but not the data in use. Indeed, when the data needs to be processed by the device or offloaded on the cloud, it is still required to be in clear to allow the accomplishment of the task. This can raise major risks for the sensitive data in use unless there are guarantees in place ensuring that the integrity of the

code or the data itself is not compromised. This goes hand in hand with the words of Gene Spafford *"Using encryption on the Internet is the equivalent of arranging an armored car to deliver credit card information from someone living in a cardboard box to someone living on a park bench"* [7], meaning that just the secure transfer of data does not provide any guarantee on the endpoint's trustworthiness. Thus, the actual behavior and execution of the task may be different from the expected one. In the past years, the number of attacks and compromised endpoints on the Internet has significantly increased [8]. In addition to this, especially due to the COVID-19 pandemic during 2020, threats like phishing, ransomware, and malware in general, reached a peak, when measured in the number of attacks [9], raising even more concerns about compromised endpoints.

The issues regarding endpoint trustworthiness and confidentiality of the data in use, may negatively impact the privacy of our data or the secrets processed on our mobile devices and on the cloud. For instance, a compromised platform may allow the disclosure of secret data to third parties at processing time or a malicious endpoint may steal all the information provided to perform the required task. These issues are even more relevant when offloading techniques are employed because ensuring the trustworthiness of remote hosting platforms is a non-trivial task. For these reasons, to fully safeguard sensitive data and secrets from unauthorized third parties and the user itself, the data needs to be protected not only at rest and in transit but also when it is in use. The problem of securing the data in use and providing trustworthiness guarantees about remote endpoints is addressed by a field of study called *Trusted* or *Confidential Computing*. Trusted Computing is grounded on the use of hardware-based techniques to isolate critical data and its processing from the rest of the system. The confinement of the trusted code, the one processing sensitive information, from the untrusted one, can be done through many approaches supported by the *Trusted Computing Base* (TCB). These consist

of cryptographic functions and memory isolation techniques that can guarantee the confidentiality and integrity of the data during the execution of the code. This particular kind of execution context is known as *Secure Execution Environment* (SEE) and is a place where applications cannot be tampered with while operating. In the last decades, diverse implementations of SEEs have been introduced, each one based on a different hardware architecture, which brings its own strengths and weaknesses. All the various solutions can be classified into three main categories:

i **External security co-processors**, like *Hardware Security Modules* (HSMs) and removable *secure elements*.

ii **Embedded security co-processors**, such as *Trusted Platform Modules* (TPMs) [10], secure elements, Google's *Titan M Chip* [11], and Apples's *Security Enclave Processor* [12].

iii **Processor secure environments** (PSEs), like *ARM TrustZone* [13], *Intel SGX* [14], *AMD Secure Encryption Virtualization* (SEV) [15], and *ARM Confidential Compute Architecture* (CCA) [16].

Regardless of the fact that the first two categories of SEEs can provide a higher degree of isolation thanks to their physical separation from the main processor, they are not suitable for a large number of applications. That is because they are either too cumbersome or their overall performance is too restricted. In contrast, PSEs are implemented on the main processor, guaranteeing higher performance and much more flexibility. Even though PSEs are nowadays employed in a wide range of applications, they are still highly bound to the hardware architecture underlying the SEE. And so, it does not get along with the high degree of heterogeneity in mobile devices, IoT, and cloud computing. For this reason, lately, the trend is to move towards architecture independent SEEs, which may suit better MCC and MEC technologies and allow the development of new innovative applications. An

example of this trend is Twine, an Embedded Trusted Runtime for WebAssembly [17]. Due to its implementation upon a WebAssembly-based architecture, it is able to abstract the application from the underlying SEE hardware and run the application in an isolated sandbox provided by the WebAssembly Interpreter (WASI). Into the architecture, a malicious application is prevented from escaping the sandbox, i.e. accessing data without authorization, and the application itself is protected from the untrusted code by the SEE isolation. Furthermore, the abstraction layer in Twine, allows developers to write their applications in any programming language that can be compiled to WebAssembly and execute them on diverse SEEs regardless of the underlying hardware architecture [17].

1.1 Project Overview

Although there is a growing trend to increase the interoperability of software among different SEEs and solutions like Twine already exist, the possibility to securely migrate mobile agents between Secure Execution Environments on different hardware architectures is still missing.

In order to fulfill these needs, a research project at Huawei Technologies Oy Helsinki Research Center is developing a framework that enables the execution and migration of mobile agents between secure enclaves (a special case of SEEs, see Section 2.3.4) independently from the hardware architecture backing the SEEs. In order to clearly frame the goals of the project, these will be listed below:

- provide an architecture able to run mobile agents within a secure enclave, independently of the hardware architecture;
- ensure that mobile agents are isolated from the platform in such a way that they cannot harm the platform itself;

- provide a secure migration mechanism between two platforms that allow mobile agents executing within a secure enclave to be safely relocated;
- ensure that the migration process does not downgrade the protection of mobile agents by employing trusted channels to establish trustworthiness between the endpoints.

The project was developed in a team at Huawei, and from this work two other papers were produced in parallel:

- *"Trusted Sockets Layer: a TLS 1.3 based trusted channel protocol"* [18]
  Authors:
  - Arto Niemi
  - Vasile Adrian Bogdan Pop
  - Jan-Erik Ekberg
- *"Towards Secure Mobile Agents"* [19]
  Authors:
  - Vasile Adrian Bogdan Pop
  - Arto Niemi
  - Valentin Manea
  - Antti Rusanen
  - Jan-Erik Ekberg

Thereby, the aim of this project is to improve the security and interoperability of applications inside IoT and mobile devices, and to ease the adoption of these solutions in the cloud computing services. Which, is in line with the main objective of the Confidential Computing Consortium (CCC) [20], a project community at the Linux Foundation.

1.1.1 Personal Contributions

As for my contribution, I participated in the project by working alongside the team. Moreover, my personal contributions can be presented by grouping them into five main endeavors:

- taking an existing size-optimized TLS library and implementing the *trusted channel protocol* as described in the first paper produced;
- participating in the design of the migration scheme described in the second paper and this thesis;
- adding a *secure local communication* between the SEE and untrusted environment to support the establishment of a *trusted channel* among SEEs on different devices;
- implementing a PoC in which the migration scheme is followed and working alongside the Huawei team contributing to the components of the secure enclave framework;
- measuring the performance of the resulting secure migration scheme.

Differently from the two papers, where either the trusted channel or the whole secure enclave framework are addressed, this thesis has a narrower focus on the design and implementation of the secure migration scheme.

In the following sections of the Thesis, firstly the basics of information security and cryptography will be discussed. And together with these, a brief explanation of hardware-based security and secure transfer protocols will be provided. Then, an overview of the current state of the art of secure execution environments and mobile agents will be presented. After the context-relevant information, the Thesis will continue by specifying the architecture of the secure enclave framework and the design of the migration scheme. Finally, it will conclude with the description of the

PoC implementation and the benchmarks related to the secure migration of mobile agents.

2 Background

In this chapter, the basics of information security and cryptography will be illustrated with a focus on the topics required to comprehend the design and the implementation of secure enclaves in the thesis. The chapter will start with the principal objectives of information security and the cryptographic primitives employed to ensure them. Next, the necessary terminology and protocols for the comprehension of Secure Execution Environments and migration mechanisms will be described. The chapter will conclude with the programming languages used to develop the framework and the migration process of the thesis and the benefits they bring.

2.1 Information Security Objectives

The terms **confidentiality**, **integrity**, and **availability**, also known as the **CIA triad**, have been largely used as the fundamental elements of security controls in information systems [21] in both practice and academic literature. However, in recent years many attempts were made to extend the scope of the CIA triad by introducing additionals terms like **authenticity**, **non-repudiation**, and **trust**. They were minted to address the new emerging issues of information security [21].

2.1.1 Confidentiality

Confidentiality has its root grounded in the military mindset of maintaining a top-down authority and control over those that have access to information, on a need-

to-know basis [21]. Meaning that the information available at higher authorization levels had to be protected against its disclosure to lower authorization levels. And so, it can be defined as follows: *"Confidentiality is the prevention of the disclosure of secret or sensitive information to unauthorized users or entities"* [22]. Confidentiality of code and data can be accomplished when there is no action, or set of actions, that an untrusted entity can make to directly read, or otherwise deduce, contents of the confidential code or data [23].

2.1.2 Integrity

Integrity can be defined as *"the prevention of unauthorized modification of protected information without detection"* [22]. The integrity of code and data can be ensured if there is no action, or set of actions, which allow an untrusted entity to modify the protected code or data without detection [23]. An example of an attack threatening the integrity of the information is the Man-in-the-middle attack (MITM) where a malicious party is interposed between the communication of two legitimate entities and, in addition to eavesdropping on the communication, alters the information.

2.1.3 Availability

"Availability is the provision of services and systems to legitimate users when requested or needed" [22]. This means, that the availability of the code and data can be ensured if there is no way for an attacker to deny service to the users of the system [23]. For instance, the goal of a Denial of Service attack (DoS) is to compromise the availability of a service by different means, from saturating the bandwidth of the server, and so, the communication channel, to crashing the server itself.

2.1.4 Authenticity

Providing authenticity *"is of great importance to ensure the genuineness of physical or electronic documents, communications, transactions, and data"* [24]. This objective can be reached by proving that the entities involved in a communication are who they claim to be. Authenticity is strictly related to integrity, insomuch that the US Code [25] includes the concept of authenticity in the definition of integrity due to their strong relationship. For instance, guaranteeing the authenticity of data that was illegitimately modified does not bring any benefit since the information differs from the original one, thus it may be useless or even harmful for the system.

2.1.5 Non-Repudiation

Non-repudiation is *"the implication that one party in a transaction is not allowed to deny having received a transaction, nor, conversely, is the other party permitted to deny having sent the transaction"* [24]. The last-mentioned property is fundamental for example in financial transactions, where the occurrence of a payment already happened should not be possible to deny. Non-repudiation is strictly related to integrity too, a payment should not only be infeasible to deny but should also be infeasible to modify once it took place.

2.1.6 Trust & trustworthiness

Although there is a subtle difference between *trust* and *trustworthy*, they cannot be considered as interchangeable terms in information security because it will lead to major security issues otherwise. The term "trust" was introduced to express the need for privacy and reliability in information security. *"Trust is a choice one makes about another system"* [26], meaning that, since "trust" is a choice, there is no explicit proof that the trusted component will behave in the way it should, but we want to trust that it will do it anyway.

Trustworthy, on the other hand, is not only a choice but is the quality of an entity to be trusted and to give some assurance that the trust will not be betrayed. The guarantee can be given through some explicit proof stating the quality of an entity to be trustworthy, and thus, it can be used in the decision process of trusting the entity or not.

2.2 Cryptographic background

In the previous section, some of the objectives of information security were exposed but without any details on how to safeguard them. Cryptography provides us the means and defines the best known approach to ensure those objectives. According to Kerckhoffs's principle *"A cryptographic system should be secure even if everything about the system, except the key, is public knowledge"* [27]. In other words, a cryptosystem complies with Kerckhoff's principle when its security depends on the confidentiality of the key, but not on the confidentiality of the system itself. This last approach aims at avoiding the so-called security through obscurity, where the cryptosystem is secret and its disclosure can compromise every key ever used, nullifying any security protection ensured by the cryptosystem.

2.2.1 Cryptographic Primitives

Cryptographic primitives are a set of low-level algorithms that are used as building blocks to construct cryptographic protocols. Each of the various primitives is ensuring one or more specific objectives of the CIA triad and together with protocols and certificates, they are used to enforce the security of information systems.

2.2.1.1 Symmetric Key Cryptography

Symmetric Key Cryptography is used to provide confidentiality of the data through encryption. As can be deduced from the name, encryption and decryption are symmetric in the sense that the same key is used for both. The encryption mechanism can be defined as a **function** *Enc* that has as an input two elements, the **plaintext** p and the cryptographic **key** k, and has as an output the **ciphertext** c. Thus, the encryption process is equal to: $Enc(p, k) = c$; while the decryption process is the reverse: $Dec(c, k) = p$.

Ciphers are algorithms implementing the encryption and decryption of messages deterministically and can be mainly categorized into *Stream Ciphers* and *Block Ciphers*. These can be distinguished by looking at the procedure used by the algorithms to process the encryption/decryption. However, the procedures used in the two categories may also be combined. For instance, AES-GCM is a mode of operation that converts the AES block cipher into an authenticated encryption (AE) stream cipher [28]. Ciphers providing authenticated encryption are a key concept in modern cryptography, and they can ensure not only the confidentiality but also the authenticity of a message. The authenticity is provided by the inclusion of the *Message Authentication Code (MAC)* (Section 2.2.1.4) in the encryption. This can be done with three different approaches, encrypt-then-MAC, encrypt-and-MAC, and MAC-than-encrypt. Among the three different possibilities, the encrypt-than-MAC is the one that provides the highest level of security due to its ability to ensure the integrity for both the ciphertext and the plaintext [29].

Stream ciphers take as an input two streams of information, the plaintext, and the pseudorandom keystream, and execute a bitwise XOR operation to generate the ciphertext as an output. The keystream is generated by a pseudorandom number generator, that takes a seed (e.g the key k) known by both parties and produces a

deterministic stream of pseudorandom numbers. The randomness of the keystreams is crucial for the ciphers because they are the elements that determine the randomness of the resulting ciphertexts, thus, the ones that play a key role in avoiding any possible pattern that may leak some information on the encrypted data. Even though it is obsolete and labeled as vulnerable, the most known implementation of stream ciphers is *RC4* [30]. Despite this, *Salsa20* [31] and *ChaCha* [32] are the most used ciphers nowadays since they are still considered to be secure.

Block ciphers on the other hand, instead of using streams of information, are working on blocks of data. Depending on the specific algorithm used, they split the plaintext into fixed-size blocks (e.g. AES uses 128 bit as a block size), which are then encrypted one by one or in parallel. The method used to encrypt the data-blocks depends on the *mode of operation* used. The latter defines how the block cipher algorithm is repeatedly applied to all the data blocks forming the plaintext.

There exist various modes of operation, each one with specific advantages and disadvantages. The simplest one is the *Electronic Code Book* (ECB) [33], where each block of data is encrypted independently with the same key. In this way, the encryption and decryption processes can be faster thanks to parallelization. However, this mode is very weak from a security point of view because a plaintext block will be always mapped to the same ciphertext. Meaning that the encrypted data will leak existing patterns in the plaintext. A more secure mode of operation is the *Cipher Block Chaining* (CBC) [33]. Here, before the encryption, each plaintext block is XORed with the previously encrypted block. This operation solves the pattern leaking problem but makes parallelization infeasible, slowing down the overall performance of the cipher. The *Galois/Counter Mode* of operation (GCM) [28] is similar to the *Counter Mode* (CTR) [33] and tries to solve both the problems in the previous modes of operation. This is possible thanks to the usage of a counter, which is encrypted with a block cipher and then XORed with the plaintext. One

of the characteristics of the GCM is that in addition to the ciphertext, it provides also authentication on the encrypted messages through a particular mechanism of hashing, chaining, and encrypting the resulting ciphertext [28]. This mechanism provides the so-called authenticated encryption described in the above "**Ciphers**" paragraph (Section 2.2.1.1).

The most popular block cipher algorithm is the Advanced Encryption Standard (AES) also know with its original name Rijndael [34] after its inventors. The algorithm selected for AES was decided in 2001 by NIST but among the various participants the SERPENT [35] cipher was the best one from a security point of view [36]. The reason why the Rijndael algorithm was selected to be standardized is due to its better performance in terms of speed even if it was weaker than SERPENT [36]. Twenty years after its standardization, AES is still the most used algorithm and deemed secure in practice. However, some attacks do exist. For instance, with the *biclique cryptanalysis* attack [37] it is possible to reduce the computational complexity required to recover the key. Another example is the use of quantum computing, which is able to break the 128-bit version of the cipher [38]. Despite this, AES with 256-bits is still assumed to be resistant to quantum computing attacks.

The just mentioned block ciphers are mostly used when the systems are not resource-restricted because they require heavy computational operations to be executed. As a consequence, when the resources are limited, e.g. on mobile and IoT devices, other *"lightweight"* block ciphers, such as PRESENT [39], are sometimes employed.

2.2.1.2 Asymmetric Key Cryptography

The Asymmetric Key Cryptography, called also *Public-Key Cryptography*, can be used to provide, (i) confidentiality by encrypting the message, (ii) integrity by signing it (Section 2.2.1.5), and/or a symmetric key via a key agreement protocol. In

contrast to symmetric key cryptography, where just one key is enough to establish a secure communication channel, asymmetric key cryptography requires both the parties to have a pair of keys, the **Secret key** *sk* (also called **Private key**) and the **Public key** *pk*. The pair of keys are mathematically related, such that the public key can be easily derived from the private key, but not the other way around. This means that it should be computationally infeasible to find out the value of the private key *sk* given the corresponding public key *pk*. The final property is fundamental for public-key cryptography because to make it functional, the *pk* needs to be distributed to all the participants in the communication system. For instance, the distribution can take place through *public-key certificates*, as discussed in Section 2.2.2. In contrast, the secret key must be known only by the entity owning the pair of keys, otherwise, the keys are compromised. The encryption process is similar to the one used in symmetric cryptography, the difference lays in the keys used to perform the encryption and decryption operations. When encrypting a message, the sender needs to use the public key of the receiver: $Enc(p, pk_{receiver})$; meanwhile, the receiver needs to use its secret key to decrypt the message: $Dec(p, sk_{receiver})$.

Public-key cryptography is extremely important especially in untrusted environments like the Internet. It gives the possibility to the holders of the recipient's public key to establish a secure communication channel over the network without the need of exchanging the symmetric key on an already established secure channel. Although it is possible to use just public-key cryptography to communicate over a secure channel e.g. with RSA-DOAEP [40], its performance is relatively slower than symmetric key cryptography. Thus, it is usually used only to exchange a key for symmetric encryption, which, will be used for further communication. The last-mentioned process can be called *Key Exchange* or *Key Encapsulation Mechanism* based on the algorithm used for retrieving the symmetric key. For instance, Diffie-Hellman (DH) [41] is the first publicly known key exchange algorithm and it

is widely used in the authenticated key exchange (AKE) mechanism especially in its version based on elliptic curves (ECDH) [42].

The most known and used algorithm in public-key cryptography is the RSA [43] algorithm, whose security is based on the factorization of two large prime numbers and the RSA problem [43]. The latter are mathematical problems for which there are no algorithms running on traditional systems that can efficiently solve them, so it is considered infeasible to retrieve the secret key from the public key. However, with the advances in quantum computing and more optimized algorithms, the difficulty to factorize a 2048 bit RSA integer has dropped significantly in 2021 [44]. Nowadays, quantum computers are threatening the security of all the algorithms that are basing their security on discrete logarithm, elliptic curve discrete logarithm, and factorization problems. To address this imminent problem, the NIST is already working on the standardization of a *Post-Quantum Cryptography* [45], which will be based on mathematical problems that are not vulnerable to quantum computing.

2.2.1.3 Cryptographic Hash Functions

Hash functions are cryptographic primitives capable of guaranteeing the integrity of the information on which they are applied. A hash function can be seen as a cumputationally efficient function $Hash$ that takes as an input a string of data of aribitrary length m, and maps it to a fixed length string called hash h: $Hash(m) = h$. A secure hash function is one-way i.e. should not be possible to reconstruct the original string from the hash string. Moreover, a cryptographic hash function should have also the following properties [23]:

- **Preimage resistance**: given a hash value h should be computationally infeasible to find any string m such taht $h = Hash(m)$
- **Second preimage resistance**: given a specific string m_1 it should be computationally infeasible to find a different string m_2 such that $Hash(m_1) =$

$Hash(m_2)$

- **Collision resistance**: it should be computationally infeasible to find two random strings m_1 and m_2, where $m_1 \neq m_2$ such that $Hash(m_1) = Hash(m_2)$

What makes hash functions ideal for checking the integrity of the data, is their ability to map two very similar strings to two completely different hash values that can be easily stored thanks to their small and fixed size. Among the various algorithms implementing a hash function, the most used nowadays is the SHA-2 hash function even though recently the SHA-3 (KECCAK) [46] hash function was standardized. The SHA-2 is not superseded by SHA-3 even if there exist preimage attacks on reduced versions of the algorithm [47][48]. That is because the full version of the SHA-2 is not vulnerable but it may be in the future. The SHA-3 instead, should be unaffected by the security flaws present in the previous SHA algorithms thanks to a different internal design. And this is explainable by the fact the SHA-3 is based on permutations [46], while the SHA-2 is based [49] on an underlying block cipher and the Merkle-Damgård construction method [50].

2.2.1.4 Message Authentication Code

MACs are similar to hash functions but they not only ensure the integrity of the string of data but prove also the authenticity of the information. Similarly to symmetric key encryption, when a receiver needs to verify a MAC, it will use the same key the sender used to create the MAC. For instance, the *Hashed MAC (HMAC)* is a particular type of MAC that employes a cryptographic hash function and a secret key to compute the digest used to verify the integrity and the authenticity of a message. Another example is the *Cipher-Based MAC (CMAC)*, which instead of a hash function, employs a block cipher and a secret key to compute the verifiable digest.

2.2.1.5 Digital Signatures

Digital signatures, called also public-key signatures, are another mechanism to safeguard the integrity and the authenticity of data. However, unlike MACs, they do not use the same key to sign and check the data but they use public-key cryptography instead. As already mentioned, the efficiency of public-key cryptography is much lower than the symmetric one, thus, the hash-and-sign paradigm is applied [51]. Indeed, if a signature scheme is provided for messages of a fixed length l and a longer message m needs to be signed. Rather than signing m itself, it is possible to apply a hash function $Hash(m)$ to reduce the message to the fixed-length l and then sign the result with the sender's secret key sk. As a result, the hash-and-sign paradigm gives the possibility to have a signature scheme for arbitrary-length messages [51]. Meanwhile, to check the digital signature, the receiver has to verify the digest with the sender's public key pk and recompute the hash of the data in order to make sure that the two values are equal. In both cases, Digital Signatures and MACs, the integrity is provided by the hash of the data and the authenticity by the encryption of the hash. Hence, even if an attacker will be able to intercept and modify the data, he will be able to compute also the hash of the new data but will not be able to generate the corresponding MAC or Digital Signature.

2.2.1.6 Issues in Cryptography

As already mentioned, quantum computing is threatening the long-term security of some of the most used cryptographic algorithms nowadays, and Post-Quantum cryptography is trying to address this issue. However, there are other issues as well, such as the difficulty of obtaining true random numbers and side-channel attacks.

The problem of generating random numbers especially applies to mobile and IoT devices due to their scarce hardware resources, but it is present in more powerful systems too. The main issue is the difficulty of implementing a perfect source of

entropy for the random number generator, like a flip of a coin. As a consequence, systems usually adopt pseudorandom number generators whose outcome may seem random, but is deterministic and derived from a random seed. The lack of randomness in the generated numbers can lead to security issues like thousands of entities having the same key or the generation of a weak one [52][53].

Side-channel attacks are another issue that is frequently encountered in cryptography. These can take advantage of hardware-specific or implementation-specific behavior to recover the secret key even when the cryptographic algorithm is working perfectly. Some examples of the effectiveness of this kind of attack, are the key extraction of a 4096-bit RSA secret key from the GnuPG software through acoustic channels [54] and the recovery of the ECDSA private key in Mozilla's Network Security Services through ElctroMagnetic Analysis [55]. For instance, the final one needs as few as 30 signatures to extract the whole private key of the signer and it can be applied also to other algorithms like DSA and RSA [55].

2.2.2 Public Key Infrastructure

The *Public Key Infrastructure* (PKI) is essential to guarantee the security of public-key cryptography. In practice, it provides sets of policies and protocols to manage the certificate used in asymmetric cryptography. As such, the PKI can be defined as a tool that is *"intended to provide mechanisms to ensure trusted relationships are established and maintained. The specific security functions in which a PKI can provide foundation are confidentiality, integrity, non-repudiation, and authentication"* [56]. The problem that PKI is trying to solve is the need for trust in the binding between a public key and its owner. PKI makes use of public-key *Digital Certificates* i.e. those defined by the *X.509* standard [57] and third-party trusted authorities to achieve this goal. The certificate contains all the necessary information to identify the entity owning the key pair and the public key itself. This information is usually verified

by a Certificate Authority (CA), which will further sign the Digital Certificate with its private key. Therefore, it authenticates the document and makes it trustworthy for other participants in the communication system, who trust the same CA. The signed certificates can be validated with the public keys of the CAs, which, usually, are well-known and pre-distributed. As a result, CAs are binding the identity of the owner to the specific public key, so, given that the CAs are trusted, also the binding will be implicitly trusted. However, in order to be valid, a certificate does not necessarily need to be signed by a Certificate Authority, it can be signed even by the entity in the certificate itself. But the problem emerging when the certificate is self-signed is that it cannot be trusted since no trusted entity signed it.

2.2.3 Remote Attestation

Remote attestation can be defined as *a process that allows an entity to create and send verifiable claims about its state to a remote third party.* There are multiple options to integrate remote attestation through a cryptographic protocol, however, regardless of the implementation, some entities need to be involved in the protocol. As illustrated in Figure 2.1, there is always a **Relying Party** or **Appraiser** which needs to make a decision about some other party or parties [58]. A **Target** which is the entity the decision needs to be taken about [58], and it means that this will be the party that has to be attested. The last one is the **Attester** which is the party that is going to collect all the claims about the target and produce the *attestation evidence.* The attestation evidence is the proof containing all the verifiable claims for the remote attestation and can be presented to the appraiser either by the attester or the target [26]. The attestation claims are produced through multiple measurements about the target, and they are signed by the **Root of Trust**. The latter is a module with known behavior, which a certificate asserts to be present on a particular platform [58]. The certificate of the module can be signed by its

manufacturer's CA and the CA in turn by a well-known web CA like *GlobalTrust*. This is going to establish a *chain of trust* that can be used to obtain trusted claims about the measurements of the target in the attestation process.

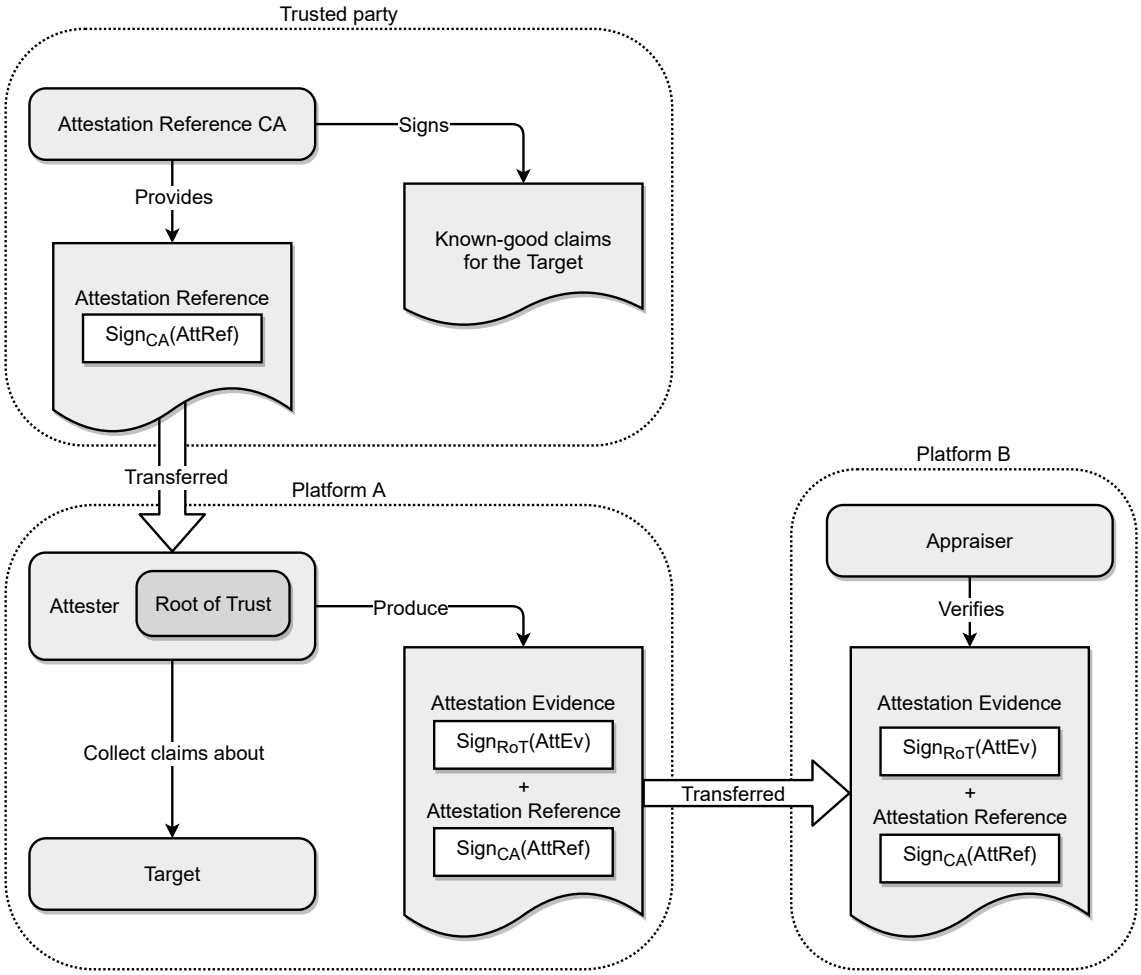

Figure 2.1: Remote attestation process with validation against CA signed attestation reference.

The appraiser, in order to validate the measurements about the target, needs to compare them against some known-good values for the state of the target. These values are known as *attestation reference* and can be provided by a third *trusted party*. For instance, one possibility is that an *attestation reference CA* (trusted by both parties through a certificate) provides to the attester the attestation reference

signed with the CA private key. As a consequence, the attester now can include the references with each attestation evidence in order to provide also the known-good values to the appraiser.

2.2.4 Secure Channel Protocols

A protocol is defining *"a set of rules or procedures for transmitting data between electronic devices"* [59] over a communication channel. Standardizing protocols is essential for communication between devices because otherwise, the communicating parties do not know how to exchange the messages. Therefore, many protocols exist, each one ensuring specific features and security properties.

The secure channel protocol is crucial in guaranteeing the security of communication on the Internet *"in a way that is designed to prevent eavesdropping, tampering, and message forgery"* [6]. A secure channel ensures data integrity, data confidentiality, authentication of the communication endpoints, replay protection, and key confirmation on a communication channel.

2.2.4.1 Transport Layer Security (TLS)

The most important secure channel protocol is TLS. TLS 1.3 is the latest, most secure, and the recommended version of the protocol, thus, it will be the version to which the TLS term refers in this thesis. The TLS protocol consists of three sub-protocols, spaced into two layers, the *Record Layer* and the *Handshake layer*. The first one is made by the *Record* protocol and is managing the messages for the handshake layer and the application data. The record protocol provides fragmentation, authenticated encryption, replay protection, and integrity check operations. The second layer instead, is formed by two subprotocols, the *Handshake* and the *Alert* protocols.

The Handshake protocol is used in the initial phase of the communication. It

defines the cryptographic parameters to be used and allows the authenticated key agreement (known also as AKE) between the entities via the use of certificates and PKI. It also supports PSK-based handshakes, where no public-key authentication is performed. The Handshake protocol in TLS 1.3 and illustrated in Figure 2.2, has fewer steps than the previous versions of TLS and it works as follows:

1. At time 0-RTT (see Figure 2.2) the **Client** sends in the *"ClientHello"* message, a $nonce_c$ (a random value to provide freshness), the supported cipher suites, and a $keyshare_c$ trying to guess the key exchange group (e.g. P-256 or X25519) the Server might use.

2. At time 0.5-RTT the **Server** selects a key exchange group offered by the Client (or sends a *HelloRetryRequest* message). Then, it generates a key pair in that group and sends in its *ServerHello* message the $nonce_s$ and the public key as its $keyshare_s$ to the Client. The shared secret computed with the two key shares is used to derive the encryption keys for all the messages sent after the *ServerHello*. Some of these are the optional *"CertificateRequest"*, the *"Certificate"*, and the *"Finished"* messages. The finished message contains a hash of all the previous messages, and once it is sent, the Server can compute the application data keys, which will be used for further communication.

3. At time 1-RTT the **Client** computes the shared secret and the encryption keys, checks the certificate of the Server, generates the application data keys, and if required, sends back its own *"Certificate"*. Once also the Client sends the *"Finished"* message, the handshake is concluded for the Client and it can exchange application data over the secure channel.

4. At time 1.5-RTT the **Server** verifies the *"Certificate"* of the Client if it was previously requested. Then, upon receipt of the *"Finished"* message from the Client, the Server also considers the handshake finished and starts exchanging

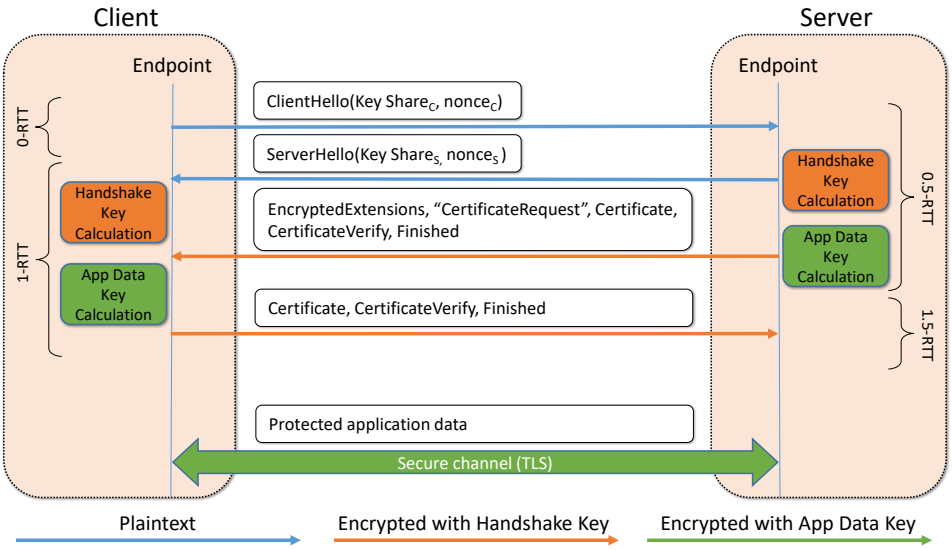

Figure 2.2: Handshake process in the Transport Layer Security protocol.

application data over the secure channel.

In contrast to the previous versions of the TLS protocol, TLS 1.3 has significantly simplified the selection of the cryptographic algorithms to be used in the communication channel. Firstly the number of supported cipher suites has been significantly reduced, both because some algorithms have become vulnerable, and because such a vast choice led to the unsafe selection of the algorithm to combine. Then, instead of using the MAC-than-encrypt construct for data integrity and confidentiality (Section 2.2.1.1) like in TLS 1.2 the version 1.3 allows only *Authenticated encryption with associated data (AEAD)* cipher modes [6]. The AEAD approach is similar to the encrypt-than-MAC way but simplified. Indeed, instead of requiring the choice of two algorithms, one for the cipher and one for the mac, it requires the selection of a single AEAD cipher mode that includes both.

2.2.4.2 Trusted Channel

A secure channel, like the one empowered by the TLS protocol, provides protection against attacks on the data *in-transit* and authentication of the endpoints. However, even though there are guarantees in place that the data cannot be tampered or eavesdropped during its transfer, there are no guarantees about the trustworthiness of the endpoints themselves. Hence, they may be compromised and may e.g. leak all the secret information to unauthorized third parties. To overcome this issue, remote attestation can be used to establish trust in the endpoints of the channel. Thus, by adding mutual remote attestation to a secure channel (e.g. TLS 1.3) it can be turned into a *trusted channel protocol*. Here, the endpoints have the guarantee that none of the parties participating in the trusted channel establishment is compromised.

One example of such a protocol is the TLS 1.3 based *Trusted Sockets Layer* (TSL) protocol presented by Niemi et al. [18]. It extends the certificate-based authentication of TLS 1.3 with mutual attestation by using just the callbacks provided by traditional TLS libraries. As such, TSL is fully compliant with the RFC 5280 [60] and the TLS 1.3 specifications, therefore it does not require any change to the implementation of the existing TLS libraries.

2.3 Terminology

2.3.1 Trusted Computing

Trusted Computing is addressing the problem of having trust in a system that is processing the data. This can be done by using hardware-based techniques to isolate critical data and its processing from the rest of the system. In an email to every full-time employee dated in 2002, Bill Gates defined Trustworthy Computing as *"computing that is as available, reliable and secure as electricity, water services and telephony"* [61]. Hence, trusted computing's goal is to have a system that behaves

exactly how it is intended to. Trusted computing aims to give the system the ability to issue reliable evidence on the state of the system, assuring that the system is trustworthy.

2.3.2 Trusted Computing Base

The Trusted Computing Base (TCB) is the composition of all the hardware and software components upon which the correct operation of a system strictly depends. The TCB components are assumed to be trusted and that behave as expected even without explicit proof. This means that the TCB should be bug and vulnerability free because by compromising even a single component of the TCB, the protection of the whole system can no longer be assumed [23]. For instance, if the operating system is part of the TCB and it becomes compromised, it can maliciously manipulate all the applications running on top of it nullifying all the security protections of the individual applications. To obtain a vulnerability-free system, it is extremely important to intensively review and check the correctness of the design and the code implementation. However, it is usually infeasible to ensure that there are no security flaws in a system, and the bigger the system is, the more this claim is true. That is because the probability of bugs increases with the size of the system. For this reason, it is recommended to have smaller TCBs in order to be able to ensure a higher degree of security.

2.3.3 Secure/Trusted Execution Environment

A Secure Execution Environment (SEE), or also known as a Trusted Execution Environment (TEE), allows an application to operate without interference from untrusted code running on the same system. The SEE is created by a set of all the components in the TCB and is strictly dependent on the correct implementation of the protection measures of the TCB [23]. As a result, if the TCB is small,

e.g. relying only on the CPU, the application inside an SEE cannot be tampered with by other applications running on the system, nor by the operating system, by the hypervisor, and by the user itself. In such a way, the application executing into the SEE is protected even from components with higher privileges than the application itself. The TCB is able to guarantee confidentiality and integrity to SEEs through cryptographic functions and isolation. The confidentiality of the SEE can be protected by encrypting the memory used or by isolating the memory and the cache from the rest of the system through memory management mechanisms into the TCB. Meanwhile, the integrity of the SEE can be verified by hashing everything that has to leave the TCB, thus ensuring that even if encrypted, nobody can modify the data without being detected[23]. There are several implementations of SEEs but the most popular ones are ARM TrustZone [62] (see Section 3.2.3.1) for mobile and IoT devices, and Intel SGX [14] (see Section 3.2.3.2) or AMD SEV [15] (see Section 3.2.3.3) for PCs and cloud environments.

2.3.4 Secure Enclave

A secure enclave can be defined as an isolated process, executed on a platform that provides confidentiality and integrity of code and data against the untrusted code running on the same system as well as sealing and attestation [63]. It is a particular implementation of an SEE, where the application inside is protected from all the other software on the platform. Its execution cannot be altered even by higher privileged code such as SMM, BIOS, VMM, OS, etc [64]. Moreover, via remote attestation, a remote party can obtain strong assurance about the precise software running in the enclave and its trustworthiness [65].

2.4 Secure Technologies

For the development of the secure enclave framework, the project team chose to employ the technologies that are considered the best from a security and interoperability point of view. *Rust* and *WebAssembly* are examples since they can bring additional values to the project compared to similar technologies such as the C programming language or the Java Virtual Machine.

2.4.1 Rust

Rust is a system programming language designed to give deep control over the running time and memory usage to developers. As with other languages like C and C++, it can be compiled directly onto the bytecode executed by hardware, thus providing excellent performance as well [66]. However, in contrast to languages like C++, Rust offers also strong safety guarantees with almost no data races, and memory errors like buffer overflows, stack overflows, and allocation/deallocation memory issues [66]. Indeed, in Rust, there is the risk to introduce memory-safety bugs only when *unsafe code* is explicitly used by the developer i.e. with the *unsafe* marker. Otherwise, by assuming that the compiler is bug-free, the code should always be protected by unintentional and intentional memory-safety bugs. All this is possible thanks to Rust's restricted *ownership* model. Here, there is a unique reference to each live object in memory and this allows to statically track the lifetime of an object at compile time (this property is broken by the unsafe marker). In this way, there is no risk in dereferencing allocated memory, which makes Rust a *memory-safe programming language* without the need to have a garbage collector at runtime that frees the allocated memory [67]. The effectiveness of Rust's approach to memory-safe programming can be seen in a recent study on the memory vulnerabilities found until 2021 [68]. The majority of the memory bugs found are only mild issues because they only introduce the risk of violating the memory-safety design of Rust by using

unsoundness libraries in the code [68].

Using the Rust programming language to achieve memory safe enclaves has already been addressed with the Rust-SGX [69] project. However, it is an implementation that is built upon the SGX SDK, thus, it is available only for Intel SGX enclaves.

2.4.2 WebAssembly

To briefly introduce WebAssembly (Wasm) [70], it can be seen as *"a binary format for software written in any language, designed to eventually run on any platform without changes"* [71]. This unique characteristic allows the compilation of Rust code into WebAssembly bytecode. The latter can be executed inside a WebAssembly Interpreter (WASI) making it runnable on any system independently from the architecture. Furthermore, WebAssembly is designed to provide also a secure execution environment thanks to its ability to sandbox the bytecode from the system where it is running. However, these security statements have been challenged by recent analyses, which are stating that Wasm still has some security issues [72]. Indeed, due to its design and the obfuscation of code inherently brought with the abstraction, many of the vulnerabilities already solved in the past, such as stack overflows, are becoming a threat again [72]. Fortunately, mitigations for the presented vulnerabilities already exist, and most of the issues related to memory safety can be addressed at compile-time by memory-safe programming languages like Rust [73]. Finally, the abstraction from the machine where the code is executing gives to WebAssembly applications not only a higher degree of security but also code mobility properties. Thus, it makes it possible to migrate the mobile code between different devices and platforms independently from the hardware architecture, which is ideal for mobile agent solutions.

3 Prior Art

3.1 Distributed Computing

At their inception, computer systems were conceived as monolithic computing devices but soon they evolved into much more flexible client-server systems able to serve a wider range of purposes. However, the high computational power required by some applications nowadays motivated the rise of even more complex architectures such as distributed computing systems. The latter consists of nodes that are performing heterogeneous computations and communicate only occasionally to achieve a common objective [74]. At the beginning of the '90s, distributed computing systems were a topic of great importance in computer science, and technologies supporting the development of distributed applications already existed [75]. For instance, the CORBA object model was providing an architecture for the implementation of distributed applications where the client-server communication was supported by the Object Request Broker [76]. Here, the network interfaces were completely abstracted from the applications and clients were able to ask for tasks to be done by servers without any knowledge about the server's location or mode of operation. Its first release was supporting only the C programming language but additional ones were supported in later versions of CORBA [76].

Although distributed systems were well known, the concept of *code mobility* emerged only with the introduction of the *Java Virtual Machine* (JVM). The latter,

together with the web browser technology, came up with the possibility of running executables on any platform independently from the hardware architecture. In contrast to previous implementations, this brought in the possibility to exchange not only data messages or objects over the network, but also program code [77]. As mentioned by Fong et al. [77] here it is wise to make a distinction between code with *strong mobility* (also known as *mobile agents*) and code with *weak mobility* (known as *mobile code*).

3.1.1 Mobile Code

As previously mentioned, a great advantage of mobile code, as opposed to traditional executable code, is its ability to move and be run on any platform. However, once the mobile code arrives at the hosting destination, its execution is started and completed only on that specific host. Therefore, mobile code will have an outcome that can be used directly by the hosting application or sent back to the source of the mobile code for further operations. The movement of code in this specific mode of operation can be called *stateless migration* because it does not require any particular state of the mobile code to be transferred for its correct execution. Carzaniga et al. [78] recognized two different approaches in the application of mobile code, *Code on Demand* (COD) and *Remote Evaluation* (REV).

3.1.1.1 Code on Demand

In the COD approach, the client requests from the server the code to be executed, the server sends the mobile code and the client executes it obtaining the desired result. At its inception with Java Applets [79] and now with JavaScript [80], this methodology is widely spread and used in any browser. Indeed, JavaScript is used in 95% of the websites nowadays, meaning that is the most used type of software worldwide [81].

3.1.1.2 Remote Evaluation

Remote Evaluation can be seen as the opposite of Code on Demand. Indeed, here it is the client that migrates the mobile code towards the server to be executed. One of the simplest examples of this methodology is the *rsh* command [82] that enables the execution of commands into remote shells. A more recent example is Amazon's serverless compute service, *AWS Lambda* [83], which provides the ability to migrate your own mobile code to Amazon's servers to be executed. For instance, this procedure comes in handy when resource-constrained devices require high demanding computational functions to be executed.

3.1.2 Mobile Agents

In contrast to mobile code, mobile agents have strong mobility features, or in other words, come with a *stateful migration* of the mobile code. As a consequence, instead of initiating the execution of the code on the host where it is migrated, it can resume its execution from the state where it was paused in the previous host. Therefore, this ability means that the mobile agents are not restricted to client-server based communication and allows them to migrate between multiple devices to execute the desired operations. Outtagarts et al. [84] have identified multiple mobile agent based applications in different domains, such as network management, wireless multimedia sensors, electronic commerce, distributed data mining, security, and many more. For all these applications, different platforms are available for their implementation. Usually, each of the platforms serve the development of mobile agents aiming at specific goals. For instance *PIAX* [85] integrates the P2P network as a mean for the development of mobile agents in distributed environments, *SensorWave* [86] aims at enhancing wireless sensor networks with mobile agents, and *TACOMA* [87] was developed to provide system support for farmworkers [84]. Many other mobile agent platforms exist [88], but among all, the *Java Agent DEvelopment Framework*

(JADE) [4] turned out to be the most popular one in the academic and industrial community [88].

Despite the increasing adoption of mobile agents in distributed systems and mobile cloud computing, there are still several security issues to be addressed. For instance, JADE, the most adopted platform, does not include any security protection by default, and even with *JADE Security* Add-On there are still major security lacks that need to be resolved to achieve data confidentiality [89].

3.2 Secure Execution Environments & Secure Enclaves

Secure Execution Environments have existed for more than a decade and in all these years various possible implementations were adopted. The multiple solutions can be categorized in three main categories: *External security co-processor*, *Embedded security co-processor*, and *Processor secure environment* [90]. A high-level architecture is presented in Figure 3.1 for each of the just mentioned categories. The Figure shows which hardware components are involved in the Secure Execution Environments or the additional components required. As shown in the legend, the fully green components are fully trusted components meanwhile, the ones that are partially green are just partially trusted, and only the trusted section is employed in the SEE. Further, the following sections will describe in detail each category and the existing implementations.

3.2.1 External Security Co-Processors

This category of solutions provides dedicated Security Co-Processors that are placed outside the System on Chip (SoC) containing the CPU as depicted on the left side of Figure 3.1. As a such, this kind of solutions have their own memory and peripherals

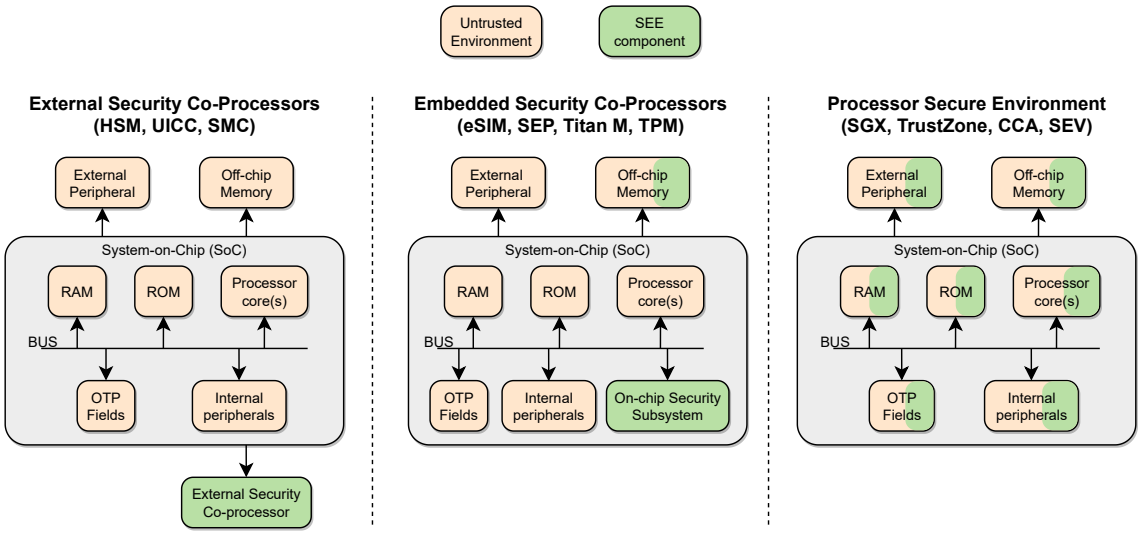

Figure 3.1: High level representation of the architectures in the different categories of Secure Execution Environments.

and in addition to this, they can also enforce higher security standards than the main device by deploying physical tamper-resistant mechanisms. Even though this kind of design is beneficial to the isolation of the Secure Execution Environment from the main system, they usually have less computational power, imply higher costs for the implementation, and are associated with higher overheads due to their location outside the SoC. These type of solutions are adopted when high-security standards are required. For instance, they can be employed by both, businesses to support payment transactions with HSM devices [91] or by customers to enable mobile payment with the use of Secure Elements (Section 3.2.1.2).

3.2.1.1 Hardware Security Modules (HSMs)

HSMs are peripheral devices or bus-connected to the host computer and are strictly dedicated to the execution of cryptographic functions such as encryption/decryption of data or providing a source of entropy for random number generators [92]. Furthermore, in order to be considered an HSM, a device needs to be physically

secure, tamper-resistant, and/or tamper-evident. This means that they are usually equipped with coatings and backup batteries to be able to detect any physical intrusion and erase all the secure information in their memory [23]. To ensure these properties, Hardware Security Modules have to undergo a special certification, the FIPS 140. The latter is a publication series for the standardization of cryptographic modules and usually certifies an HSM with one of the four levels of security defined in the standard. The vast majority of HSMs are now certified with the FIPS 140-2 standard [93], however, a new FIPS 140-3 standard [94] was just introduced and the the transition to the new certification has already begun [95]. The high level of security provided by HSMs is mostly used in critical operations, such as protecting transactions in POS devices and ATMs or ensuring the security of highly sensitive information on the web. Although these devices provide a high level of assurance, they cannot be always adopted due to their design. For instance, they always require an external device to be plugged in, which usually is cumbersome, thus, making them incompatible with mobile devices.

3.2.1.2 Removable Secure Elements

In contrast to the cumbersome external HSMs, Secure Elements provide integrated hardware and software embedded in a mobile handset. The latter is enabling three fundamental functionalities, a secure memory to store secrets and personal data, cryptographic functions for secure data exchange and encryption, and a secure environment for code execution [96]. A Secure Element can be implemented as a removable component in different solutions. The first option is to use the already existing Subscriber Identity Modules (SIMs) as a Secure Element for the SEEs. Indeed it already carries out this task by authenticating the user in the network of the operator [96]. A modern option is to use Universal Integrated Circuit Cards (UICC) which are able to host multiple Java Card-based software applications. This

particular solution has the advantage of interfacing with the mobile device but has the drawback that it is tightly bound to the provider of the card since all the applications must be authorized by the issuer security domain of the card. Another example of a removable Security Module is the Secure Memory Card (SMC) which is a combination of a memory card and a smartcard [97]. It can provide high security levels while being able to host a large number of applications due to the properties of the two combined components.

3.2.2 Embedded Security Co-Processors

In contrast to the previous category, embedded security co-processors do not require the users to acquire and insert additional hardware into the device to execute an application inside a Secure Execution Environment. Moreover, by being integrated with the CPU on the SoC (see Figure 3.1), it allows a greater share of resources while still providing a high level of isolation. Although this approach is more suitable for mobile devices, it still has significant constraints on the available resources, thus it may not be suitable for the majority of applications [90].

3.2.2.1 eSIM - eUICC

The eSIMs and eUICCs are embedded secure elements that are aiming to solve the problem of little space availability on wearable devices [98]. The problem is addressed by soldering directly on the mobile device the eSIM or eUICC, so, allowing the adoption of smaller hardware components. Since the soldered element cannot be substituted, the eSIM and eUICC are enabling also the remote provisioning of already issued SIM profiles.

3.2.2.2 Apple Security Enclave Processor (SEP)

According to Apple SEP's patent [12], *"SEP may be isolated from the rest of the SOC (e.g. one or more central processing units (CPUs) in the SOC, or application processors (APs) in the SOC)"*. This statement and the following ones in the document are giving us only little information on the actual implementation of SEP. The security enclave processor seems to be built upon a dedicated ARMv7-A *"Kingfisher"* core and to be strongly isolated from the AP [99]. In addition, SEP is provided with dedicated hardware peripherals and memory that can be accessed only by itself [99]. However, the memory of the security enclave processor is very limited (e.g. 4096 bytes) and needs to use the external RAM to be able to operate [99]. Therefore, it uses AES encryption to isolate the information stored on the external RAM from the main processor.

3.2.2.3 Google's Titan M chip

The Titan M chip is designed to provide strong physical isolation from the main processor. Its CPU is a separate ARM Cortex-M3 microprocessor and the cache, memory, and persistent storage are not shared with the rest of the system [11]. However, its resources are constrained, for instance, it tries to pack as many as possible security features in only 64 KBytes of RAM [11]. An important note on Google's Titan M chip is that it tries to avoid the so-called *"security through obscurity"*. As a matter of fact, the design and the firmware of Titan M are going to be disclosed to the public and only the root keys used to sign the Titan M firmware will be kept secret by Google [11].

3.2.2.4 Trusted Platform Modules (TPMs)

A Trusted Platform Module is very similar to a Secure Element, and in the words of Andrew Martin *"it may be helpful to consider the TPM as like a smartcard soldered*

to the PC's motherboard" [100]. Its main objective is to provide a root of trust to the platform, which consequently helps in protecting against software attacks by providing a sealed storage mechanism and local/remote attestation [23][100]. This solution was largely adopted in high-end desktop devices especially to provide the *Secure Boot* option. When the system is powered on, the Root of Trust for Measurement (RTM) is the first code to be executed and it will repetitively compute the hash of the new code loaded and will verify its correctness. This particular operation can be included in the storage sealing process, in this way, each time a new piece of code is loaded during startup, if its hash is not correct, it cannot decrypt the sealed memory due to the wrong resulting key.

3.2.3 Processor Secure Environments

The last category of hardware support for SEEs is the Processor Secure Environments. These are able to provide much more performance to the SEEs with respect to the other solutions because they are implemented directly on the main processor (see Figure 3.1). The isolation between the Secure Execution Environment and the rest of the software is relying on hardware logic able to control which resources the CPU can access in a certain state. Due to its particular design, this category of solutions is usually more cost-effective, but they are more subject to side-channel information leakage.

3.2.3.1 ARM TrustZone

Arm TrustZone [62], since its initial proposal in 2002 [13][64] has become the prevalent technology implementing TEEs in mobile environments and it is starting to be employed also in industrial control systems, servers, and IoT [101]. TrustZone provides hardware-based security for System on Chip architectures, which are widely used in mobile environments. The design principle used by TrustZone is to split the

Execution Environment into two separate worlds, each one with its own user and kernel space, cache, memory, I/O, and other resources [13]. The first one is named *Normal World* or *Rich Execution Environment* (REE) because here runs the typical OS of the device and all the untrusted applications. The second one, called *Secure World* (TEE), contains a secure small kernel that is running in supervisor mode. The latter provides the basic OS primitives to the *Trusted Application* (TA) running in user mode [101][13]. The *Secure Monitor* is underlying both the environments and it can be triggered by the two worlds with the *System Monitor Call*. The secure monitor is in charge of all the mechanisms for the secure context switching between worlds and protects the access to the *NS bit*, which identifies the currently executing world in the processor [101]. Due to its architecture design, TrustZone is able to ensure almost the same level of performance as normal environments, the only overhead is caused by the context switching between the two worlds. Despite its advantages in terms of performance, TrustZone has also various drawbacks that are limiting its adoption. In the first place, since the Secure World comprises both the kernel and the user space, it implies a size of the TCB relatively larger compared to other existing solutions such as Intel SGX. Then, the TA in the user space is strictly dependent on the trusted kernel, thus, reducing the interoperability of the various TAs between different TEEs. Furthermore, due to its design, this solution is not suitable for cloud services because it only enables a single secure world. Cloud services instead, need to instantiate multiple Secure Execution Environments on a single physical machine (e.g. through VMs or Secure Enclaves). Finally, considering that TrustZone is not encrypting the memory in use, it is also vulnerable to memory attacks such as Cold-boot [102] threatening the confidentiality of the application by someone able to physically access the device [103].

3.2.3.2 Intel Software Guard Extensions (SGX)

In 2015, Intel introduced the Software Guard Extensions technology with the Skylake processors, the 6th generation of Intel Core processors. SGX enables applications to create hardware-based Secure Execution Environments called SGX enclaves. In contrast to other existing SEEs, the Trusted Computing Base of Intel SGX is minimal and it is composed only of the processor hardware and software and the secure enclave itself. Thanks to its particular design, encryption mechanisms, and its essential TCB, Intel SGX guarantees the confidentiality and the integrity of applications running inside the enclave even in a fully compromised environment [64]. Since Intel SGX causes severe overloads due to the memory encryption of each enclave, it comes also with dedicated memory to improve its performance, called Enclave Page Cache (EPC). In the latest implementations of Intel SGX, the memory of the EPC is limited to 256MiB and it is storing the unencrypted pages of the enclave until it gets full, afterward, it encrypts the pages of the enclave and stores them in the main memory [17]. Even though this might be seen as a good solution, it has some inconveniences too. First of all, Intel SGX is available only for Intel processors, which are not widely used in mobile devices because it was mainly developed for cloud applications with the need for severe security requirements. Another issue with Intel SGX is that it causes a lot of overhead which lowers the performance of the application inside the secure enclave from 9.54 times up to 19.31 times compared to the normal environment [64]. Furthermore, there is also a major issue for the developers using Intel SGX. In order to make the software run inside the secure enclave, they have to split the software into two distinct sections, the trusted and untrusted one. The last-mentioned requirement is complicating the migration process on Intel SGX SEEs by a non-trivial amount [64]. Finally, despite the fact that this seems to be the most secure solution nowadays, also Intel SGX is vulnerable to some attacks, the most known ones are Meltdown [104] and Spectre [105], which, are

using side-channel attacks to exfiltrate confidential information from the enclave.

3.2.3.3 AMD Secure Encrypted Virtualization (SEV)

One year later, in 2016, AMD presented the Secure Encrypted Virtualization technology (SEV) [15], which, leverages a Platform Security Processor (PSP) and a memory encryption engine to provide hardware-based security to the system [106]. AMD SEV comes with a solution similar to ARM TrustZone for the implementation of SEEs but with the difference that SEV is specifically designed for cloud environments. In contrast to TrustZone, AMD SEV is designed to protect the whole Virtual Machine running upon the hypervisor and not only the Trusted Application inside the Secure World. This is possible because AMD SEV instantiates secure Virtual Machines as SEEs protecting them from physical threats, other VMs, and the VMM itself [64]. The last design is possible because the memory of each Virtual Machine is encrypted with its own key generated by the PSP, which provides also APIs to the hypervisor in order to manage the keys generated for the encrypted VMs. Even though encryption is usually a heavy operation, thanks to the memory encryption engine provided by SEV, its performance are identical to the normal VMs [64]. However, one of the issues in AMD SEV is that the whole Virtual Machine is considered as being part of the SEE, thus, the size of the TCB becomes really big, so, it increases the probability of having unexpected behaviors and security issues.

3.2.3.4 ARM Confidential Compute Architecture

The ARM Confidential Compute Architecture (CCA) [16] was unveiled this year 2021 with the new ARMv9 microprocessors. More specifically, it is implemented on the new ARMv9-A architecture and it aims at protecting the security and privacy of the data while it is in use. The ARM CCA is built on TrustZone as its foundation and is addressing the limitations of the latter technology. Indeed, TrustZone was

previously available just for silicon vendors and OEMs. Meanwhile, the CCA's ambition is to provide all the developers with the means to execute their applications inside SEEs [107]. To achieve this goal the CCA provides attestable isolated environments called *Realms*. The Realm environment does not substitute the TrustZone but it is designed to run alongside the Secure World and the Normal World. In fact, multiple Realms can be created on-demand by both, the applications running in the Normal World and the TrustZone. The latter is called *Arm Dynamic TrustZone Technology*, and enables TAs to perform memory-intensive operations, since TrustZone has a limited memory allocated at boot by the Secure Monitor.

The CCA architecture places the Realm in an isolated environment where even higher privileged components (e.g. hypervisor and TrustZone) cannot access its execution. The protection between these components is enforced by the Secure Monitor running at EL3. However, differently from the previous TrustZone architecture, the Secure Monitor manages the memory access through the Granule Protection Table, which is used to check authorization for memory accesses. Furthermore, each memory assigned to the Realm, Secure World, Normal World, or the Monitor, is encrypted by the hardware before being written to the DRAM [107].

The Realms are not executing directly on the Secure Monitor, but on the Realm Management Monitor (RMM). The RMM is in charge of managing the execution environment of the Realm in order to guarantee its confidentiality and integrity even between multiple Realms. However, the RMM is much smaller than a typical hypervisor since it relies on the Normal World's one for the (i) scheduling, (ii) resource allocation, (iii) interrupts management, and (iv) device emulation [108]. The control by the Normal World over these operations is enabled by the Realm Management Interface (RMI) provided by the RMM. This implies also that the RMM has no control over the execution of the Realm since the scheduling is performed by the Normal World, and so, no availability guarantee is ensured. Finally, as aforemen-

tioned, the Realms are attestable by third parties. This operation is enabled by the Realm Service Interface (RSI), which allows Realms to request attestations for the platform and the Realm itself.

3.2.3.5 Secure enclave frameworks

Keystone Keystone is an Open Framework for configuring, building, and instantiating customized TEEs [109]. Keystone's architecture can be seen as a mix of ARM TrustZone and Intel SGX architectures [110]. Indeed, it takes inspiration from ARM TrustZone, but thanks to the management of the memory in RISC-V, Keystone is capable of handling multiple enclave instances instead of having a single Secure World. This characteristic is enabled by the per-hardware-thread view of physical memory via machine-mode and Physical Memory Protection (PMP) registers in RISC-V [109]. Furthermore, the present design allows enclaves to access disjoint memory partitions while also opening up supervisor-mode and the Memory Management Unit (MMU) for further isolation required by the enclave use [109]. As a result Keystone's enclave provides higher flexibility by enabling both, the implementation of a lightweight or a full supervisor-mode OS.

Twine Twine is an Embedded Trusted Runtime designed to execute unmodified language-independent applications [17]. To accomplish this, Twine is leveraging WebAssembly's feature of sandboxing the software runtime in order to abstract the underlying environment from the application [17]. The abstraction gives the possibility to the software developers to write their application in any language without any constraint imposed by the underlying architecture. Thus, Twine is not a full implementation of an SEE but can be seen as a mean to allow the applications inside Twine to run independently from the SEE in which it is enclosed [17]. For the moment Twine is implemented only on Intel SGX but it has the potential to be implemented in other SEEs as well. Although its usage is theoretically possible on

different architectures, the authors of Twine have not provided details on the attestation and migration processes. Thus, the interoperability of the software between different SEEs may be complex and difficult to implement.

3.2.4 Existing Attacks

The following section gives a brief description of the existing attacks against SEEs. The focus of the attacks and the existing vulnerabilities are addressing mainly the Processor Secure Environments due to their large adoption and the security issues induced by the isolation at the main processor level. The attacks can be distinguished into three main categories based on the means they are using. The first ones are leveraging side-channels to exfiltrate secret data from the cache, memory, or execution processes, the second ones are aiming at modifying the state of the secure data with fault-injection attacks, meanwhile, the last one is exploiting the vulnerabilities in the TEE software code.

3.2.4.1 Side Channel attaks

A *side-channel* is a communication channel that was not intended or designed to transfer information between a sender and a receiver, and where the sender is not aware that he is disclosing information [23]. In other words, the data leaked is only a side effect due to the implementation of some functionality. The side channel can be built upon timing, electromagnetic emanations, power consumption, acoustic emanation, thermal emanations, and possibly other means [23].

This kind of threats, require an active attacker, which leverages timing side channels [111][112] to infer on secret directly from the execution of secure code [113], or from the data stored in the cache through attacks like *Prime + Probe* [114] and *Flush + Reload* [115]. Side channel attacks can be used also in combination with other techniques to exfiltrate information, some recent examples are the Meltdown

[104], Foreshadow [116], and Spectre [105] attacks. These are exploiting the vulnerabilities of speculative execution in modern processors to store secret data in cache and then infer on it with the use of side channel attacks.

3.2.4.2 Fault-injection attacks

Fault-injection attack is primary a physical attack on the device to inject the fault in the system deliberately to change its intended behavior and by such compromise the security algorithm/crypto in order to access the encrypted data. In order to achieve this goal, the attacker is using unconventional means whose original purpose are not to modify the data on the system. Some examples using these means are the *Rowhammer* [117] and the *CLKSCREW* [118] attacks, where physical properties of the system are exploited in order to manipulate the data and the code running on the system. For instance, Rowhammer is leveraging the implementation of the system's memory through capacitors. The latter cannot be perfect and need constant refreshes to maintain the charge in the cell indicating the bit stored. Rowhammer takes advantage of this physical imperfection and by accessing with specific patterns the adjacent cells, it can impact the charge on the target cell and flip its value.

These kinds of attacks can be used not only to damage the integrity of the code but also to defeat the availability of an entire system. For example, with the SGX-Bomb [119] attack it is possible to exploit the integrity protection mechanism of Intel SGX enclaves and perform a severe DoS attack on the system.

3.2.4.3 SEE Software implementation Attacks

As already mentioned in the TCB definition, having a large amount of code is increasing the probability of presenting bugs and vulnerabilities in the software. This behavior can be observed especially in the modern processor secure environments, where more code is needed to guarantee the isolation of the SEEs. These vulnera-

bilities can be introduced in both, the implementation of the SEE and the trusted application running inside the SEE. Some of the existing attacks are taking advantage of typical software vulnerabilities like the lack of input validation [120][121][122] [123]. Other attacks instead are using more sophisticated mechanisms like the confused deputy [124] or the rollback technique [125].

4 Architecture

In this chapter, we provide an overview of the secure enclave framework developed at Huawei, focusing on the aspects that are relevant to the migration scheme. We do not, for example, cover the Software Development Kit (SDK) in detail. While the SDK is essential for end-users of the framework, it is still under development and its full description is left for future work.

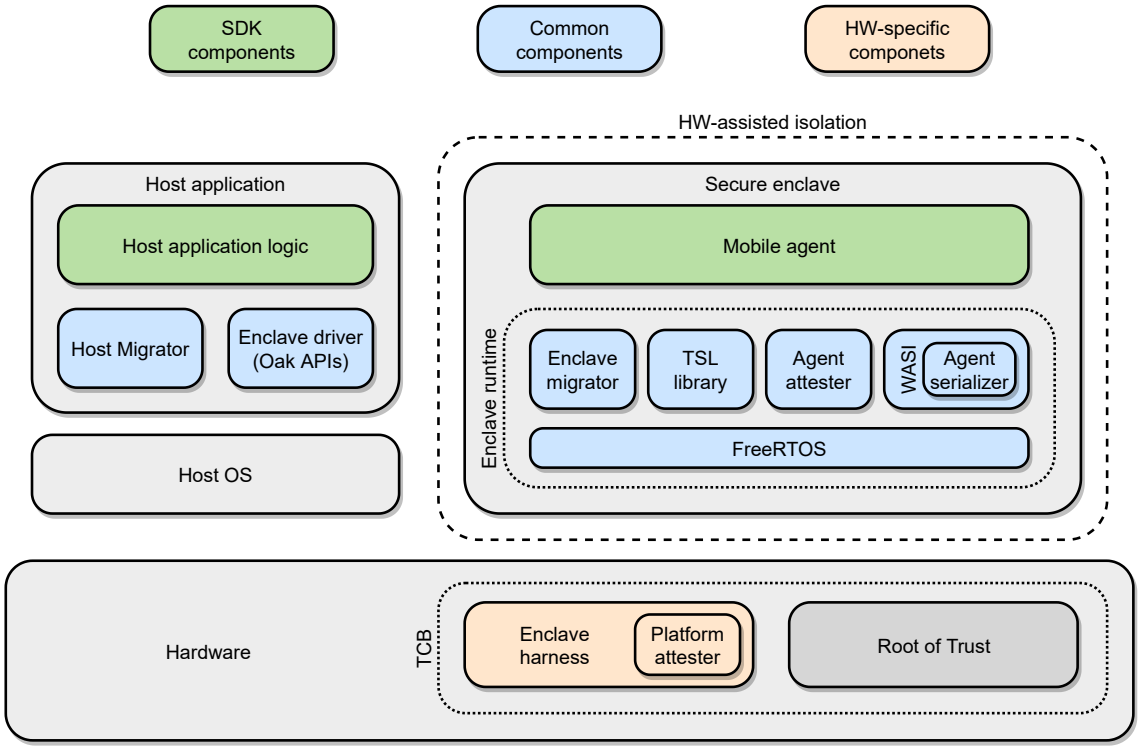

Figure 4.1: Secure enclave framework architecture.

The elements constituting the secure enclave framework are illustrated in Figure 4.1. The presented architecture is identical on every instance of our secure enclave, independently from being a *source/migrator* or *destination/target* instance in the migration scheme. In the figure, green boxes represent components that can be implemented by the developer using the SDK planned for the project. Here, the *Mobile agent* is the migratable and architecture-agnostic element that the architecture aims to protect. The blue and yellow components instead, are developed by the OEM and are the core of the secure enclave architecture. The implementation of the yellow modules is specific to the hardware architecture of each platform, while the implementation of the blue ones is common in any instance of the secure enclave framework regardless of the underlying hardware architecture.

The architecture is composed of three main elements:

- The *host application* or *host app*, running in the normal operating system of the host device, along with the other untrusted user-space applications.
- The *secure enclave*, an SEE protected by the hardware-assisted isolation.
- The *enclave harness*, which is part of the TCB and responsible for the hardware-assisted isolation of the secure enclave.

Each of these components is divided into more specific submodules, each with a distinct task for the correct operation of the infrastructure, as described below.

4.1 Host application

The host application is running in the user space of the device along with the other untrusted applications managed by the (untrusted) OS. The component consists of three submodules, (i) the *host application logic*, (ii) the *host migrator*, and (iii) the *enclave driver*.

4.1.1 Host application logic

The host application logic is the actual end-application written by the developer. Therefore, it usually requires a great amount of resources that typically are not available inside a secure enclave. So, for this reason, the host application logic is located in the untrusted environment, in order to take advantage of all the resources available on the platform. Therefore, whenever the implemented program has the need to perform a task that requires SEE-guarantees and possibly migration capabilities, it has the ability to launch an enclave-bound mobile agent that shall perform the duty. The implementation of the host application logic can be achieved within the framework SDK and it is the only module in the untrusted environment that is not provided by the OEM.

4.1.2 Host Migrator

The host migrator submodule is one of the core components running in the untrusted environment and is contributing to the correct migration of the mobile agents between the different secure enclave instances. The main duties of this module are:

- to handle the initiation of the migration procedure,
- to manage the communication between the host app and the secure enclave,
- and to handle the TCP communication among the source and target instances.

Given that the module is providing the TCP connection between the source and destination enclave, all the data exchanged among the two secure enclave instances have to pass through the host migrator. However, even though the migrator runs in the REE, the security of the mobile agent migration is not threatened because all migration messages leaving the secure enclave are already encrypted within the enclave by the *Trusted Sockets Layer* (TSL) library present in the *enclave runtime* (see Section 4.2.2.5).

4.1.3 Enclave driver

The role of the enclave driver component is to provide to the host application logic the means to administrate the secure enclave on the platform. These, mainly consist of the possibility of instantiating secure enclaves on-demand and loading the mobile agent into the secure enclave. In addition to this, the enclave driver manages also the communication between the host application and both, the enclave runtime and enclave harness. The latter capability is required for the execution of the security-critical tasks (as mobile agents) in the SEE requested by the end-user's programs, and the migration of the mobile agent itself.

4.2 Secure enclave

The secure enclave is a Secure Execution Environment and as such, it is running inside the *hardware-assisted isolation* domain provided by the enclave harness. Each instance of a secure enclave accommodates an enclave runtime and a mobile agent. The mobile agent can be instantiated in two different manners, (i) either by the host application logic through the enclave driver, (ii) or by receiving it from another secure enclave instance. The enclave runtime instead, is preinstalled on the platform and cannot be migrated to other devices.

The source code of the enclave runtime and its submodules is the same for all instances, but must be separately compiled for each hardware architecture. All this is possible thanks to the support provided by the enclave harness, which is specifically implemented for each platform and provides an abstraction layer between the hardware and the secure enclave (see Section 4.3).

4.2.1 Mobile Agent

The mobile agent is a WebAssembly bytecode (Wasm) program. This allows it to be executed in a platform-agnostic environment. This remarkable ability is what makes it suitable for its migration between different secure enclave instances independently from the hardware architecture. The last-mentioned feature is supported by the enclave runtime, which ensures that the destination instance is trustworthy and the security guarantees are not violated during its relocation.

Similarly to the host application logic, the mobile agent is not provided by the OEM and its implementation can be accomplished within the framework SDK. A further value brought by WebAssembly is that the mobile agent can be developed in any programming language as long as it can be compiled to WebAssembly bytecode.

4.2.2 Enclave runtime

The enclave runtime incorporates a real-time operating system[1] (RTOS) that ensures support for all remaining subcomponents of the module. The RTOS is based on the *FreeRTOS kernel* [126], a kernel with a tiny footprint and proven robustness. In particular, the *FreeRTOS-rust* project [127] is used, which is a wrapper for the FreeRTOS kernel that simplifies its usage in embedded applications written in Rust, or in this case, inside secure enclaves.

In addition to an execution environment, the kernel provides also (i) scheduling and (ii) intertask communication for the tasks[2] executing within the enclave runtime. And so, it empowers the "simultaneous" execution of multiple submodules and the

[1] An OS that normally provides a highly deterministic execution pattern and response to external events. Usually employed in embedded systems that must satisfy strict deadlines (in the order of milliseconds) in response time.

[2] For more details on the concept of "Task" in the FreeRTOS kernel and RTOS in general refer to [128]

exchange of information between them.

4.2.2.1 WebAssembly Interpreter

The WebAssembly Interpreter (WASI) is the module that creates the execution environment for the mobile agent and it consists of our WASI prototype. The prototype is an extension of the *WASMI interpreter* [129] which adds the *pausing*, *serialization*, *deserialization*, and *resumption* features to the standard version of it. Although interpreters that already offer these features exist, such as *Wasmer* [130], our decision was driven by other factors. The choice of adopting the WASMI interpreter can be mainly attributed to three characteristics:

- The **Rust-based implementation**, which is preferred for its memory-safety guarantees.
- The **small code size**. This is critical because in our architecture the interpreter runs in an SEE. And so, by being part of the elements that are executed in an SEE it is recommended to have a small footprint in order to lower the probability of a bug occurring. In addition to this, the secure enclave may have restricted resources (e.g. Intel SGX).
- The ***no_std* support**, that is a crate-level[3] attribute in Rust which means that the crate will not include the standard library of Rust (and any API providing platform integration), and so making it platform-agnostic. Further, this reduces the code footprint making it ideal for our usage.

An additional benefit of the Wasm interpreter and WebAssembly bytecode is that each mobile agent will be executed within a sandboxed environment that separates it from the enclave runtime through fault isolation techniques [131]. And so, any

[3]A crate is a compilation unit in Rust

mobile agent will be prevented from escaping the sandbox without going through appropriate APIs [131].

4.2.2.2 Agent serializer

The agent serializer can be seen as a submodule of our WASI prototype. It is the component that contains all the features added to the WASMI interpreter. These features play a key role in the migration procedure since they are performing preparatory and conclusive operations on the mobile agent's code and state. As mentioned in the above section, the additional features are:

- **Pausing**: the execution of the mobile agent needs to be suspended for the migration to take place. This activity is far from trivial because the module needs to be aware if the mobile agent is performing a critical operation that cannot be interrupted or if the execution is in a state that allows the suspension, and thus the migration to occur.

- **Serialization**: once the mobile agent was paused in a safe state, it is not ready to migrate yet. The running bytecode as well as the current state of execution and the internal data structures, need to be serialized in order to be transferred. As an additional step, during the serialization process, the data is also compressed in order to optimize the transmission of the information.

- **Deserialization**: once the serialized mobile agent arrives at the target secure enclave, it needs to be deserialized and decompressed to resume its execution. This operation will result in obtaining on the destination instance the same internal representation of the mobile agent as it was on the source one.

- **Resumption**: finally, the bytecode, state, and data structures are loaded back into the Wasm interpreter and the execution is ready to be resumed from where it was interrupted.

4.2.2.3 Enclave migrator

The *enclave migrator* is the migration logic inside the secure enclave. Its main task is to orchestrate the migration process between the source and target secure enclave by employing the *enclave attester*, the *TSL library*, and the *agent serializer* modules. To accomplish this duty, the enclave migrator is supporting the Trusted Sockets Layer protocol (TSL) [18] in establishing a trusted channel between the instances. The last-mentioned protocol will ensure that both instances are trustworthy and that none of the security guarantees ensured by the hardware-assisted isolation will be violated during the transfer of the mobile agent.

In addition to managing the modules participating in the migration scheme, the enclave migrator handles also the communication between the secure enclave and the host migrator. The communication between these elements is assisted by the shared memory provided by the enclave driver module.

4.2.2.4 Agent attester

The agent attester module is one of the two components producing the attestation claims and evidence for the trusted channel establishment. The evidence are decisive in the remote attestation procedure since they state the trustworthiness of the attested party. The specific attestation claims produced by the agent attester are computed on the mobile agent's state and bytecode. The claims are then combined with the freshness guarantees and the channel bindings in order to form the attestation evidence that will be signed by the enclave runtime. The attestation evidence over the mobile agent together with the ones over the platform, define the identity of the mobile agent within a specific migration and can be used by the appraiser to check its trustworthiness.

4.2.2.5 Trusted Sockets Layer library

The TSL library is the unit implementing the trusted channel protocol defined by Niemi et al. [18]. The library is an extension to a proprietary, size-optimized, standalone TLS 1.3 library. The decision of using this specific library was driven by its (i) tiny code size footprint, (ii) lack of dependencies, (iii) ease-of-extension using callback interfaces. In addition to this, our TLS 1.3 library gives also the possibility to compute the *early_exporter_master_secret*[4]without transmitting early data in the connection. This secret is employed by the TLS-Exporter[4] in the generation of the challenge that is included in the attestation evidence.

4.3 Enclave harness

The enclave harness is part of the TCB and is supporting the instantiation of secure enclaves on different platforms. It is specifically implemented for each hardware architecture to leverage its specific hardware isolation techniques. In addition to hardware-assisted isolation, the enclave harness features also an abstraction layer between the hardware and the secure enclave. This brings a major benefit because it gives the possibility to have a single implementation of the enclave runtime that can run on any hardware architecture. Another relevant component to our scheme is the platform attester that will be presented next.

4.3.1 Platform attester

The platform attester is the second component in the architecture that is in charge of generating attestation claims and evidence in the remote attestation process. Its role is to measure the platform TCB and the enclave runtime in order to produce and sign the attestation claims about these two elements. The claims over the TCB

[4]See RFC 8446 for further details [6]

are specific to each platform architecture since they are dictated by the platform's attestation mechanisms. For instance, *TPM-based attestations* can be used in devices equipped with a TPM module, meanwhile *TEE-based attestation* can be used in ARM-based mobile devices.

5 Migration design

As specified in the previous section, architecture agnostic code, like the WebAssembly bytecode is ideal for mobile agent applications. However, this property alone is not enough. By definition, a mobile agent needs a migration mechanism supporting its mobility between various platforms.

5.1 Requirements

Although migration of mobile agents and migration of SEEs are not new concepts, the migration of mobile agents between SEEs on different architectures is still a rather unexplored area. By definition, mobile agents are software with strong mobility features. As such, the migration mechanism should ensure that a mobile agent can move from one platform to another via a stateful migration.

On top of this, our mobile agents are running inside SEEs. And so, their mobility capabilities should not have an impact on the security guarantees enforced during their execution.

5.1.1 Security Requirements

Even though the architecture described in the previous chapter serves as an infrastructure for both the execution and the migration of mobile agents, the security requirements are issued separately for these two procedures. For the execution of mobile agents, our requirements are extracted from [132][19][133]:

1. **Security of the mobile agent against attacks from the platform**: The platform shall provide confidentiality, integrity, and availability to the mobile agent state and execution, such that the agent is protected from attacks by the untrusted environment of the platform. In addition the platform shall be able to give reliable evidence and authentication of the isolated SEE hosting the mobile agent to ensure trustworthiness in the system.

2. **Security of the platform against attacks by the mobile agent**: The mobile agent shall be distinguishable via an identity for access control, auditing, and non-authentic variants of it. Furthermore, the mobile agent shall be sandboxed from the platform such that it cannot damage or attack the platform.

3. **Security of the mobile agent against attacks from the network**: The mobile agent migration shall occur over a trusted channel (see Section 2.2.4.2).

4. **Security of the mobile agent migration**: The migration shall assure that mobile agents keep executing in a secure environment, i.e. if the requirements *1. & 2.* hold in the source platform they must also hold in the target platform.

The last two requirements heavily rely on the architecture, the migration scheme and the trusted channel supporting the relocation of mobile agents. The following requirements are specific to the secure migration of mobile agents and this thesis:

- **SR1 (Secure channel).** The communication shall provide confidentiality, integrity, authenticity, and freshness of the messages exchanged on the channel. The channel establishment shall provide mutual authentication of the channel endpoints while preserving their privacy.

- **SR2 (endpoint trustworthiness).** For the establishment of a trusted channel, the endpoints shall mutually attest to each other by providing reliable at-

testation evidence about themselves and their TCB state. The trusted channel shall never be confirmed without valid attestation.

- **SR3 (Channel binding).** The mutual attestation shall be strongly bound to a specific Trusted Sockets Layer handshake. As such it shall be valid only if generated by the endpoints participating in the communication to protect from relay and collusion attacks[1].

- **SR4 (Privacy).** The attestation privacy, identities, and mobile agents shall be protected against eavesdropping from unauthorized third parties even on the local platform.

- **SR5 (Forward security).** The disclosure of long-term keys shall not endanger past migrations by any means.

5.2 Migration phases

The secure migration of mobile agents in the secure enclave framework developed by the Huawei team, consists of multiple steps. These can be named as (i) *snapshotting*, (ii) *serialization/deserialization*, (iii) *attestation*, (iv) *transmission*, and (v) *resumption*. The secure migration scheme described in this thesis and proposed in the paper *"Towards Secure Mobile Agents"* by Pop et al. [19] is formed by a composition of these steps.

In the presented migration scheme, the phases can be distinguished between the ones that are specific to the mobile agent and the ones that involve the whole architecture. For instance, the snapshotting, serialization, deserialization, and resumption phases imply operations just over the mobile agent's state and code and

[1]A situation where the attacker is able to extract the attestation from a valid secure enclave and present it to the appraiser as its own.

occur fully within the same secure enclave (source or target). Meanwhile, the attestation and the transmission phases have a broader scope. That is because they include also the state of the platform (e.g. in the case of the attestation), or because they address the whole SEE-to-SEE communication mechanism supporting the Trusted Sockets Layer protocol implementation.

5.2.1 Snapshotting & Resumption

Snapshotting is the process of pausing the mobile agent and providing to the serialization procedure the internal representation of the interpreted code and state of the mobile agent. In contrast to just stopping the WebAssembly bytecode running on the Wasm interpreter, pausing it is a more demanding operation. In the pausing procedure, the execution state needs to be saved/snapshotted in order to be able to resume its execution from the exact state where it was suspended. To achieve this, the running mobile agent needs to be in a safe state for the snapshotting to take place. The executing bytecode cannot just be paused at any state because it may be in the middle of a critical operation that cannot be interrupted. To address this issue, there are two possible solutions:

1. To insert calls to a function in the mobile agent source code that would periodically check if it is time to trap the execution or not. These calls can be placed in loops or functions that have high probability to be executed frequently.

2. To insert a similar call as above, but at compilation time. In this scenario, the call can be optionally inserted by the compiler in a way that assures its call so often that will give the possibility of trapping the execution.

Due to its straightforward implementation, the solution adopted for our proof-of-concept is the first one and the trapping function is included in the agent serializer module of the architecture. Its integration in our migration process gives us the

possibility to pause the mobile agent in a safe state and continue by snapshotting it. The objects representing the WebAssembly bytecode and the execution state can be fetched from the WASMI interpreter. For instance, the *Globals*, *Memories*, *Functions*, and *Tables* objects representing the WebAssembly bytecode and values are contained in the *ModuleInstance* structure. Meanwhile, the *ValueStack* and *CallStack* objects can be retrieved from the *InterpreterState* structure and represent the state of the mobile code. At this point, once all the data objects are brought together, the serialization phase is ready to take over.

The resumption phase is the last one and it enables the mobile agent to resume its execution. As such it occurs on the destination enclave. Once the mobile agent has arrived and its interpreted code and state have been decompressed and deserialized (as described in the next section), the resumption phase can take place. This takes the deserialized objects from the deserialization phase, builds back the internal representation of the WebAssembly bytecode and state, and loads them into the new Wasm interpreter. At this point, the interpreter has all the data structures representing the identical state where the mobile agent was interrupted, and so, its execution is ready to be resumed from the instruction present just after the trapping function call.

5.2.2 Serialization & Deserialization

The serialization procedure gives the possibility to the mobile agent to be transmitted over the network. This phase consists in converting the internal representation of the WebAssembly bytecode and its state in a stream of bytes able to be transferred over the network.

The serialization takes the results of the snapshotting phase and uses the CBOR data serialization format [134] to encode the outcome data structure. As can be seen in Listing 5.1 the CBOR structure includes both, the mobile agent's interpreted code

and state as they were snapshotted in the previous phase.

```
List <Globals>      # Global variables
List <Memories>     # Memory buffers
List <Functions>    # Source code
List <Tables>       # Function references
List <ValueStack>   # Local variables
List <CallStack>    # PC and return sites
```

Listing 5.1: Data structure representing the mobile agent's code and state that are transferred in the migration process

The CBOR serialized structure above represented is generated with the *Serde* framework. Serde is a tool for serializing and deserializing Rust data structures to and from different formats, CBOR included [135]. In order to optimize the data transfer between the different modules of the architecture and the transport over the network, the serialized data is subject also to compression operations. These are not performed over all the CBOR structure, but just over the objects that usually allow a high degree of compression, like the memory buffers. For instance, small mobile agents do not use entirely their memory pages, and so the remaining memory is full of zeroes that can be highly compressed.

The result of the above operations is the plaintext migration package that will be sent to the target secure enclave. Once the data has arrived at the target, the procedures must be performed in reverse to get back the internal representation of the mobile agent execution. The collection of all these tasks carried out in a reverse manner can be defined as the deserialization phase and it is the last phase before the resumption.

5.2.3 Attestation

The attestation phase plays a key role in the establishment of the trusted channel between the source and the target secure enclaves. It produces the proof used in the remote attestation to prove the trustworthiness of the migration endpoints. As described in the Architecture, the agent attester and the platform attester are the modules in charge of producing the attestation claims over the mobile agent and platform components (see Section 4.2.2.4 & 4.3.1). In particular, the two modules produce the following claims:

1. The **TCBAttestationClaims**, produced by the platform attester and consisting of the measurements of the TCB (enclave harness included).

2. The **RuntimeAttestationClaims** generated by the platform attester and related to the enclave runtime.

3. The **AgentAttestationClaims**, produced by the agent attester and including both, the code and the state of the mobile agent.

It is important to note that the AgentAttestationClaims are computed just at the source secure enclave, and so verified just by the target enclave. Meanwhile, the TCBAttestationClaims and the RuntimeAttestationClaims shall be computed on both endpoints and must be mutually validated by the enclaves. As illustrated in Listings 5.3, 5.4, & 5.5, the claims are described as ASN.1 data types. ASN.1 is a notation described in the X.680 specification [136] and its purpose is to define the syntax of structured information data. Furthermore, as can be seen in the structures of the claims, all three include the *challenge* generated by the trusted channel. The just mentioned challenge is computed by employing the *TLS-Exporter* to extract keying material from the handshake of the Trusted Sockets Layer protocol. The TLS-Exporter is a specific mechanism specified in the TLS 1.3 protocol [6] (on whom the TSL protocol is based) and it is the recommended technique for exporting

handshake-specific secret key material. As such it ties each of the aforementioned claims to a unique handshake, and so, a unique trusted channel.

Designing the attestation of the TCB and the runtime is a task far from trivial. Indeed, this implies designing the operations of the platform attester, which has to be implemented specifically for each platform. The major problems arise from the type of attestation evidence that each platform is able to provide for the TCB and enclave runtime.

For instance, as can be seen on the left side of the Figure 5.1, ARM-based mobile phones usually confirm the trustworthiness of the device by attesting the existence of a *device key* in their TEE after they have been securely booted. The secure boot is a mechanism where an initial immutable image is cryptographically verifying the digital signature of the image that it loads. This process is done in turn by the just loaded image with the next image to be loaded up to the OS kernel [137]. In this scenario, the attestation of the key contains claims that describe the TEE and device state e.g. whether the secure boot was used. And so, even though these devices typically do not provide measurements over the state of the TCB like TPMs. It is possible to implicitly trust the TCB components (enclave harness included) under the condition that the *device key* has been remotely attested. The device key in turn will be used to attest the *harness key*, which again will be used to attest the enclave runtime. As a result, the attestation evidence produced by the platform attester, in this case, are two, the *TCBAttestationEvidence* and the *RuntimeAttestationEvidence* as shown in Figure 5.1, defined in Listings 5.3 & 5.4, and represented in Figure 5.3 as *TCBEv* & *RuntimeEv*.

Another example of platform attestation is on platforms equipped with a TPM. Here a different mechanism is adopted to attest the TCB. The TCBAttestation-Claims consist of dynamic measurements collected by the TPM during the boot process of the device. The measurements are collected in specific registers of the

TPM and include the enclave harness as well. For the attestation, the TPM provides specific APIs that take the just mentioned registers signs them, and provides the signature composing the TCBAttestationEvidence. Furthermore, the TPM also provides the possibility to include user-defined values in the registers. As a consequence, this enables the inclusion of the enclave runtime in the TCBAttestationEvidence as shown on the right section of Figure 5.1.

These two examples are clear proof of the discrepancy in the attestation procedures of the TCB for each platform and that is why in the *TCBAttestationEvidence* sent in the handshake, also the *attestationType* is included (see Listing 5.3). The type is used as a mean for the verifier to select the correct validation procedure while processing the *TCBAttestationReference*. Where the last-mentioned records are preinstalled, signed by the *Migration SubCA*, and represent a good value for the state of the platform as will be explained more in detail in Section 5.2.3.1.

From the examples above, we can also imply that the RuntimeAttestationEvidence is not always required to be sent separately. That is because its claims may be included in the TCBAttestationEvidence. However, the RuntimeAttestationClaims even though are generated by the platform attester, are not platform-specific. In fact, the claims over the enclave runtime consist just of its SHA-256 digest as represented by the *runtimeHash* and the *challenge* in the Listing 5.4.

As the RuntimeAttestationClaims, also the AgentAttestationClaims are independent from the platform since the implementation of the agent attester is common across all secure enclaves. As illustrated in Listing 5.5 the AgentAttestationClaims consist of the *codeHash*, the *currentStateHash*, the *challange*, an optional *log*, and the *timestamp*. The first two fields are the SHA-256 digests computed respectively on the WebAssembly bytecode and the state of the execution. The challenge, as in the TCBAttestationEvidence, provides the freshness and channel binding guarantees. The last two fields provide additional information to the target, e.g. the

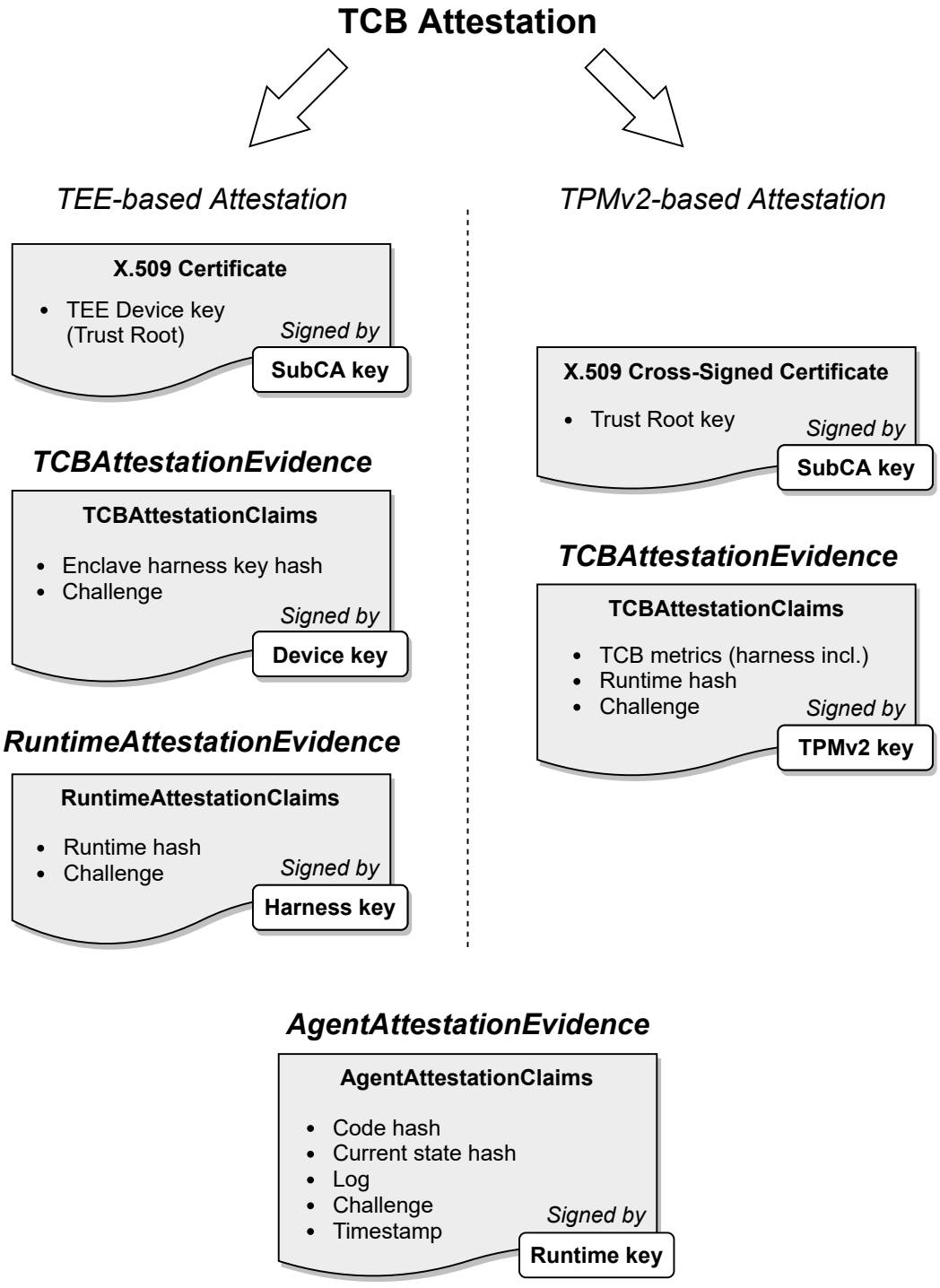

Figure 5.1: Attestation claims contained in the attestation evidence of the three components: (i) TCB, (ii) enclave runtime, and (iii) mobile agent; and the private keys used to sign them.

last platforms where the mobile agent was executed in the log field or the time of pausing in the timestamp.

The *AgentAttestationEvidence* containing the claims about the agent is illustrated in Listing 5.5 and Figure 5.1, and it is represented in Figure 5.3 as *AgentEv*. The latter evidence is optionally signed by the enclave runtime and included in the certificate issued for the TSL endpoint authentication ephemeral public key. Here, the signature over the AgentAttestationEvidence is optional since the runtime is signing also the issued certificate. And so, the signature can be omitted as an optimization to avoid the double signature by the runtime. The certificate also includes the records mentioned earlier, which are (i) the TCBAttestationEvidence, (ii) the TCBAttestationReference, and if available (iii) the RuntimeAttestationEvidence and (iv) the RuntimeAttestationReference. The collection of all these records resulted in the *AttestationEvidence* ASN.1 data structure illustrated by Listing 5.2 and represented in the Figure 5.3 as *AttEv*. The ASN.1 structure is encoded in the certificate accordingly to the *Distinguished Encoding Rules* (DER) defined in the X.690 specification [138]. The choice of the ASN.1 DER encoding was done on the base that it is binary (compact), unique (can be safely signed), and widely supported by encoders.

```
AttestationEvidence ::= SEQUENCE {
  tcbEvidence           TCBAttestationEvidence,
  tcbReference          TCBAttestationReference,
  runtimeEvidence   [0] RuntimeAttestationEvidence      OPTIONAL,
  runtimeReference  [1] RuntimeAttestationReference     OPTIONAL,
  agentEvidence     [2] AgentAttestationEvidence        OPTIONAL }

TCBAttestationReference ::= SEQUENCE {
  metrics               SEQUENCE OF ReferenceMetric,
  signerInfo            SignerInfo,
  signatureAlg          AlgorithmIdentifier,
  signature             OCTET STRING }

ReferenceMetric ::= SEQUENCE {
  metric                OCTET STRING }

RuntimeAttestationReference ::= SEQUENCE {
  runtimeVersion        OBJECT IDENTIFIER,
  runtimeHash           OCTET STRING,
```

```
  signerInfo            SignerInfo,
  signatureAlg          AlgorithmIdentifier,
  signature             OCTET STRING }

SignerInfo ::= SEQUENCE {
  identity        CHOICE {
    x509Cert        [0] SEQUENCE OF Certificate,
    rawPubKey       [1] SEQUENCE OF OCTET STRING } }
```

Listing 5.2: Attestation extension ASN.1 structure data types

```
TCBAttestationEvidence ::= SEQUENCE {
  attestationType       OBJECT IDENTIFIER,
  tcbClaims             TCBAttestationClaims,
  signerInfo            SignerInfo,
  signatureAlg          AlgorithmIdentifier,
  signature             OCTET STRING }

TCBAttestationClaims ::= SEQUENCE {
  metrics           [0] SEQUENCE OF Metric,
  challenge         [1] OCTET STRING }

Metric ::= SEQUENCE {
  metric                OCTET STRING }
```

Listing 5.3: TCB Attestation Evidence ASN.1 structure data types

```
RuntimeAttestationEvidence ::= SEQUENCE {
  runtimeClaims         RuntimeAttestationClaims,
  signerInfo            SignerInfo                  OPTIONAL,
  signatureAlg          AlgorithmIdentifier         OPTIONAL,
  signature             OCTET STRING                OPTIONAL }

RuntimeAttestationClaims ::= SEQUENCE {
  runtimeHash       [0] OCTET STRING,
  challenge         [1] OCTET STRING }
```

Listing 5.4: Runtime Attestation Evidence ASN.1 structure data types

```
AgentAttestationEvidence ::= SEQUENCE {
  agentClaims           AgentAttestationClaims,
  signerInfo            SignerInfo,
  signatureAlg          AlgorithmIdentifier,
  signature             OCTET STRING }

AgentAttestationClaims ::= SEQUENCE {
  codeHash          [0] OCTET STRING,
  currentStateHash  [1] OCTET STRING,
  challenge         [2] OCTET STRING,
  log               [3] OCTET STRING            OPTIONAL,
  timestamp         [4] GeneralizedTime }
```

Listing 5.5: Agent Attestation Evidence ASN.1 structure data types

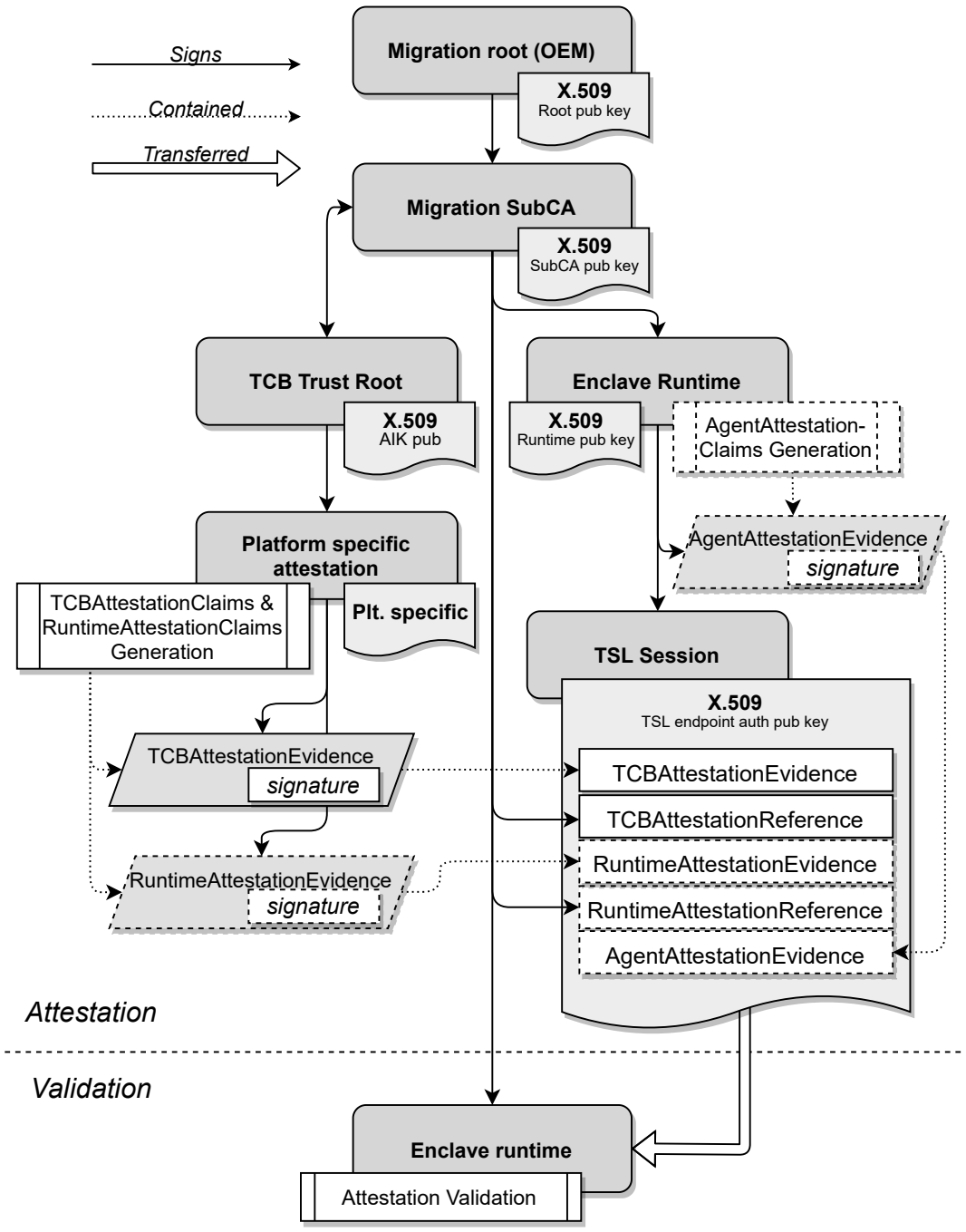

Figure 5.2: Trust chain employed in the migration process. The Trust chain includes the parties generating the attestation evidence and the attestation reference as well as the appraiser validating the attestation. The dashed boxes in the Figure represent the optional elements in the attestation.

For all the attestation evidence, a PKI-like trust chain is assumed for their verification, as shown in figure 5.2. Each secure enclave will carry all the required certificates to bind through the trust chain the AttestationEvidence to the *OEM-specific Migration rootCA*. The trust chain for the attestation verification is built through X.509 certificates and it may need cross-signed certificates from the *TCB Trust Root* and the *Migration SubCA* to achieve compatibility between the secure enclave and the specific TCB. As a result, the Migration SubCA is required to sign four categories of records employed in the attestation procedure:

1. **The X.509 certificate containing the public key of the enclave runtime**. This will be leveraged to sign the X.509 certificate of the endpoint ephemeral authentication public key used in the trusted channel protocol. As a consequence, also the AttestationEvidence will be signed by the secure enclave because it is included as an extension in the certificate of the TSL session key.

2. **The cross-certified TCB Trust Root**, which is employed in the TCB attestation procedure. The Trust Root for each architecture may differ based on the mechanism used to attest the trustworthiness of the TCB, and so, the Trust Root needs to be signed by the Migration SubCA and vice versa. This mutual signature is required by the relying party in the validation procedure in order to trust the Trust Root.

3. **The TCBAttestationReference**. This contains a list of reference values that are compared against the metrics of the TCBAttestationClaims and if valid, they prove the trustworthiness of TCB.

4. **The RuntimeAttestationReference**. This includes the version of of the runtime running on the attested platform and the hash reference for its state.

5.2.3.1 Attestation validation

The endpoints participating in the trusted channel communication must always perform the validation of the attestation. To successfully complete the validation procedure, the secure enclave needs the evidence of the attested components and the attestation references for the TCB and eventually the enclave runtime. As described above, the evidence is generated by the attested endpoint and represents the state of the platform and in case the state of the mobile agent. The references instead, are generated by the Migration SubCA and represent known-good values for the state of the platform. The references are signed by the Migration SubCA in order to define the accepted states for the platform. And so, if the platform provides metrics in the evidence that differ from the references, it means that the platform has been compromised.

The validation of the attestation is accomplished in four steps. In the first one, the signatures over the TCBAttestationClaims, and eventually the RuntimeAttestationClaims and AgentAttestationClaims are verified to be valid. In the second one, the challenge (included in the claims of each component) is compared against the one provided by the TLS-Exporter for the current trusted channel handshake. In the next step, the signature over the references received from the attested party is verified in order to ensure also that it belongs to the Migration SubCA. Finally, the metrics in the evidence are compared with the ones contained in the references to make sure that they are matching. If an error occurs in any of the just described steps the validation of the attestation fails and the establishment of the trusted channel is aborted.

In addition to the attestation validation procedure, in the case the mobile agent has to be received, before its resumption, its actual state should be verified. This process can be performed by comparing the AgentAttestationClaims against the mobile agent received in the application data of the trusted channel. If the hash of

the received data matches with the claims in the AgentAttestationClaims, then the mobile agent can be considered trusted and its resumption can take place.

5.2.4 Transmission

The transmission phase consists of the actual transfer of the data from the source to the destination enclave. In this phase, the communication between the two secure enclave instances is provided by the Trusted Sockets Layer library. Normally, a TLS 1.3 library employs a TCP socket to send the data over the network and establish a secure channel with the other endpoint. However, in our prototype, this would mean to implement into the secure enclave all the layers of the TCP/IP model from the TSL down. That is because just in this way the secure enclave can communicate directly over the network. On the other hand, this approach is not optimal, because it would increase the code size of the secure enclave, and thus its attack surface. To overcome this issue, a different strategy was adopted. Instead of including the TCP/IP stack in the secure enclave, the TSL records are forwarded from the trusted channel library to the host application. Thus, obtaining an architecture where the TCP endpoint is in the host application and the TSL endpoint in the secure enclave as illustrated in Figure 5.3. Therefore, the channel underlying the TSL connection can be split into two distinct sections: (i) the communication channel internal to the host device (Shared memory IO), and (ii) the external one connecting the two TCP endpoints over the network (TCP/IP).

5.2.4.1 Shared memory IO Communication

The first section of the channel underlying the TSL protocol is the *Shared memory IO* represented in Figure 5.3. This section is in charge of transferring messages between the secure enclave (TSL endpoint) and the host application (TCP endpoint). Since the secure enclave is instantiated by the host application (more specifically its

submodule "enclave driver"), at the time the instantiation occurs, the host application reserves also the shared memory for the communication. The shared memory can be written and read by both endpoints in our prototype. Enabling so, the transfer of messages between the host application and the secure enclave. The message exchange is based on a mechanism enforcing IO interrupts to synchronize the access to the shared memory, and this is handled differently by the two components. The transfer of information is always initiated by the secure enclave through an IO interrupt that will be received by the host application. Every time an interrupt is performed, the execution of the secure enclave will hold until a reply will be received back. On the host application side instead, the channel is managed by handling the IO interrupts coming from the secure enclave. These will take place every time the secure enclave wants to exchange a message on the shared memory and will warn the host application that a new request has been made. For instance, when the secure enclave needs to receive a message, it employs a polling mechanism, that causes multiple IO interrupts until a message or an error is returned.

5.2.4.2 TCP/IP Communication

The second half of the communication underlying the trusted channel protocol is handled by the host migrator and is illustrated in Figure 5.3 as *TCP/IP*. Its role consists in managing the messages exchanged with the TSL library and forwarding them to the destination device. The communication with the secure enclave is administrated by the main process of the host migrator, meanwhile, the TCP connection is handled by two additional threads. The first one handles the outgoing messages, returning an error if the procedure was not successful, while the second one handles the incoming messages. Here, when a message needs to be received, the main process is polling the receiving thread for the message and then forwards it to the TSL library. The reason underlying this design consists of having a TCP

connection that is asynchronous with the rest of the application. And so, preventing the host application from being blocked while waiting for a TCP message to arrive.

5.3 Migration flow

The TLS 1.3 protocol enables an efficient establishment of a secure channel between entities over the network and it has been very well analyzed from its inception. However, for a secure migration to take place, the guarantees of a secure channel are not enough. Indeed, also the trustworthiness of the two endpoints participating in the communication channel need to be assured. This can be accomplished through the addition of mutual attestation and strong channel bindings as described in the novel Trusted Sockets Layer protocol [18]. The TSL is based on TLS 1.3 and is fully compliant with the specification: the extra guarantees can be implemented with the callbacks offered by most of the TLS libraries. In fact, it was easily implemented in our proprietary size-optimized TLS 1.3 library and in OpenSSL [18]. On these grounds, the Trusted Sockets Layer has been considered as the optimal protocol to use for the implementation of the secure migration between secure enclave instances. The TSL library used in the PoC is the one based on our size-optimized TLS 1.3 library since the code size footprint is a decisive factor in our prototype.

The design process to integrate the TSL protocol into our project's architecture resulted in the migration scheme illustrated in Figure 5.3. The migration process is always initiated by the host application on both instances, with a difference in the type of migration requested. In the case the secure enclave contains a mobile agent, the request is to start the migration as the *Source enclave*, meanwhile, when the mobile agent is not present the migration is started as the *Target enclave*. Once the request from the host application logic reaches respectively the enclave migrator (on the source enclave) or the enclave harness (on the target), the modules initiate the migration (see Figure 5.3). In the former case, the enclave migrator just invokes

the Trusted Sockets Layer library to initiate the migration. Meanwhile, in the last case, the enclave harness sets up a new secure enclave where the enclave migrator immediately invokes the TSL library to establish the trusted channel.

As the TLS 1.3 protocol, TSL requires just 1.5 roundtrips to establish the trusted channel and send the serialized mobile agent. The first half of the roundtrip, at time 0-RTT, consists of just the *ClientHello* message, which differently from TLS 1.3 contains also the $AttReqExt_s$ defined in the TSL protocol [18] and depicted in Figure 5.3.

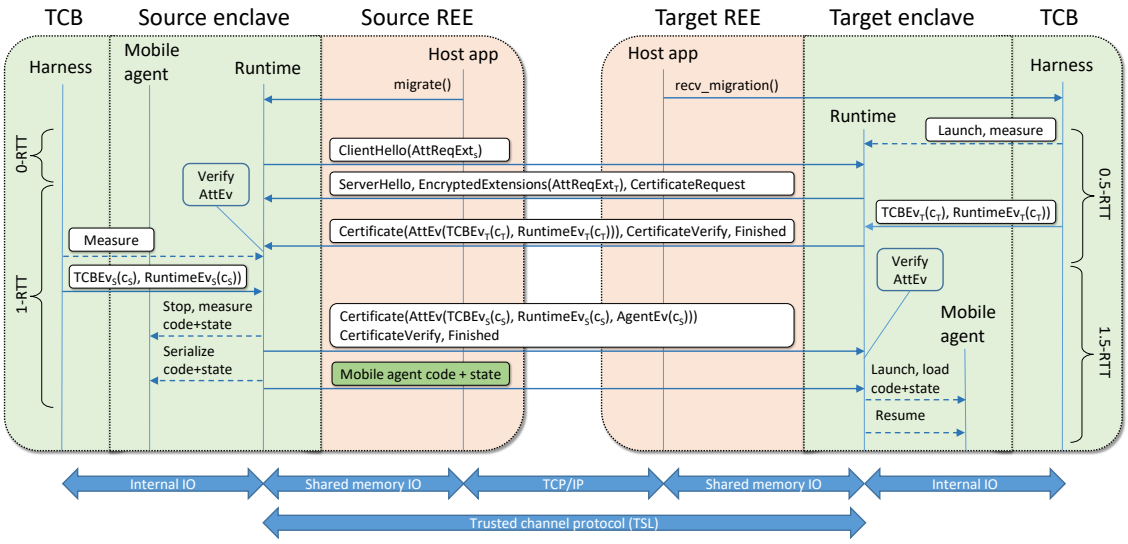

Figure 5.3: Migration scheme designed for the secure migration of mobile agents between secure enclave instances.

At time 0.5-RTT the target enclave performs all the operations required by the TLS handshake plus some operations for the TSL protocol. Firstly, it includes in the *EncryptedExtesions* the $AttReqExt_t$ (see Figure 5.3) in order to request the source enclave attestation as well. Then it invokes the platform attester module in order to obtain the evidences and the references (i.e. as described in the attesta-

tion) that have to be encoded in the AttestationEvidence. Finally, it inserts the AttestationEvidence in the X.509 certificate issued for the ephemeral TSL endpoint authentication public key and signs it.

At the time 1-RTT on the source enclave side, in addition to the TLS operations, the TSL library proceeds with the required operations for the mutual attestation. As a first step, it validates the target's TCBAttestationEvidence and eventually the RuntimeAttestationEvidence against TCBAttestationReference and RuntimeAttestationReference. Next it invokes the platform attester to get the platform attestations, performs the snapshotting phase, and invokes the agent attester for the agent evidence. Once these operations are complete, all the evidences composing the AttestationEvidence are encoded into the extension of the X.509 certificate, which will be signed and issued for the ephemeral TSL endpoint authentication public key. After the source enclave sends the *client_Finished* message, the trusted channel is established and the serialization and transmission phases of the mobile agent can take place.

At time 1.5-RTT the target enclave receives and validates the certificate of the source enclave. And so, if the certificate and attestation are valid, the target accepts also the mobile agent's state and code and compares them against the AgentAttestationEvidence to check their integrity. Next, the serialized data is forwarded to the enclave migrator, where the deserialization and resumption phases are processed. At this point, the target optionally sends back a confirmation that the agent was successfully resumed, thus terminating the migration procedure and closing the trusted channel.

6 Implementation

For the realization of the Proof-of-Concept (PoC) all three main modules, host application, secure enclave, and enclave harness have been implemented.

Each module and submodule in the architecture of the secure enclave framework was developed separately. The implementation of the components in the architecture was split among the team members of this project. As a consequence, this thesis will focus mostly on the modules that are the result of my personal contribution, and so, on the modules directly involved in the migration scheme. For instance, the exact implementation of the enclave harness will not be addressed, since its main role is to support the hardware-assisted isolation of the secure enclave and I did not contribute to its implementation.

In our PoC, Rust was the most favored and used programming language due to its memory-safe code properties. Despite this, also the C programming language was employed since also external libraries were used. In Figure 6.1 are shown the technologies used for the development of the different modules in the architecture. The green-colored components were implemented in Rust and the red ones in C. As it is possible to note, two modules have both colors. That is because the modules include or have to interface with code written in C, this will be later described for the enclave runtime and enclave migrator respectively in Sections 6.2.2 and 6.2.2.3. The mobile agent is the only yellow-colored component since it is the only one that is compiled to WebAssembly. Finally, some components of the architecture are left

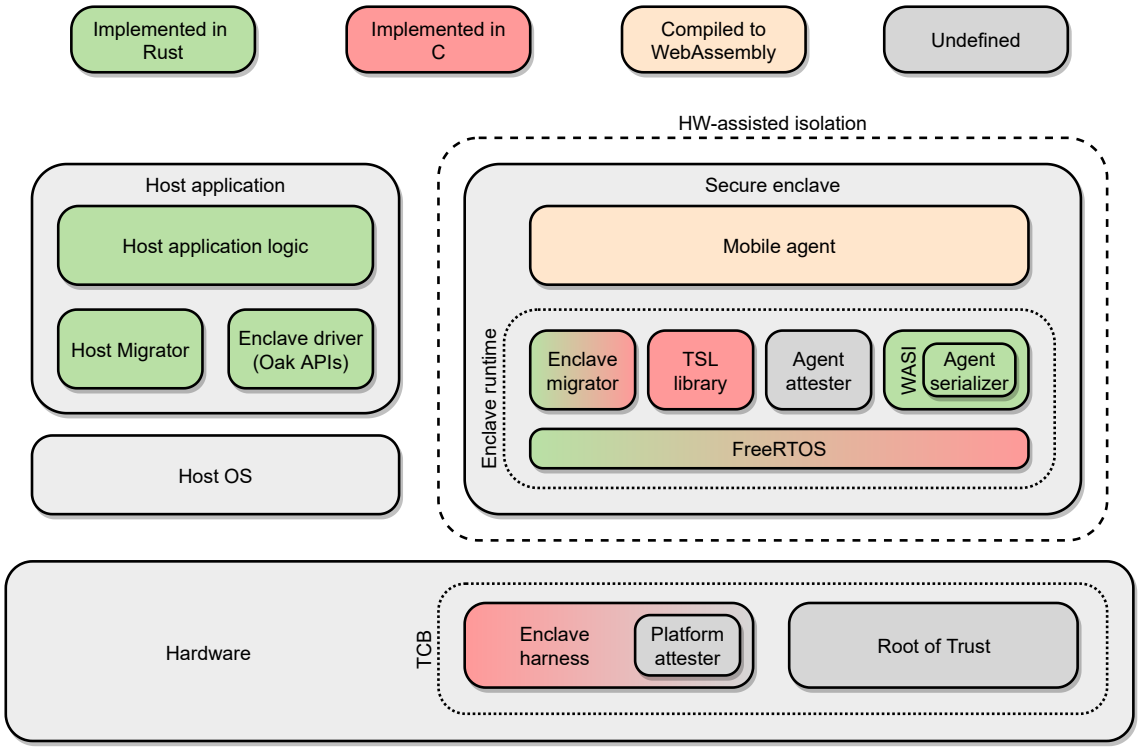

Figure 6.1: Technologies used to implement the componets of the architecture employed in the PoC.

in gray as a mean to show either that they were not fully implemented yet (e.g. the agent and platform attester) or that other technologies were used to achieve low-level hardware compatibility (e.g. enclave harness).

The next sections will follow the same outline as the Chapter 4 in order to explain more in detail the module's implementation or just to give some insights on how it was developed.

6.1 Host application

6.1.1 Host application logic

Given that the host application logic is intended to be developed by the end-user of the framework, for the completeness of the PoC a small demo program was developed. The component is written in Rust and its main duties are to (i) define the target instance that will receive the migration of the secure enclave and (ii) collaborate with the enclave driver to load the mobile agent in the newly instantiated secure enclave (i.e. execute a security-critical task into the SEE).

6.1.2 Enclave driver

The implementation of the enclave driver component is based on the *Google Project Oak* [139]. The Oak project is aiming at creating a specification and a reference for the secure transfer, storage, and processing of data. Furthermore, it provides the tools to create on-demand secure enclaves on different hardware architectures. Oak is a very vast project, providing many functionalities in the administration of secure enclaves. However, the enclave driver in our project exploits just some of its functionalities. Actually, Oak is exploited just for the instantiation of the secure enclaves which consists in loading the enclave runtime and optionally the mobile agent. Even though the Oak Project already provides some communication capabilities to the secure enclaves, in our enclave driver the communication is implemented through a shared memory between the host application logic and the enclave runtime. That is because our project uses enclaves for a different purpose than Oak. Indeed, in the Oak project, the secure enclaves are designed to offer some services over the network. Meanwhile, in our project, they serve as SEE where the end-user's programs can run their (security-critical tasks as) mobile agents.

The design of this communication channel is described in the "Shared memory IO

Communication" in Section 5.2.4.1. The channel was developed in Rust, and consists mainly in managing the shared memory access and defining the IO interrupts used by the host application and the secure enclave to communicate.

6.1.3 Host migrator

The host migrator, enclosed in the Host app of Figure 5.3, acts as an endpoint for both, the TCP connection with the target instance and the shared memory IO communication with the secure enclave. Both the endpoints within the module were developed in the Rust programming language and their duties are spread over three different threads:

- The **main thread**, which is handling the endpoint communication over the shared memory and administrating the main logic of the component.
- The **sender thread**, a child thread, which is part of the TCP endpoint and in charge of sending the messages over the TCP connection.
- The **receiver thread**, another child thread, which is part of the TCP endpoint as well and is waiting for messages arriving over the network.

The role of the endpoint managing the internal communication over the shared memory, mainly consists in processing the IO interrupts provoked by the secure enclave. Each IO interrupt that is received by the enclave driver has a specific identifier stating its origin and purpose. So, once the interrupt has been labeled as related to the migration procedure, it is forwarded to the host migrator. The last-mentioned module will handle the interrupt and will write back the response to the shared memory. The IO interrupt can be categorized into (i) *send request* and (ii) *receive request*. The first type is always related to sending messages over the TCP connection. As such, it is immediately forwarded to the TCP sender thread. The second type, based on the content of the request, can be either forwarded to

the TCP receiver thread or handled by the main thread. In both cases, the secure enclave is in a polling state and is asking either the TCP receiver for new messages on the TCP channel or the host application for the moment the migration procedure should start.

As aforementioned, the TCP endpoint is managed by two different threads. The TCP sender thread is simply forwarding the message on the TCP connection and replying with the result of the operation. The TCP receiver instead, is continuously waiting for both, (i) either a message to arrive on the TCP connection or (ii) a receive request from the secure enclave. In the former case, the thread just saves it until also a receive request arrives. In the last case, it either replies with the TCP messages arrived or by informing that nothing arrived.

The orchestration of all threads in the host migrator is performed by leveraging the *std::thread* [140] Rust crate. Meanwhile, the communication between them is fulfilled by the *std::sync::mpsc* [141] crate. The latter was extremely useful because it provides the means to easily handle asynchronous channels between the threads without any worry about concurrency. Finally, the last crate employed is the *std::net* [142], which enables the communication of the two TCP threads over the network.

6.2 Secure enclave

The predominant programming language for the implementation of the secure enclave (mobile agent excluded) is Rust. However, given the various external projects and libraries integrated into the architecture, the C language has a notable share of code as well (see Figure 6.1). The majority of the C code is due to the *FreeRTOS kernel* used as a base for the enclave runtime and to our size-optimized implementation of the TSL protocol.

6.2.1 Mobile Agent

As specified in the Architecture (see Section 4.2.1), the mobile agent is the only component that has code mobility properties and is compiled to WebAssembly bytecode. For our Proof-of-Concept, the mobile agent is a key store with facilities for generating keys and using them for cryptographic operations. In particular, the agent implements the XXTEA block cipher [143] in the C programming language providing encryption and decryption operations to the end-user.

As a mean to check the successful migration of the mobile agent's state and code, the agent was trapped during the execution of the encryption and decryption procedures. And, after the resumption, it was verified that the encrypting/decryption operations have been completed successfully.

6.2.2 Enclave runtime

The FreeRTOS-rust project [126] is employed to ease the integration of the FreeRTOS kernel together with the other components of the architecture. In its implementation, the most relevant aspect for us is the mechanism employed for the communication between the different modules over the FreeRTOS kernel. For instance, the WebAssembly Interpreter and the enclave migrator are executed on different tasks of the kernel, and so, they need a way to communicate with each other. The latter is implemented by leveraging the *FreeRTOS Queues* [144] for inter-task communications. These are provided by the kernel and further wrapped [145] by the FreeRTOS-rust project for the Rust programming language.

6.2.2.1 WebAssembly Interpreter

The PoC is employing our WASI prototype to create the execution environment for the mobile agent (see Section 4.2.2.1). The WASMI interpreter is developed in Rust and so it is the agent serializer, which is our extension to it. Their integration in the

enclave runtime is in the form of a task in the FreeRTOS kernel, in order to execute independently from the other components in the enclave runtime.

6.2.2.2 Agent serializer

The first features implemented by the extension are the snapshotting and the resumption described in Section 5.2.1. Their implementation was straightforward since the necessary data structures were already present in the internal state of the WASMI interpreter. For the trapping mechanism, the function to call was defined in the extension code as well. In fact, to make it callable from the mobile agent was enough to declare it into the mobile agent's source code.

The implementation of the serialization and deserialization phases instead was more challenging. Indeed, even though the Serde framework [135] provides the means to serialize the data structures, it was raising some issues. The cause was the smart pointers[1] employed by the Rust programming language. For instance, the main issue with these was that when two pointers to the same register were undergoing the serialization procedure, their deserialization was resulting in two pointers referencing two different registers instead of the same one. This issue was arising because the mobile agent's state does not contain the addresses of the registers but their values. This is needed because addresses on the target secure enclave will be different from the ones in the source enclave. Thus, even though before the serialization the pointers were referencing the same register, after the deserialization they were pointing to different registers but with the same value. After several trials, the issue was solved through the usage of a structure where the pointers and their value were tracked in order to reconstruct them as they were before the serialization.

The size optimization within the serialization phase is provided by the compression crate in Rust. In particular, the *BZip2Encoder* [147] is used to compress the

[1]For more details on the smart pointer in Rust see [146]

memory buffers in the mobile agent's state (see Listing 5.1). The reason why only these buffers are compressed is that they mostly contain zeroes in a simple mobile agent implementation. Thus, they have a high compression rate compared to the other data in the structure.

6.2.2.3 Enclave migrator

The enclave migrator is written in Rust and C code, and it is implemented as a task in the FreeRTOS kernel. The main logic and the mechanisms allowing the communication with both, the other tasks on the enclave runtime and the host application were developed in Rust. Meanwhile, the administration of the TSL library was written in C. To solve this discrepancy, the Rust's Foreign Function Interface (FFI) is used, which allows C code to be called from Rust and vice versa.

The communication with the WASI prototype is performed through the FreeRTOS Queues mechanism provided by FreeRTOS-rust. Meanwhile, the communication with the host application is handled through the shared memory IO communication described in Section 5.2.4.1.

The administration of the TSL library and the agent attester, although are written in C, are planned to be reimplemented in Rust code as well. The TSL library is directly called by the enclave migrator, which implements also the necessary callbacks for the trusted channel. The callbacks, as described in the TSL protocol contain the calls to the attestation and certificate generation procedures as well as their validation.

6.2.2.4 Agent attester

For the PoC the agent attester was implemented just to provide the channel bindings of the attestation to the Trusted Sockets Layer handshake. The channel bindings are generated by calling the TLS-Exporter mechanism provided by the TSL library and

are inserted into a sample of the ASN.1 structure illustrated in the Listing 5.2. From the listing, only the *challenge* (channel bindings) is implemented at the moment and the remaining evidence/claims are left for future work.

6.2.2.5 Trusted Sockets Layer library

The Trusted Sockets Layer library is a stand-alone implementation of the TLS 1.3 protocol. The library is developed in the C programming language and is code-size optimized with just 20-63KByte of footprint. This makes it ideal to be used in the enclave runtime, even if our architecture is not dependent on any specific library. The only requirement is to provide the necessary callbacks in order to implement the trusted channel protocol [18] and the IO callbacks for the transmission of the TSL protocol messages.

The TLS 1.3 library offers a callback function interface for implementing the transmission of TSL protocol messages. The default callbacks use the Unix socket interface to send messages over TCP/IP. In our case, the library runs in a secure enclave where there is no direct access to the TCP/IP stack. Therefore, the IO callbacks are written on our own, in order to send and receive messages via the shared memory channel between the host application and the secure enclave as described in the shared memory IO communication in Section 5.2.4.1.

6.3 Enclave harness

The proof of concept was implemented on two different platforms, a device with a *x86* architecture and one with an *ARM* architecture. On both of them the enclave harness is located at the hypervisor privilege level and use memory protection mechanisms. These consist in the *Intel VTx* [148] and *ARM EL2 and second stage memory translation* [149][150] virtualization technologies, provided by the x86 and ARM architectures. On both architectures, the enclave harness is based on the Kernel-

based Virtual Machine (KVM), which thanks to the just nominated virtualization technologies, provides isolation of the enclave from the untrusted environment.

6.3.1 Platform attester

The secure boot binding, and so the platform attester was not implemented in the PoC on the two devices and is left for future work. However, as described in the paper by Pop et al. [19], the detailed design for its development is already in place and the team is working towards its implementation.

7 Benchmarks & Analysis

The proof-of-concept described in Chapter 6 was subjected to functional and performance testing (benchmarks). In this section, we report a series of benchmarks that prove the efficiency of the migration process. The section will begin by reporting the configuration of the benchmarks that have been run and a performance analysis on the results. Subsequently, the chapter will continue with a discussion of the results and security measures enforced in the migration process.

7.1 Benchmarks setup

Like the enclave harness, the PoC has also been implemented on two platforms. The first one is a laptop with an Intel Core™i7-6600U 2.60GHz CPU and 16 GB of RAM. The second is a Raspberry Pi 4 model B board with a Broadcom BCM2711 (ARMv8) 1.5GHz CPU and 4 GB of RAM. As mentioned in the previous chapter, the isolation of the secure enclaves from the untrusted environments on the two platforms is provided by the KVM running at the hypervisor privilege level. The untrusted environment, where also the host application was deployed, is the Linux OS on both devices. In particular, the Linux OS distributions were *Ubuntu 20.04.2 LTS* on the laptop and *Debian 10* on the Raspberry Pi board. The benchmarks were performed in mainly two configurations, (i) with the source and target secure enclaves on different platforms and (ii) with the source and target secure enclaves on the same platform. In the former case, both the secure enclaves and the host

applications were on two different devices and linked through a LAN network. In this configuration, the connectivity had a latency of around $0.35ms$. In the second one instead, the two secure enclaves were running on two different instances of the KVM and the host applications were two separate processes connected through the loopback addresses of the platform. In the latter configuration, the latency is negligible and around $0.07ms$.

The mobile agent used as a reference is the XXTEA cipher implementation described in Section 6.2.1 and it was migrated while the encryption and decryption procedures on the mobile agent were executing. The mobile agent's state and code size once serialized, was about 10.2 KB. This was the only information transferred on the trusted channel as application data.

7.2 Benchmarks methodology & Results

The performance measures of the migration process in the PoC were carried out using multiple timers in the source code of the implementation. For both the secure enclave instances, multiple procedures were benchmarked, like the time employed by the handshake, the serialization, the cryptographic operations, etc. Since the measurements were computed on both enclave instances, some of the operations have been benchmarked on just one or both instances. This behavior comes from the fact that many measures include operations that are performed synchronously on both instances i.e. the TSL handshake. Meanwhile, other ones like the serialization, deserialization, and others occur entirely within the same secure enclave instance. Comparing the times taken on both instances allowed us to verify that the measurements of the various operations in the migration do not have significant discrepancies. This is of great importance because the two enclaves are isolated and so, the total time is obtained from the union of the unique measures on the two instances (i.e. serialization and deserialization) plus the common measures (i.e.

trusted channel protocol).

In the first configuration of the benchmarks, the main objective was to measure the performance of the migration process in a realistic scenario. Therefore, the source secure enclave was executed on the Raspberry Pi 4 board and the migration of the mobile agent was towards the target secure enclave running on the laptop device. This configuration may simulate, for example, the offloading mechanism employed by mobile devices in the Mobile Cloud Computing, since the mobile agent is moved to execute some operations on a more powerful device. The measurements for this case were run 100 times in order to have statistically valuable benchmarks and their results are presented in Figure 7.1 and Table 7.1.

The Figure 7.1 represents in a more intuitive way the findings in Table 7.1. The bars in the figure represent the main workloads performed in the migration procedure. Moreover, they are partially inclusive as a way to provide more details on each operation performed. In the results, the time required for the attestation procedure was not mentioned since the channel bindings require around $0.1ms$, and the attestation claims over the mobile agent, enclave runtime and TCB are not computed. Meanwhile, The time spent over the TCP channel was not included as well because due to the architecture of the PoC it was measured by the host application, and so, asynchronously to the other measures. However, the overhead imputable to the TCP connection can be neglected in our configuration since the latency is about $0.35ms$.

The objective of the second benchmark instead, was to measure the impact of the security mechanisms on the total time required for the migration procedure. Hence, the measurements of the migration procedure were performed several times, and each time with a slightly modified mobile agent that was allocating increasing amounts of memory. The increasing allocation of memory was simulating a bigger mobile agent and so a higher amount of time required for the mobile agent to be

	Communication	Serialization	Deserialization	Trusted channel protocol	Migration Control	TSL handshake	Transfer Mobile Agent	Crypto	Certificate generation	Certificate validation	Other hs operations
count	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000	100.000000
mean	65.891200	9.671870	4.820440	57.622310	8.268890	52.217580	5.404730	32.885170	10.023980	3.021650	6.286780
std	16.496608	0.038502	1.340309	16.284302	0.663176	16.276552	0.067711	8.371582	8.098112	0.386874	12.181487
min	43.224000	9.567000	2.661000	35.309000	7.714000	29.924000	5.309000	27.066000	5.193000	2.793000	-30.638000
25%	52.642750	9.643000	3.826500	44.407500	7.912750	38.887500	5.360750	27.407500	5.204000	2.807750	-2.022250
50%	62.485500	9.671500	4.688500	54.469000	8.071500	49.119500	5.380500	29.779000	7.790000	2.813000	5.642000
75%	77.315000	9.696500	5.764000	68.992000	8.339750	63.518750	5.449500	33.484500	8.476750	2.998750	14.130750
max	115.918000	9.783000	8.986000	107.969000	11.003000	102.592000	5.584000	73.642000	48.070000	4.210000	34.193000

Table 7.1: Time measurements in ms for the migration of our mobile agent with our migration scheme.

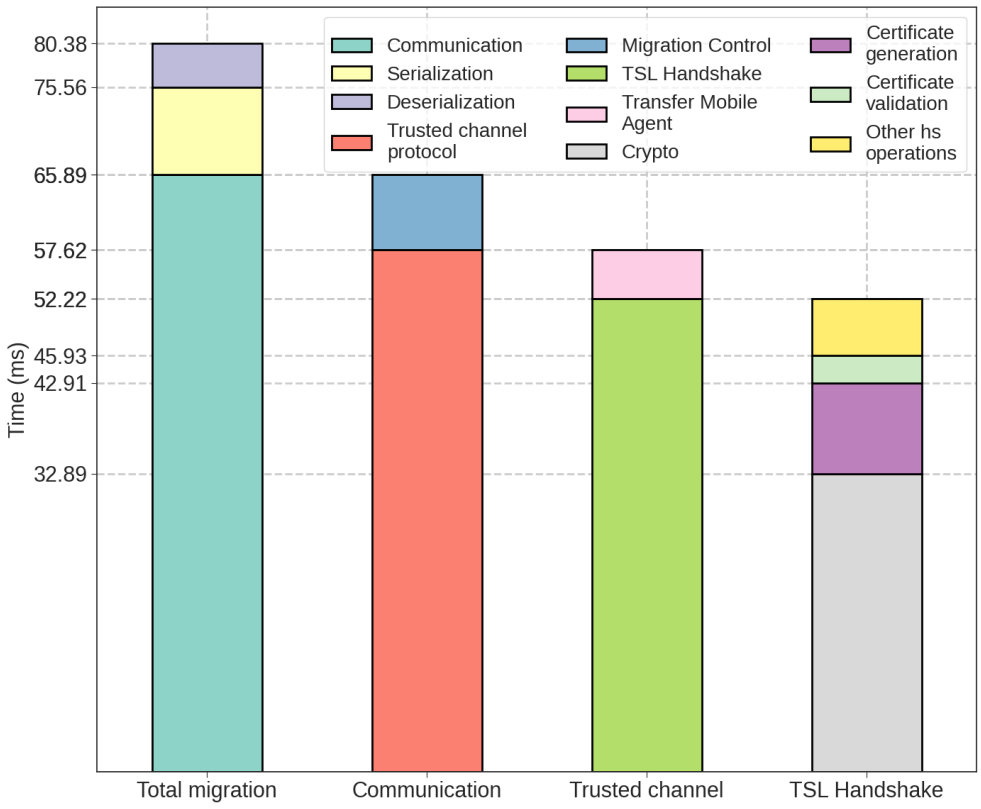

Figure 7.1: Distribution of the average migration time for our mobile agent. The bars are partially inclusive to provide more details on each operation performed.

relocated. In this configuration, the measurements for every mobile agent size were run 100 times and on both platforms. The results are illustrated in Figure 7.2 and are represented on a logarithmic scale for a better visualization for the reader.

The figure is able to show not only the performance discrepancy of the two devices, but also the impact of heavy computational operations, such as the compression algorithm used in the serialization procedure. Moreover, from the graph, it is possible to perceive that the impact of the trusted channel handshake and its

suboperations is significant compared to the total migration time until the mobile agent size is about 63-92KB. After that point, the time required for the migration is increasing proportionally to the size of the mobile agent.

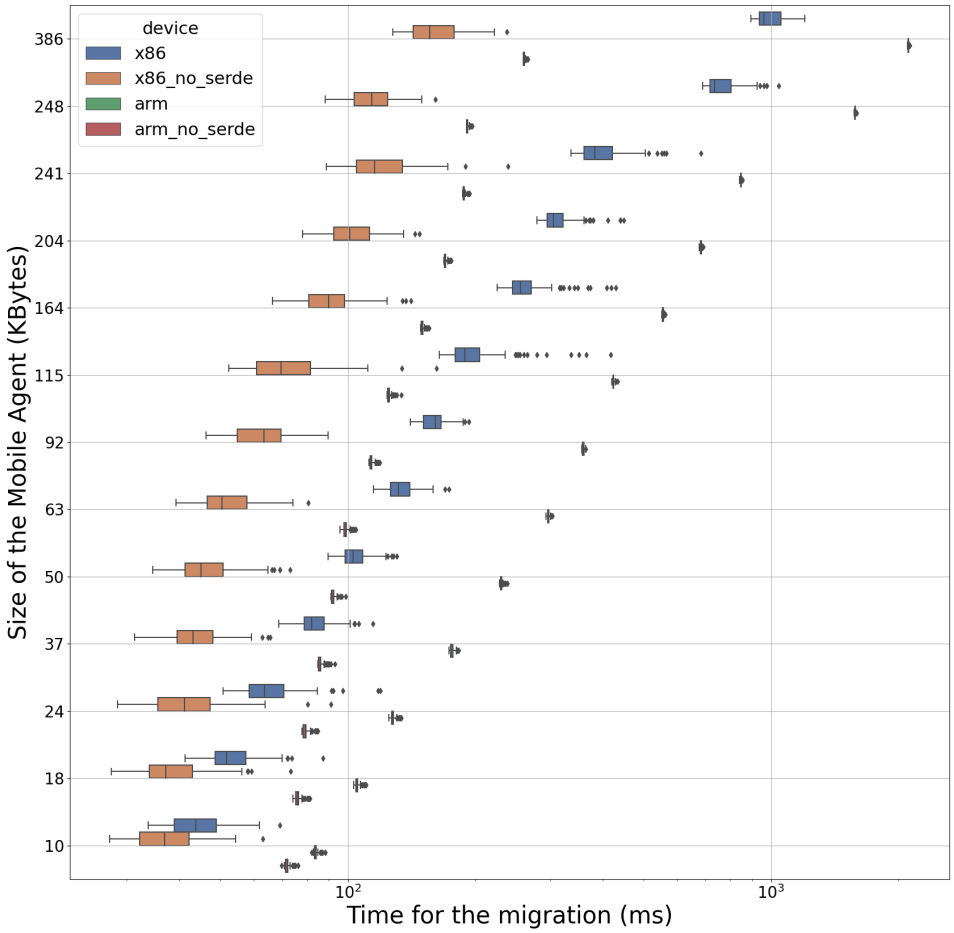

Figure 7.2: Mobile Agent size vs. Migration time with logaritmic scale. The graph compares the migration time required for mobile agents with different sizes on both platforms with and without the compression in the serialization phase.

7.3 Security analysis

The security requirements for the execution of the mobile agent are satisfied by the Secure Execution Environment, the WebAssembly sandboxing, and the migration scheme. In particular, the security requirements stated in Section 5.1.1 about the migration of mobile agents are satisfied as per below:

- **SR1 (Secure channel).** Our migration scheme employes the Trusted Sockets Layer library, which is an extension to a size-optimized TLS 1.3 library. As such the TSL library follows the TLS 1.3 specification, thus fulfilling the requirements of a secure channel. Furthermore, the endpoint of the TSL connection is placed inside the secure enclave, and so also the execution of the TSL endpoint (secrets included) is protected by attacks from the network and untrusted environment.

- **SR2 (endpoint trustworthiness).** In our migration scheme the attestation process includes the TCB, enclave runtime, and optionally the mobile agent. If just one of these modules fails to be attested, the trusted channel cannot be established. The trustworthiness is confirmed during the validation process of the attestation. The attestation reference is provided along with the attestation evidence in order to be used in the validation process. The references are signed by a trusted third-party authority (i.e. an attestation CA) and represent good values for the attestation evidence.

- **SR3 (Channel binding).** As specified in the Trusted Sockets Layer protocol [18], the binding of the attestation to the trusted channel is performed by including the challenge computed by the TLS-Exporter in the attestation. The challenge is strongly bound to each trusted channel handshake since its value is affected by handshake-specific values. In our migration scheme, the challenge is included in the evidence of each attested component, in such a

way that for each connection a new attestation must be provided. For the same reason, relay attacks are not possible. Even if an attacker A is able to extract an attestation from a valid endpoint P, the attestation will be valid only in the specific TSL handshake between A and P, and cannot be reused in another connection between A and the victim without detection.

- **SR4 (Privacy).** The privacy of the TSL endpoints is provided by both, the trusted channel protocol and the SEE architecture. The trusted channel protocol protects the identities and attestation evidence of TSL endpoints against eavesdroppers from the network. Meanwhile the hardware-assisted isolation protects them against attacks from the untrusted environment on the local platform.

  The privacy of the mobile agent is always protected on the local platform since its execution is performed only inside the secure enclave. Moreover, the mobile agent cannot be fooled to migrate on a compromised secure enclave since before its replacement the target platform must be attested. Thus, a compromised secure enclave will be able to provide just invalid attestations, which will prevent the trusted channel to be established.

- **SR5 (Forward security).** TLS 1.3 does not allow re-using the same endpoint authentication ephemeral key for multiple handshakes and the TSL protocol does not allow resumption. Thus, each session has a unique key and handshake that must include the mutual attestation of the endpoints. As such the only long-term secret that an attacker may try to compromise is the enclave runtime key but this is kept in the SEE and so it is protected against attackers from the untrusted environment. If an attacker is able to compromise the runtime, the attacker will not be able to receive or send mobile agents, since it cannot provide a valid attestation. However, even if the runtime gets compromised, the mobile agents sent in the past can be still considered legitimate, since at

that time the attestations were valid and so the runtime was not compromised.

8 Conclusions

In this thesis, a design and implementation for migrating architecture-agnostic mobile agents between secure execution environments (SEEs) was presented. The thesis work was part of a secure enclave framework project at Huawei Technologies Oy Helsinki Research Center. The main problem addressed in this thesis was that current SEEs are usually dependent on specific hardware architectures, which leads to several limitations for the development of applications executing within SEEs. That is because a new implementation is required for each architecture. In the project and the thesis, a solution was developed where the above limitations had been solved. The solution consists of running architecture-agnostic mobile agents within hardware isolated secure enclaves that are implemented across different platforms. My personal contributions to the project can be summarised as consisting of two points. On one hand, I contributed by participating in the design process of the migration scheme, by implementing the TSL library, by implementing the communication channel between the secure enclave and the untrusted environment, by taking part in the PoC implementation, and by performing the benchmarks. On the other hand, the thesis itself contributes by providing a detailed description of the problem, the migration architecture and the benchmarks results.

By definition, mobile agents are software programs able to move between multiple devices. Our migration scheme empowers the mobility capabilities of mobile agents between secure enclaves. In order to ensure that the security of the mobile agent

is not compromised during its relocation, the migration scheme employs a trusted channel protocol. The protocol makes use of mutual attestation and strong channel binding as a means to guarantee the trustworthiness of the endpoints. In such a way, the mobile agent can have reliable proof that the system, where it is relocating, will ensure a high level of security for its execution.

The proof-of-concept of the secure migration scheme was implemented as part of the project and the thesis work and demonstrates the relocation of a mobile agent between secure enclaves on different platform architectures. The performance analysis performed on the migration scheme reveals concrete measurements of the time employed for the secure migration of mobile agents. Proper implementation and measurements of the time required in the computation of the attestation claims is not addressed in this thesis, but it could be part of a discussion to be addressed in future work.

Finally, the PoC demonstrates that a live migration of mobile agents between secure enclaves has a relatively low overhead, around $80ms$ per migration. Therefore, it can be concluded that the presented architecture and implementation techniques seem suitable for real applications such as the offloading techniques in Mobile Edge Computing or the transfer of secrets between personal IoT and mobile devices.

References

[1] K. Akherfi, M. Gerndt, and H. Harroud, "Mobile cloud computing for computation offloading: Issues and challenges", *Applied Computing and Informatics*, vol. 14, no. 1, pp. 1–16, Jan. 2018.

[2] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—A key technology towards 5G", *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[3] Y. Guo, C. Jiang, T.-Y. Wu, and A. Wang, "Mobile agent-based service migration in mobile edge computing", *International Journal of Communication Systems*, vol. 34, no. 3, e4699, 2021.

[4] *Java Agent DEvelopment Framework*, Aug. 2021. [Online]. Available: `https://jade.tilab.com/` (visited on 08/29/2021).

[5] L. Martin, "XTS: A Mode of AES for Encrypting Hard Disks", *IEEE Security Privacy*, vol. 8, no. 3, pp. 68–69, May 2010.

[6] E. Rescorla and T. Dierks, "The transport layer security (TLS) protocol version 1.3", Tech. Rep. RFC 8446, 2018.

[7] G. Spafford, "Attributed to in Risks Digest 19.37 review of @LARGE, by David H. Freedman and Charles C. Mann, Sept. 1997", *The RISKS Digest, Volume 19 Issue 37*, vol. 19, no. 37, Sep. 1997.

[8] Ponemon Institute LLC, *The Third Annual Study on the State of Endpoint Security Risk*, 2020. [Online]. Available: https://www.morphisec.com/hubfs/2020%20State%20of%20Endpoint%20Security%20Final.pdf.

[9] B. Pranggono and A. Arabo, "COVID-19 pandemic cybersecurity issues", *Internet Technology Letters*, vol. 4, no. 2, e247, 2021.

[10] A. Segall, *Trusted Platform Modules: Why, when and how to Use Them*, ser. Computing and Networks. Institution of Engineering and Technology, 2016. [Online]. Available: https://books.google.fi/books?id=uY-EDQAAQBAJ.

[11] N. Modadugu and B. Richardson, *Building a Titan: Better security through a tiny chip*, May 2021. [Online]. Available: https://android-developers.googleblog.com/2018/10/building-titan-better-security-through.html (visited on 05/19/2021).

[12] M. Gulati, M. J. Smith, and S.-Y. Yu, "Security enclave processor for a system on a chip", US8832465B2, Sep. 2014. [Online]. Available: https://patents.google.com/patent/US8832465/en (visited on 05/19/2021).

[13] W. Li, Y. Xia, and H. Chen, "Research on ARM TrustZone", *GetMobile: Mobile Computing and Communications*, vol. 22, no. 3, pp. 17–22, Jan. 2019.

[14] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, *Innovative Technology for CPU Based Attestation and Sealing*. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.405.8266&rep=rep1&type=pdf.

[15] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption", *White paper*, 2016.

[16] A. Ltd, *Arm Confidential Compute Architecture*. [Online]. Available: https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture (visited on 09/26/2021).

[17] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An Embedded Trusted Runtime for WebAssembly", *arXiv:2103.15860 [cs]*, Mar. 2021.

[18] A. Niemi, V. A. B. Pop, and J.-E. Ekberg, "Trusted Sockets Layer: A TLS 1.3 based trusted channel protocol", *accepted to NordSec 2021*, 2021.

[19] V. A. B. Pop, A. Niemi, V. Manea, A. Rusanen, and J.-E. Ekberg, "Towards Secure Mobile Agents", *submission to EuroSys22*, 2021.

[20] *Confidential Computing Consortium - Open Source Community*, Apr. 2021. [Online]. Available: https://confidentialcomputing.io/ (visited on 04/14/2021).

[21] S. Samonas and D. Coss, "THE CIA STRIKES BACK: REDEFINING CONFIDENTIALITY, INTEGRITY AND AVAILABILITY IN SECURITY.", *Journal of Information System Security*, vol. 10, no. 3, 2014.

[22] R. B. Lee, "Security Basics for Computer Architects", *Synthesis Lectures on Computer Architecture*, vol. 8, no. 4, pp. 1–111, Sep. 2013.

[23] J. Szefer, *Principles of secure processor architecture design*, ser. Synthesis lectures on computer architecture 45. Morgan and Claypool, 2019.

[24] Quickbase, *Information Security: A Closer Look*, May 2021. [Online]. Available: https://www.quickbase.com/articles/information-security-a-closer-look (visited on 05/14/2021).

[25] *Title 44 - PUBLIC PRINTING AND DOCUMENTS; CHAPTER 35 - COORDINATION OF FEDERAL INFORMATION POLICY; SUBCHAPTER III - INFORMATION SECURITY*. [Online]. Available: https://www.govinfo.gov/content/pkg/USCODE-2011-title44/pdf/USCODE-2011-title44-chap35-subchapIII.pdf.

[26] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote Attestation Procedures Architecture", *RATS Working Group*, Jul. 2020.

[27] F. Petitcolas, *La cryptographie militaire*, 1883. [Online]. Available: `https://www.petitcolas.net/kerckhoffs/crypto_militaire_1.pdf`.

[28] D. McGrew and J. Viega, "The Galois/counter mode of operation (GCM)", *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[29] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm", in *Advances in Cryptology — ASIACRYPT 2000*, T. Okamoto, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2000, pp. 531–545.

[30] A. Maximov and D. Khovratovich, "New State Recovery Attack on RC4", in *Advances in Cryptology – CRYPTO 2008*, D. Wagner, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2008, pp. 297–316.

[31] D. J. Bernstein, "The Salsa20 Family of Stream Ciphers", in *New Stream Cipher Designs: The eSTREAM Finalists*, M. Robshaw and O. Billet, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 84–97.

[32] D. J. Bernstein *et al.*, *ChaCha, a variant of Salsa20*, 2008. [Online]. Available: `http://cr.yp.to/chacha/chacha-20080120.pdf`.

[33] S. Almuhammadi and I. Al-Hejri, "A comparative analysis of AES common modes of operation", in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, Apr. 2017, pp. 1–4.

[34] J. Daemen and V. Rijmen, *AES proposal: Rijndael*, 1999. [Online]. Available: `https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf`.

[35] E. Biham, R. Anderson, and L. Knudsen, "Serpent: A New Block Cipher Proposal", in *Fast Software Encryption*, S. Vaudenay, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1998, pp. 222–238.

[36] R. J. Anderson, E. Biham, and L. R. Knudsen, "The Case for Serpent.", in *AES Candidate Conference*, 2000, pp. 349–354. [Online]. Available: `https://www.cl.cam.ac.uk/~rja14/Papers/serpentcase.pdf`.

[37] A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique Cryptanalysis of the Full AES", in *Advances in Cryptology – ASIACRYPT 2011*, D. H. Lee and X. Wang, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 344–371.

[38] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's Algorithm to AES: Quantum Resource Estimates", in *Post-Quantum Cryptography*, T. Takagi, Ed., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 29–43.

[39] A. Bogdanov, L. R. Knudsen, G. Leander, *et al.*, "PRESENT: An Ultra-Lightweight Block Cipher", in *Cryptographic Hardware and Embedded Systems - CHES 2007*, P. Paillier and I. Verbauwhede, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2007, pp. 450–466.

[40] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and Efficiently Searchable Encryption", in *Advances in Cryptology - CRYPTO 2007*, A. Menezes, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2007, pp. 535–552.

[41] W. Diffie and M. Hellman, "New directions in cryptography", *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[42] R. Haakegaard and J. Lang, "The elliptic curve diffie-hellman (ecdh)", 2015. [Online]. Available: https://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf.

[43] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[44] C. Gidney and M. Ekerå, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits", *arXiv:1905.09749 [quant-ph]*, Apr. 2021. (visited on 04/20/2021).

[45] *NIST's Post-Quantum Cryptography Program Enters 'Selection Round'*, text, Jul. 2020. [Online]. Available: https://www.nist.gov/news-events/news/2020/07/nists-post-quantum-cryptography-program-enters-selection-round (visited on 04/20/2021).

[46] M. J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", Aug. 2015. (visited on 04/20/2021).

[47] K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang, "Preimages for Step-Reduced SHA-2", in *Advances in Cryptology – ASIACRYPT 2009*, M. Matsui, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2009, pp. 578–597.

[48] C. Dobraunig, M. Eichlseder, and F. Mendel, "Analysis of SHA-512/224 and SHA-512/256", in *Advances in Cryptology – ASIACRYPT 2015*, T. Iwata and J. H. Cheon, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2015, pp. 612–630.

[49] G. M. Lilly, "Device for and method of one-way cryptographic hashing", US6829355B2, Dec. 2004. [Online]. Available: https://patents.google.com/patent/US6829355B2/en (visited on 05/17/2021).

[50] H. Tiwari *et al.*, "Merkle-Damgård Construction Method and Alternatives: A Review", *Journal of Information and Organizational Sciences*, vol. 41, no. 2, pp. 283–304, 2017.

[51] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Oxon, OX: CRC Press, Dec. 2020.

[52] A. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, *Ron was wrong, Whit is right*, 2012. [Online]. Available: `https://infoscience.epfl.ch/record/174943`.

[53] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices", in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 205–220.

[54] D. Genkin, A. Shamir, and E. Tromer, "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis", in *Advances in Cryptology – CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2014, pp. 444–461.

[55] S. u. Hassan, I. Gridin, I. M. Delgado-Lozano, *et al.*, "Déjà Vu: Side-Channel Analysis of Mozilla's NSS", in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1887–1902.

[56] J. Weise, *Public key infrastructure overview*, 2001. [Online]. Available: `http://highsecu.free.fr/db/outils_de_securite/cryptographie/pki/publickey.pdf`.

[57] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X. 509 public key infrastructure certificate and CRL profile, January 1999", Tech. Rep. RFC 2459.

[58] G. Coker, J. Guttman, P. Loscocco, *et al.*, "Principles of remote attestation", *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, Jun. 2011. (visited on 04/21/2021).

[59] *Protocol*, Apr. 2021. [Online]. Available: `https://www.britannica.com/technology/protocol-computer-science` (visited on 04/20/2021).

[60] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", Internet Engineering Task Force, Request for Comments RFC 5280, May 2008.

[61] B. Gates, *Bill Gates: Trustworthy Computing*, Apr. 2021. [Online]. Available: `https://www.wired.com/2002/01/bill-gates-trustworthy-computing/` (visited on 04/21/2021).

[62] T. ALVES, "TrustZone : Integrated Hardware and Software Security", *White Paper*, 2004.

[63] J. G. Beekman, "Improving Cloud Security using Secure Enclaves", Ph.D. dissertation, UC Berkeley, 2016. [Online]. Available: `https://escholarship.org/uc/item/0jn0n1xs` (visited on 04/06/2021).

[64] O. Demigha and R. Larguet, "Hardware-based solutions for trusted cloud computing", *Computers & Security*, vol. 103, p. 102 117, Apr. 2021. (visited on 03/31/2021).

[65] F. Alder, A. Kurnikov, A. Paverd, and N. Asokan, "Migrating SGX Enclaves with Persistent State", in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2018, pp. 195–206.

[66] N. D. Matsakis and F. S. Klock, "The rust language", *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, Oct. 2014.

[67] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System Programming in Rust: Beyond Safety", in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17, New York, NY, USA: Association for Computing Machinery, May 2017, pp. 156–161.

[68] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs", *arXiv:2003.03296 [cs]*, Feb. 2021. (visited on 04/24/2021).

[69] H. Wang, P. Wang, Y. Ding, *et al.*, "Towards Memory Safe Enclave Programming with Rust-SGX", in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 2333–2350.

[70] A. Haas, A. Rossberg, D. L. Schuff, *et al.*, "Bringing the web up to speed with WebAssembly", in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, New York, NY, USA: Association for Computing Machinery, Jun. 2017, pp. 185–200.

[71] D. Brown, *WebAssembly Security, Now and in the Future*, Mar. 2021. [Online]. Available: `https://training.linuxfoundation.org/announcements/webassembly-security-now-and-in-the-future/` (visited on 04/25/2021).

[72] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly", 2020, pp. 217–234.

[73] M. Vassena and M. Patrignani, "Memory Safety Preservation for WebAssembly", *arXiv:1910.09586 [cs]*, Oct. 2019.

[74] D. Sitaram and G. Manjunath, "Chapter 9 - Related Technologies", in *Moving To The Cloud*, D. Sitaram and G. Manjunath, Eds., Boston: Syngress, Jan. 2012, pp. 351–387.

[75] A. Rubin and D. Geer, "Mobile code security", *IEEE Internet Computing*, vol. 2, no. 6, pp. 30–34, Nov. 1998.

[76] *CORBA History*. [Online]. Available: `https://www.corba.org/history_of_corba.htm` (visited on 09/05/2021).

[77] P. W. Fong, "Viewer's discretion: Host security in mobile code systems", 1998. [Online]. Available: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.30.4493&rep=rep1&type=pdf`.

[78] A. Carzaniga, G. P. Picco, and G. Vigna, "Designing distributed applications with mobile code paradigms", in *Proceedings of the 19th international conference on Software engineering*, ser. ICSE '97, New York, NY, USA: Association for Computing Machinery, May 1997, pp. 22–32.

[79] *Java Applet Tutorial - javatpoint*. [Online]. Available: `https://www.javatpoint.com/java-applet` (visited on 09/06/2021).

[80] *JavaScript | MDN*. [Online]. Available: `https://developer.mozilla.org/en-US/docs/Web/JavaScript` (visited on 09/06/2021).

[81] E. Elliott, *How Popular is JavaScript in 2019?*, Jul. 2019. [Online]. Available: `https://medium.com/javascript-scene/how-popular-is-javascript-in-2019-823712f7c4b1` (visited on 09/06/2021).

[82] *Rsh(1): Remote shell - Linux man page*. [Online]. Available: `https://linux.die.net/man/1/rsh` (visited on 09/06/2021).

[83] *AWS Lambda – Serverless Compute - Amazon Web Services*. [Online]. Available: `https://aws.amazon.com/lambda/` (visited on 09/06/2021).

[84] A. Outtagarts, "Mobile Agent-based Applications : A Survey", *International Journal of Computer Science and Network Security*, vol. 9, Jan. 2009.

[85] Y. Teranishi, "PIAX: Toward a Framework for Sensor Overlay Network", in *2009 6th IEEE Consumer Communications and Networking Conference*, Jan. 2009, pp. 1–5.

[86] A. Boulis, C.-C. Han, R. Shea, and M. Srivastava, "SensorWare: Programming sensor networks beyond code update and querying", *Pervasive and Mobile Computing*, vol. 3, pp. 386–412, Aug. 2007.

[87] *TACOMA - Operating system support for agents*. [Online]. Available: `http://www.tacoma.cs.uit.no/index.html` (visited on 09/06/2021).

[88] K. Kravari and N. Bassiliades, "A Survey of Agent Platforms", *Journal of Artificial Societies and Social Simulation*, vol. 18, Jan. 2015.

[89] T. Bayer and C. Reich, "Security of Mobile Agents in Distributed Java Agent Development Framework (JADE) Platforms", in *ICONS 2017*, Venice, Italy, Apr. 2017.

[90] N. Asokan, "Hardware-assisted Trusted Execution Environments: Look Back, Look Ahead", in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, New York, NY, USA: Association for Computing Machinery, Nov. 2019, p. 1687.

[91] *What is a Payment Hardware Security Module (HSM)? | Thales*. [Online]. Available: `https://cpl.thalesgroup.com/faq/hardware-security-modules/what-payment-hardware-security-module-hsm` (visited on 05/04/2021).

[92] S. Mavrovouniotis and M. Ganley, "Hardware Security Modules", in *Secure Smart Embedded Devices, Platforms and Applications*, K. Markantonakis and K. Mayes, Eds., New York, NY: Springer, 2014, pp. 383–405.

[93] N. I. o. S. a. Technology, "Security Requirements for Cryptographic Modules", U.S. Department of Commerce, Tech. Rep. FIPS 140-2, Dec. 2002.

[94] K. Schaffer, "FIPS 140-3 Derived Test Requirements (DTR): CMVP Validation Authority Updates to ISO/IEC 24759", Tech. Rep. FIPS 140-3, 2019.

[95] I. T. L. Computer Security Division, *FIPS 140-3 Transition Effort | CSRC | CSRC*, Jul. 2019. [Online]. Available: `https://csrc.nist.gov/projects/fips-140-3-transition-effort` (visited on 05/04/2021).

[96] G. Madlmayr, O. Dillinger, J. Langer, C. Schaffer, C. Kantner, and J. Scharinger, "The benefit of using SIM application toolkit in the context of near field communication applications", in *International Conference on the Management of Mobile Business (ICMB 2007)*, Jul. 2007, pp. 5–5.

[97] M. Reveilhac and M. Pasquet, "Promising Secure Element Alternatives for NFC Technology", in *2009 First International Workshop on Near Field Communication*, Feb. 2009, pp. 75–80.

[98] G. Intelligence, *Understanding SIM evolution*, 2015. [Online]. Available: `https://data.gsmaintelligence.com/api-web/v2/research-file-download?id=18809300&file=understanding-sim-evolution-1482139874006.pdf`.

[99] T. Mandt, M. Solnik, and D. Wang, "Demystifying the secure enclave processor", *Black Hat Las Vegas*, 2016.

[100] A. Martin, *The ten-page introduction to Trusted Computing*, 2008. [Online]. Available: `https://ora.ox.ac.uk/objects/uuid:a4a7ae67-7b2a-4516-801d-9379d613bab4`.

[101] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems", in *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020, pp. 1416–1432.

[102] J. A. Halderman, S. D. Schoen, N. Heninger, *et al.*, "Lest we remember: Cold-boot attacks on encryption keys", *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, May 2009.

[103] T. Groß, M. Busch, and T. Müller, "One key to rule them all: Recovering the master key from RAM to break Android's file-based encryption", *Forensic Science International: Digital Investigation*, DFRWS 2021 EU - Selected Papers and Extended Abstracts of the Eighth Annual DFRWS Europe Conference, vol. 36, p. 301 113, Apr. 2021.

[104] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: Reading Kernel Memory from User Space", in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, 2018, pp. 973–990.

[105] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre Attacks: Exploiting Speculative Execution", in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1–19.

[106] R. Buhren, C. Werling, and J.-P. Seifert, "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation", in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1087–1099.

[107] *Arm Introduces Its Confidential Compute Architecture*, Jun. 2021. [Online]. Available: `https://fuse.wikichip.org/news/5699/arm-introduces-its-confidential-compute-architecture/` (visited on 09/26/2021).

[108] *Arm Confidential Compute Architecture software stack*. [Online]. Available: `https://developer.arm.com/documentation/den0127/a` (visited on 09/26/2021).

[109] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments", in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–16.

[110] K. Suzaki, K. Nakajima, T. Oi, and A. Tsukamoto, "Library Implementation and Performance Analysis of GlobalPlatform TEE Internal API for Intel SGX and RISC-V Keystone", in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Dec. 2020, pp. 1200–1208.

[111] D. Brumley and D. Boneh, "Remote timing attacks are practical", *Computer Networks*, Web Security, vol. 48, no. 5, pp. 701–716, Aug. 2005. (visited on 04/23/2021).

[112] B. B. Brumley and N. Tuveri, "Remote Timing Attacks Are Still Practical", in *Computer Security – ESORICS 2011*, V. Atluri and C. Diaz, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2011, pp. 355–371.

[113] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 1996, pp. 104–113.

[114] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical", in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 605–622.

[115] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack", in *23rd USENIX Security Symposium*

(USENIX Security 14), San Diego, CA: USENIX Association, 2014, pp. 719–732.

[116] J. V. Bulck, M. Minkin, O. Weisse, *et al.*, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution", in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, 2018, pp. 991–1008.

[117] Y. Kim, R. Daly, J. Kim, *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors", *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, Jun. 2014.

[118] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management", in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, 2017, pp. 1057–1074.

[119] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack", in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17, New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1–6.

[120] B. Sean, *TRUSTNONE*. [Online]. Available: `http://theroot.ninja/disclosures/TRUSTNONE_1.0-11282015.pdf`.

[121] B. Gal, *Bits, Please!: TrustZone Kernel Privilege Escalation (CVE-2016-2431)*, Jun. 2016. [Online]. Available: `http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html` (visited on 04/23/2021).

[122] D. Rosenberg, *Qsee trustzone kernel integer over flow vulnerability*, 2014. [Online]. Available: `https://wikileaks.org/sony/docs/05/docs/Hacks/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf`.

[123] D. Shen, *Exploiting trustzone on android*, 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf.

[124] A. Machiry, E. Gustafson, C. Spensky, *et al.*, *BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.* 2017. [Online]. Available: https://www.researchgate.net/profile/Chad-Spensky/publication/316913734_BOOMERANG_Exploiting_the_Semantic_Gap_in_Trusted_Execution_Environments/links/592db6a7a6fdcc89e752a89b/BOOMERANG-Exploiting-the-Semantic-Gap-in-Trusted-Execution-Environments.pdf.

[125] Gal, Beniamini, *Project Zero: Trust Issues: Exploiting TrustZone TEEs*, Jul. 2017. [Online]. Available: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html (visited on 04/23/2021).

[126] *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions.* [Online]. Available: https://www.freertos.org/index.html (visited on 09/27/2021).

[127] *Lobaro/FreeRTOS-rust*, Jun. 2021. [Online]. Available: https://github.com/lobaro/FreeRTOS-rust (visited on 07/13/2021).

[128] *Tasks and Co-routines [Getting Started].* [Online]. Available: https://www.freertos.org/taskandcr.html (visited on 09/11/2021).

[129] *Paritytech/wasmi*, Jul. 2021. [Online]. Available: https://github.com/paritytech/wasmi (visited on 07/13/2021).

[130] *Wasmer - The Universal WebAssembly Runtime.* [Online]. Available: https://wasmer.io/ (visited on 07/13/2021).

[131] *Security - WebAssembly*, Mar. 2021. [Online]. Available: https://webassembly.org/docs/security/ (visited on 03/31/2021).

[132] W. M. Farmer, J. D. Guttman, and V. Swarup, "Security for mobile agents: Issues and requirements", in *Proceedings of the 19th national information systems security conference*, vol. 2, Baltimore, Md., 1996, pp. 591–597.

[133] W. Jansen and A. Karygiannis, *Mobile agent security*, 1999-10-01 1999. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=151186.

[134] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC Editor, Tech. Rep. RFC 8949, Dec. 2020, RFC8949.

[135] *Serde - Rust*. [Online]. Available: https://docs.serde.rs/serde/index.html (visited on 07/16/2021).

[136] I. Rec, *X. 680 Information technology–Abstract Syntax Notation One (ASN. 1): Specification of basic notation*. [Online]. Available: https://www.itu.int/ITU-T/studygroups/com17/languages/X.680-0207.pdf.

[137] *Secure Boot and Image Authentication (Whitepaper)*, 2019. [Online]. Available: https://www.qualcomm.com/media/documents/files/secure-boot-and-image-authentication-technical-overview-v2-0.pdf.

[138] *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*, 2002. [Online]. Available: https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf.

[139] *Project-oak/oak*, Jul. 2021. [Online]. Available: https://github.com/project-oak/oak (visited on 07/05/2021).

[140] *Std::thread - Rust*. [Online]. Available: https://doc.rust-lang.org/std/thread/index.html (visited on 07/22/2021).

[141] *Std::sync::mpsc - Rust*. [Online]. Available: https://doc.rust-lang.org/std/sync/mpsc/index.html (visited on 07/22/2021).

[142] *Std::net - Rust*. [Online]. Available: https://doc.rust-lang.org/std/net/index.html (visited on 07/22/2021).

[143] *XXTEA*, Apr. 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=XXTEA&oldid=1018895659 (visited on 09/11/2021).

[144] *Queues for task and interrupt message passing in FreeRTOS real time embedded software applications*. [Online]. Available: https://www.freertos.org/Embedded-RTOS-Queues.html (visited on 09/11/2021).

[145] *Freertos_rust::Queue - Rust*. [Online]. Available: https://docs.rs/freertos-rust/0.1.2/freertos_rust/struct.Queue.html (visited on 09/11/2021).

[146] *Smart Pointers - The Rust Programming Language*. [Online]. Available: https://doc.rust-lang.org/book/ch15-00-smart-pointers.html (visited on 09/11/2021).

[147] *Compression::prelude::BZip2Encoder - Rust*. [Online]. Available: https://chalharu.github.io/rust-compression/compression/prelude/struct.BZip2Encoder.html (visited on 09/11/2021).

[148] *Intel® VT: Intel® Virtualization Technology - What is Intel® VT? |...* [Online]. Available: https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html (visited on 09/11/2021).

[149] *Learn the architecture: AArch64 Virtualization*. [Online]. Available: https://developer.arm.com/documentation/102142/0100/Stage-2-translation (visited on 09/11/2021).

[150] *Learn the architecture: AArch64 Exception model*. [Online]. Available: `https://developer.arm.com/documentation/102412/0100/Privilege-and-Exception-levels` (visited on 09/11/2021).