POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Direct Anonymous Attestation

Supervisor
Prof. Antonio Lioy
Dott. Ignazio Pedone

Candidate
Lorenzo De Siena

December 2022

Contents

Chapter 1

Introduction

Today, the rapid expansion of distributed systems, such as Internet of Things and Cloud Computing, is bringing up increasing issues in keeping systems secure and trustworthy. Internet of Things refers to physical devices connected to Internet providing specific services to individuals and companies. Such devices represent critical elements, as they generate a big amount of data to be stored and processed and are rarely updated. Moreover, since they are produced by a large variety of companies, they are not well standardized, thus making it particularly difficult their trustworthiness. Therefore, the number of potentially malicious devices connected to the Internet and, often placed inside homes and companies, is drastically increasing.

Another steadily increasing phenomenon is the use of Cloud Computing as new paradigm for the distribution of IT services. Indeed, the Cloud allows to easily and almost instantaneously create infrastructures and systems based on virtualized objects, that enable services to scale with agility and in a more cost-effective way than past solutions. Moreover, Cloud objects are often automatically distributed worldwide. The Cloud Provider autonomously manages the cloud infrastructure and takes care of its security and provisioning. However, this leads to the need for Trust towards the Cloud Provider and the infrastructure, which, in turn, results in new challenges and in the need for new instruments able to identify non expected behaviours in the infrastructure itself.

One of the tools to achieve a better degree of security and trust is the Trusted Platform Module (TPM), that allows verification of the system state through Remote Attestation. The latter is a process through which a platform, the Attester, demonstrates its integrity state to an external entity called Verifier. Instead, the TPM consists of a tamper-resistant device containing a secure crypto processor devoted to the protection of integrity measurements. They derive from measurements of platform software and settings which are processed through cryptographic hash algorithms and then stored in the so called Platform Configuration Registers (PCRs). The Remote Attestation can evaluate the state of a system including all software executed from the startup until the boot phase or, thanks to the support of the Operative System (OS), until the moment of the attestation. In order to do so, the OS monitors and measures all security critical files which are opened

or modified and then creates a log of events containing the related measurements. In Linux, the module dedicated to this function is called Integrity Measurement Architecture (IMA). During the attestation phase, the host sends it to the Verifier along with an attestation of PCR registers, so that it verifies the values of PCR registers contained in the attestation and validate its status.

The attestation is based on the generation of a digital signature by the TPM on the state of its own PCR registers. The signature is made with an Attestation Key (AK) which is certified by a third-party (CA) after having verified the host identity, usually through the TPM Endorsement Key (EK). This allows the CA to have a link between the Attester identity and its emitted signatures. Inside the attestation, especially when IMA log is used, some sensible information could be present. This could be sufficient to monitor the user of the platform, thus giving rise to the need to create a privacy-preserving way of performing system state attestation.

The Direct Anonymous Attestation (DAA) is a set of cryptographic schemes that allow remote authentication preserving, at the same time, the platform privacy. The DAA is based on the use of group signatures, i.e. anonymous digital signatures that can be generated after acquiring a credential from the group membership issuer (Issuer) and becoming a group Member. Such signatures can be verified by a Verifier who will not be able to identify the signer among group members. The DAA also grants the ability to revoke any malicious credentials and invalidate additional signatures. The difference with traditional digital signatures stands in the fact that group signatures possess a mechanism that randomizes the key used for the creation of the signature based both on the credential released by the Issuer and the Member secret DAA key. Therefore, the Issuer, even if generating and knowing the credentials, is not able to distinguish the credential which the signature has been generated from, so it cannot distinguish the signer identity.

The first DAA protocol, based on symmetric Pairings on Elliptic Curve (ECC-DAA), was proposed by Brickell, Chen, and Li [1] and later on adopted in the TPM 2.0 specification. The latter was created with the idea to define the essential and generic API to allow DAA execution, leaving most of the computational part to the host platform which the TPM is part of. This allowed to minimize the performance impact due to the TPM limited computational capacity and, moreover, it allowed to make the DAA implementation flexible, resulting in the possibility to define new future protocols with the same API.

DAA has also the capability of optionally creating anonymous signatures having linking capabilities, i.e. giving to the Verifier the capability to understand if two different signatures were created by the same Signer. This allows to relax the signature privacy properties and to use the DAA in a wider range of applications.

The aim of this thesis is to create a simple Remote Attestation Framework based on DAA, which allows to execute the attestation in the closest possible way to the traditional Remote Attestation, both in terms of performance and process. Moreover, our goal is to make it work in a Cloud oriented virtual environment and to create functions that could be integrated in the pipelines of deploy of the

various distributed objects to obtain objects that autonomously execute DAA with an external Verifier.

For this purpose, we have created an implementation of a hybrid DAA scheme from the implementation provided by Wesemeyer et al. [2] based on Chen and Li's proposed ECC-DAA scheme. The hybrid scheme brings pseudo-anonymity advantages provided by DAA using linking capabilities and at the same time allows for the use of pre-existing, higher performance implementations typical of traditional Remote Attestation.

The implementation consists of 6 modules that execute the various steps of DAA and perform the functions of the different actors involved. The Issuer and Verifier have been implemented as http servers that expose the APIs to perform a DAA. The group Member, instead, performs the setup of the TPM and, by contacting the provided APIs, joins the group requesting to the Issuer a credential. Then it generates, certifies, and exchanges an Attestation key (AK) with the Verifier. Once the AK has been agreed upon, the Member can perform cycles of remote attestation with the Verifier providing an attestation of PCR registers associated with the BIOS and IMA Measurement Log. Lastly, the Verifier checks the attestation and the measurements logs alerting eventual validation problems.

1.1 Structure of the report

The following document uses a bottom-up approach, by first introducing the technologies used and then presenting the design and implementation of the solution. This is followed by functional and performance tests on the developed solutions. Lastly, the final considerations and the future works are discussed.

In chapter 2 we present the TPM starting from the concepts of Root of Trust, we describe its architecture and the key management. This is followed by an overview of TCG Software Stack (TSS). We mainly focus on the fundamental aspects necessary for understanding the developed implementation.

In chapter 3 we describe the Remote Attestation, the actors and the modalities with which it is used. Moreover, we define the Integrity Measurement Architecture (IMA) focusing on the Measurement log file, its components and relationship with PCR registers and TPM.

The chapter 4 deals with the standard ISO 20008:2013 concerning the anonymous digital signature, in particular the group signature. We show the common characteristics among the various mechanisms present in the standard and we describe the linking-capabilities and open-capabilities. We tackle the problem of credential revocation and we discuss the various existent revocation mechanisms. Lastly, we summarize the properties of the standardized mechanisms.

In chapter 5 we focus on the Direct Anonymous Attestation, in particular on the [2] scheme which will be used in the implementation. We also describe all protocol steps and the various types of signature that can be created. In conclusion, we discuss several security problems of such scheme.

In chapter 6 we examine the implementations design and the reasons for our implementation choices. It is also present an overview of any application and communication protocol among them.

In chapter 7 we discuss the choices, projects, and libraries useful in the implementation. In particular, we will discuss the ecc-daa project on which we relied and the projects provided by IBM as utilities for projects that use TPM.

The chapter 8 shows some functional tests useful also to understand the functioning of the applications, moreover, some performance tests are proposed in order to obtain some measurements on their execution times.

Lastly, chapter 9 contains the conclusions and the final comments on the implementation. It is then followed by advice for future works to be able to improve the current solution or integrate it in other attestation frameworks.

Chapter 2

Trusted Platform Module

TPM is a specification conceived by Trusted Computing Group (TCG) [3], an initiative of a large group of IT companies with the aim to promote the development of Trusted Computing by relying on both hardware and software implementations. TPM 1.1b, released in 2002, was the first major version of the specification. It already included the main fundaments of the current standard, such as PCR registers (and the concept of Root of Storage), key generation, cryptographic operations and *remote attestation* (and the concept of Root of Reporting). In 2004, the TPM 1.2 version was published. For the first time, *Direct Anonymous Attestation* was introduced as a new method for remote attestation. Lastly, TPM 2.0 was released in 2014 but it is still continuously updated. Among its new features, it is noticeable the introduction of algorithm agility and the support for Elliptic Curve Cryptography (ECC). Nowadays, almost every PC, smartphone and server has a built-in TPM on its motherboard or CPU.

2.1 Trust

For TCG, "trust" is meant to convey an expectation of behavior [3]. The TPM in a computer reports a result of the measurements made on the software and hardware in a way that allows to determine if the system is behaving as expected and, consequentially, to establish the trust.

Trusted Building Block

The TCG defines the Trusted Building Blocks (TBB) as the collection of components necessary to form the Root of Trust [3]. For example, in a PC, the TBB is represented by the combination of the Core Root of Trust for Measurement (CRTM), the connection between the CRTM storage and the motherboard, the path that connects the storage of the CRTM and the CPU, the connection between the TPM and the motherboard, and the connection path between the TPM and the CPU. It is critical that the TBB is reliable and does not affect the trust goal of the platform.

Roots of Trust

TCG defines Roots of Trust primarily as the system elements that must be trusted because a misbehavior would not be detectable. Secondly, the set of Roots of Trust is the minimum set of functionalities necessary to allow the measurement of the trustworthiness of the system [3]. The TCG identifies three Roots of Trust in a trusted platform:

- **Root of Trust for Storage (RTS)** is defined as a trusted place to store data that can be changed only by adhering to strict security policies. The TPM shield its memory from external entity access, in fact, some of the data in the TPM memory locations are not sensitive and the TPM does not protect it from disclosure, for example in the case of PCR values. Instead, other data are sensitive and the TPM does not allow access without proper authorization, for example a private part of a key [3]. Since the TPM can be trusted to prevent inappropriate access to its memory, the TPM can act as an RTS;

- **Root of Trust for Measurement (RTM)** oversees sending integrity-relevant information (measurements) to the RTS [3]. This is typically played by the host's CPU controlled by the first set of instructions executed when a new chain of trust is established (typically on system reset). These instructions are called Core Root of Trust for Measurement (CRTM);

- **Root of Trust for Reporting (RTR)** reports on the contents of RTS. An RTR report is typically a digitally signed digest of the contents of selected and accessible values within a TPM [3].

Transitive trust

Whether Roots of Trust establishes the reliability of an application that is about to be executed and, in turn, the application establishes the reliability of a subsequent application, we can say that we have achieved the *transitive trust*, due to the fact that we can trust applications based on the trust of the Roots of Trust [3]. The first way to reach transitive trust is to making sure of the trustworthiness of the next application through a *trust policy* executed before running the next application. A second way is to make sure that the measurements taken before the execution of the next application are available to an independent evaluation who will establish the trust later on. The TPM may support either of these methods.

2.2 TPM architecture

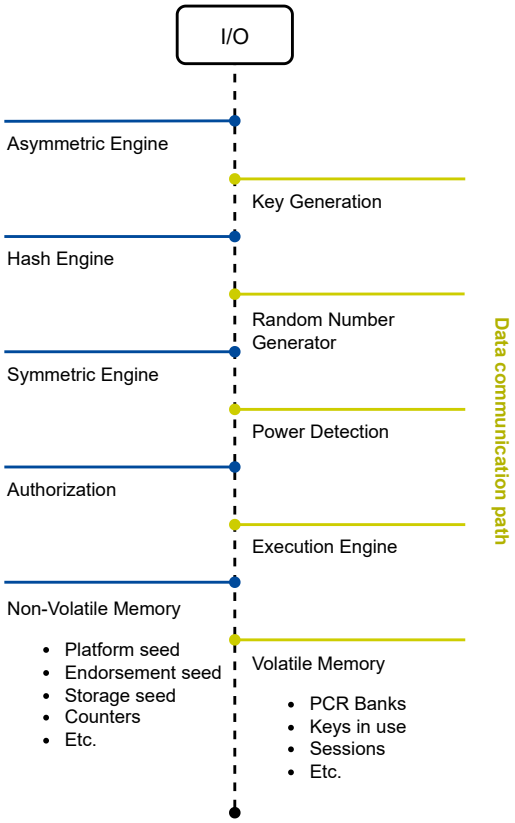

Figure 2.1. TPM 2.0 architecture

In order for the implementation to respect the TPM specification, as shown in Figure 2.1, it is necessary that it contains all the following components [3], whose mutual interaction allows the operation of the TPM:

- **I/O buffer**: it is the component through which the host interacts with the TPM by sending and receiving data.
- **Hash engine**: used both by the host or as part of other TPM operations, it performs generic hashing operations.
- **Random Number Generator (RNG)**: it is the source of randomness in the TPM. The TPM uses RNG for the generation of nonces, keys, and randomness required in signatures. It is interesting to note that, since the RNG is a critical component in the infrastructure, the TPM supports more than one entropy source. It is the manufacturer's discretion how to use this potential.
- **Key generation**: it allows to generate keys of the various protocols supported by the implementation. Keys can be generated from the RNG or from hierarchy seeds using approved Key Derivation Function (KDF).

- **Asymmetric and symmetric engines**: they are used to perform symmetric and asymmetric cryptographic operations like encryption, decryption, signature and signature verification.
- **Authorization Subsystem**: it oversees the execution of authorization checks before allowing a command to run.
- **Power Detection**: it detects changes in power status and allows the TPM and host to detect them.
- **Volatile memory**: it is the memory used to hold data whose lifespan ends upon TPM restart. Examples are PCRs, current sessions, loaded temporary keys, etc.
- **Non-volatile memory**: this is the memory area that contains data that persist even when the power is removed. Here are contained sensitive data such as the values of the seeds and any persistent key. Because of this, these areas contain Shielded Locations which can be accessed only through Protected Capabilities. Depending on the implementation, there may be enough free memory left to allow applications to make permanent some transient object, for example application-specific keys.

2.2.1 Platform Configuration Registers

The **Platform Configuration Registers (PCRs)** are internal registers of the TPM designed to contain the result of a hash and which present a known value at start-up, usually initialized with all 0s or all 1s. The registers are grouped into banks and each bank can be configured at TPM startup by specifying which hash algorithm that bank should use in *TPM2_Extend* procedure to change the PCR value. Each register within a bank is identified by an index starting at 0 through which the particular register can be referenced at the time of its use. There can be a variable number of banks and registers, however, the specification states that in the *PC Client Platform profile* the TPM must contain at least one bank of at least 24 registers [3].

The PCR value can be updated only through the TPM2_Extend procedure that consists in updating the values of a PCR register on the basis of the already present value and of an input value, as following [3]:

$$PCR_{new} = Hash_{bankAlg}(PCR_{old} \| inputvalue)$$

The PCR value reset to the initial value at the TPM reset or in case of resettable registers, usually those with index greater than 15, through the explicit call to *TPM2_pcr_reset*.

2.3 TPM Hierarchies

The TPM uses hierarchies to organize objects so that they can be related and managed as groups [4]. At the root of the hierarchies there are *primary keys*, while

under them there may be *non-primary keys*, forming a tree. The cryptographic root of the hierarchy is the *primary seed*: a random number that is generated and kept secret by the TPM. The seed is used to generate primary keys. In turn, primary keys act as encryption keys for the non-primary keys placed under them.

Hierarchies can be volatile, i.e. erased at each reboot, or persistent, thus retained at the reboot [4]. Three persistent hierarchies are defined: platform, storage and endorsement. Each one of them is utilized for specific use cases.

Platform hierarchy

The *Platform hierarchy* is intended to be under the control of the platform manufacturer who can exercise it through the code used in the initial stages of platform startup. To allow this, it displays a unique feature compared to other hierarchies. At each restart, it is enabled by default and has an authorization value set with an empty password. In fact, since the intent is to be available to the platform firmware, no authorization is required to use it in the first boot phase. After starting and until the next reboot, the platform firmware decides whether to keep it enabled and, if necessary, what policy to adopt in order to use it [4].

Endorsement hierarchy

The *Endorsement hierarchy* is the privacy-sensitive hierarchy to be used for what concerns the user privacy. The TPM vendor and the platform vendor certify that the primary keys of this hierarchy, called Endorsement Keys (EKs), relate to an authentic TPM mounted on an authentic platform. A primary key from this hierarchy can only be used in signing operations and not in encryption and decryption operations. Creating and certifying an EK are privacy sensitive operations as they allow the key to be traced back to the particular TPM [4].

Storage hierarchy

The *Storage hierarchy* is meant to be used by the platform owner. It is intended for non-privacy-sensitive operations [4].

NULL hierarchy

The *NULL hierarchy* is a special hierarchy, as it is volatile and does not require any authorization to be used. In fact, its primary seed changes at each restart, making unusable all previously created keys. This allows to use the TPM as a cryptographic co-processor, which executes cryptographic algorithms on externally generated keys after loading them into the hierarchy [4].

2.4 Key generation

One of the most interesting features of the TPM is the ability to generate cryptographic keys and protect their secrets by not allowing them to leave the security perimeter. The generation of the keys is based on the internal RNG. Indeed, it is not possible to generate the keys with other sources of randomness. This guarantees that each key that is produced is derived from a reliable source of entropy and cannot be negatively affected by any careless application [4].

2.4.1 Key and Primary Key generation

All keys that have a parent are wrapped (encrypted) with the parent key and returned to the caller. This avoid the need for the TPM to memorize those keys, which, instead, can also be stored in untrusted locations by the caller.

As mentioned above, at the root of a hierarchy there are primary keys that, as such, have no parents. The TPM allows the creation of an unlimited number of primary keys thanks to the use of primary seeds. In the generation process of a primary key, the primary seed of the hierarchy, together with a set of data that characterize the generation of the key, the *public template*, are given as input to a *Key Derivation Function* (KDF) [4].

The *public template* consists of the description of the key to be generated, the algorithm, the length of the key, the policy and the type of key (encryption, signature, etc.). Furthermore, a *unique data* can be added by the application to allow the creation of two different keys, despite using the same template.

2.4.2 Persistence

The keys can possibly be persisted in the non-volatile memory of the TPM through the *EvictControl* procedure. This allows to have the key already available in the internal memory, rather than otherwise have to be loaded or re-generated at each use [4].

2.4.3 Duplication attributes

The TPM allows in some circumstances to move a key from one parent to another one, even if this belongs to a different hierarchy or different TPM. This process is referred to as *key duplication*. When a key is duplicated, all the children wrapped by it, and their descendants, become available in the destination. There are two attributes of each key that control the ability to perform a key duplication [4]:

- **fixedTPM**: if this parameter is active, the key can never be duplicated.
- **fixedParent**: in case it is active, the key cannot be duplicated directly, but only through a duplication of its parent.

2.4.4 Restricted singing key

The keys created in the TPM can be used to sign a user input or, in the case of special keys called *restricted singing keys*, also to attest data from internal data structures such as PCR registers or keys. These special keys are obtainable by specifying the "*restricted*" attribute when creating the key. For security reasons, the specification mandates that restricted keys can only be used for signing and cannot for other purposes such as encryption and decryption of data.

The restricted key can actually sign any input, even if not generated by the TPM, as long as it does not start with 4 fixed special bytes called *TPM_GENERATED* [3]. This is possible due to a requirement imposed by the specification, that is, in order for a signature to be performed on user input, the hash it is trying to sign must have been created by the TPM. To do this, TPM provides *TPM2_Hash*, which allows to compute an hash over a user input and also returns a ticket with which the TPM, at the signing stage, can verify that it has created the input hash. Thanks to this requirement, the TPM can fully verify the input that it will then go on to sign and it can be ensured that the input does not start with *TPM_GENERATED*. In doing so, the specification reserves the use of the special bytes exclusively as a prefix to the attestations of its internal structure thus guaranteeing that if the signature was created by a restricted key then the attestation and signature were created by the TPM itself.

2.5 TCG Software Stack

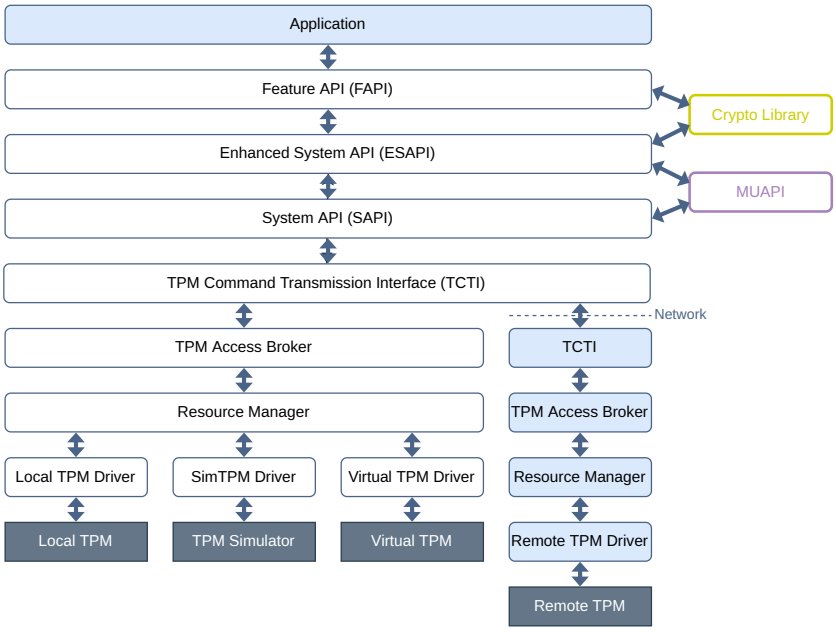

Figure 2.2. TSS stack overview

To avoid low-level communications with the TPM and to simplify the development of applications that use the TPM, the *TCG Software Stack* (TSS) [5] has been standardized. It is a multi-level stack in which each level offers services at the higher one. This allows the calling application not to deal with some problems related to the use of the TPM itself.

Indeed, one of the most critical aspects of the TPM is the fact that it is a system with limited resources. Therefore, in order to carry out complex operations, it is necessary that the caller takes care of explicitly managing resources. Another issue is the inability of the TPM to autonomously manage two callers at the same time. Thus, if necessary, an external actor has to perform resources switching. As shown in Figure 2.2, the TSS consists of:

- **TPM Device Driver**: the layer in charge of sending commands and receiving responses to a particular TPM[5];
- **Resource Manager**: is the layer that deals with resource management of the TPM. Indeed, the TPM has limited resources, so, when needed, it has to handle the loading of objects, sessions and policies and, when no longer needed, to flush them for allowing the execution of another call with the available resources [5]. Its actions must be transparent to the upper layers of the TSS and its presence is not mandatory. For example, when a single-application is running, such as BIOS, the application itself may be sufficient to manage the resources of the TPM;
- **TPM Access Broker (TAB)**: manages, on the other hand, simultaneous access to the TPM. Its purpose is to ensure that, regardless of the number and the order in which the various application calls reach the TPM, the result will be the same as if the TPM were called by only one application at a time without interference between the various calls [5];
- **TPM Command Transmission Interface (TCTI)**: manages all communications towards the TPM in a standard way, regardless of the type of underlying TPM, being hardware, virtual, simulated or remote. There are currently two standard interfaces: the *Command/Response Buffer* (CRB) and the legacy *TIS interface* [5];
- **Marshaling/Unmarshalling (MUAPI)**: is in charge of creating the byte streams required by the TPM, starting from properly populated data structures (marshalling) and decomposing the response byte streams from the TPM into the appropriate data structures (unmarshalling) [5]. It is used by both SAPI and ESAPI and, for this reason, it maintains its own separate API;
- **System API (SAPI)**: is an API layer that permits access to all the functions of the TPM [5]. It was created with the aim of being used by all those software aiming at interacting at a lower level with the TPM, namely BIOS, firmware, OS, expert application, etc.;

- **Enhanced System API (ESAPI)**: is an interface built on top of the System API with the main purpose of reducing the complexity required to the applications working with low-level calls to the TPM. It is needed that the calling application is able to perform cryptographic operations [5]. In fact, to reduce the complexity of the communication, ESAPI prevents users to personally handle complex interactions with the TPM, such as encrypted sessions or TPM policies. Although it presents a marked improvement in terms of ease of use compared to the System API, it is still necessary to know in depth the details of interfacing with the TPM:

- **Feature API (FAPI)**: represents the API layer designed to be used by generic applications. It aims to make usable as many TPM functions as possible without knowledge of TPM low-level details [5].

Chapter 3

Remote attestation

A system which underwent an involuntary or malicious alteration of its parts could constitute a threat. Once having defined the integrity requirements of an intact system, it is possible to use them as parameters to determine whether a system is corrupted or not, and eventually take compensatory measures to eliminate the threat. Since a system in an incorrect state may have compromised its own protection systems, it is not sufficient for the system to self-check in order to guarantee detection of an undesired state.

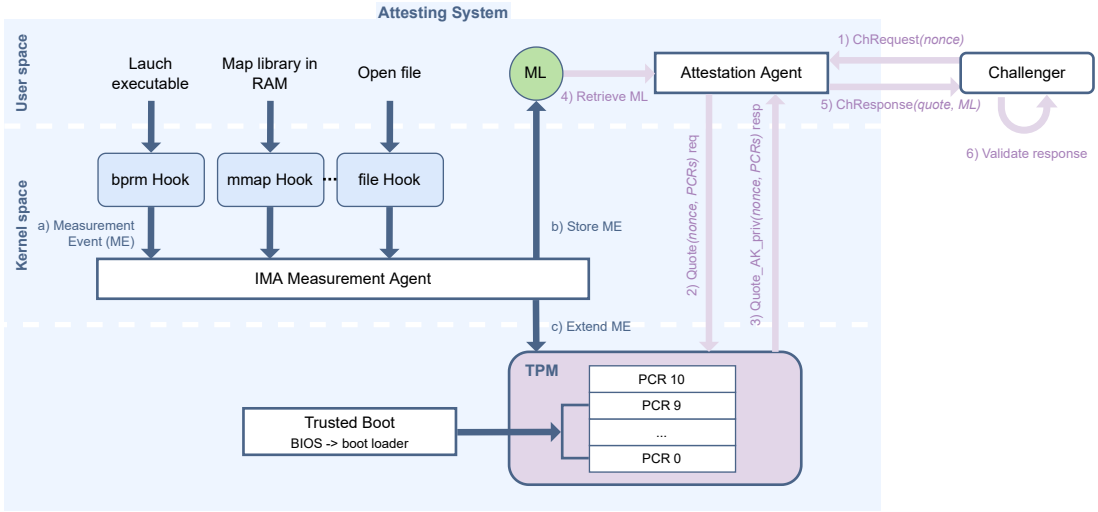

Figure 3.1. Remote attestation

The *Remote Attestation* mechanism allows a remote challenger (the Verifier) to certify the status of a system, that is, to understand if that system meets certain integrity requirements. In detail, the Verifier challenges the system to be certified, the *Attestation Agent*, with a nonce that guarantees the freshness of the response. The Attestation Agent asks the TPM for an attestation of the PCR registers, called Quote, usually of the ones with an index from 0 to 10, signed with a previously

certified Attestation Key (AK). The Attestation Agent can add to the Quote a series of additional information to support the Verifier, allowing the assessment of the validity of the values contained in the PCRs. As we will see, on Linux this information includes the IMA measurement log file. The Quote along with the additional information compose the *Integrity Report* which is sent to the Verifier. The Verifier validates the *Integrity Report* and verifies whether the Quote is fresh and authentic, thus demonstrating if the system is in a healthy state. An overview of the entire process in a Linux environment is presented in the Figure 3.1.

3.1 Measured Boot

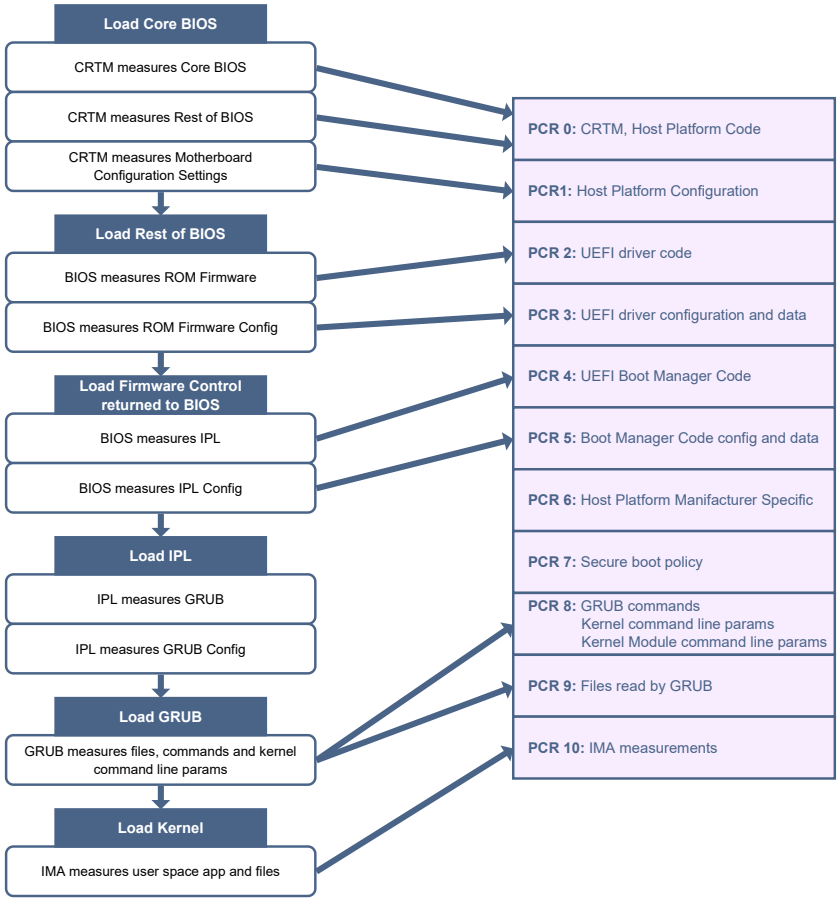

Figure 3.2. Measured boot

As already discussed in section 2.1, the role of the RTM is to send the relevant information to the RTS. In the implementation, the platform CPU sends information to the TPM to update the PCR registers through the *Extend* procedure, which, as mentioned, is not reversible.

When the TPM and the platform startup, the PCR registers are in a known default state. The CRTM, which is the code that is first executed in the host

CPU, *measures* both the software that will be run immediately after it and the configurations that might affect its behavior. The measurements will be extended into the PCR registers. The following software, the BIOS for example, will, in turn, perform the same operation on the software to be subsequently executed, for example the boot loader, thus extending the measurements in other PCR registers. The process continues in this way until the operating system is fully booted. At each step each software extends new PCR registers with the measurements obtained. The PCR registers index, in which the measurements are extended, are standardized by the TCG and can be found in Figure 3.2. If the measurements are correctly performed and the resulting values in each step represent a trusted state of the system, for *transitive trust*, all the chain will be also trustworthy. This approach generates the so-called *Chain of Trust*. Therefore, if the CRTM is trusted, we can have a measure of the state of the system and validate its correctness.

It should be noted that the TPM has no ability to directly interfere with the boot process. The running system, on the other hand, can influence its execution based on the status of the measurements in the TPM. In fact, the process can use the state of the TPM to authorize sequential software components in the startup chain. This can be achieved, for example, by verifying digital signatures of software components to be performed based on a list of trusted keys. This type of validation of the boot process is called **Secure Boot**. Instead, we refer to the process as **Trusted Boot** if the possibility of completing the system startup is constrained to a verification by an external entity (a Verifier), which, via *remote attestation*, verifies the validity of the system status.

3.2 Integrity Measurement Architecture

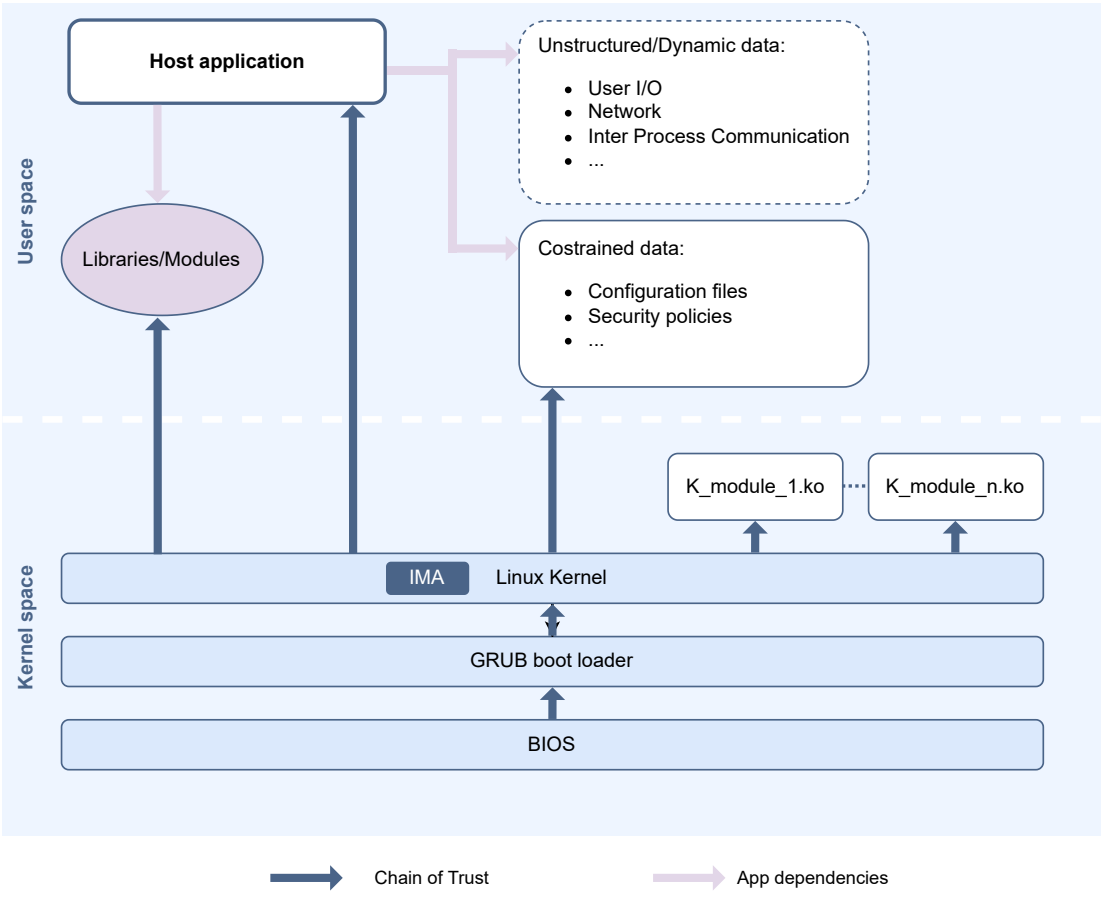

Figure 3.3. Ima overview

The **Integrity Measurement Architecture (IMA)** [6], whose architecture is shown in Figure 3.3, is a Linux kernel module that is part of the *Linux Integrity Subsystem*. It was introduced in 2009 as a TCG-compliant implementation for measuring dynamic executable contents. It allows the extension of the *Chain of Trust* from the BIOS to the application layer and brings the concepts of Measured Boot to the kernel level allowing to:

- *Detect* both remotely and locally, within the limits of the case, accidentally or maliciously modified files (executables, configurations, kernel modules, etc.);
- *Appraising* a file measurement to a golden value inserted directly in the attributes of the file itself;
- *Enforcing* local file integrity.

The IMA is designed to be the centralized point of the kernel in which the OS perform the measurement of files, do the storage of the measurement, extend the PCR registers, make appraise of the measurement with respect to golden values, and audit of results.

3.2.1 Measurement Log file

In boot process, it is easily possible to identify a well-defined ordered sequence of measurements to be carried out. Once the operating system is started, instead, a large amount of software components are simultaneously executed in an indeterminable order, leading to an indefinable order of measurements. It is therefore important to find a way to allow an external Verifier to check the correctness of the system status, taking into account that the measurements order and, thus, of the final measurement, may be different even in the presence of a non-malicious machine.

For this reason, the IMA manages a *Measurement Log file* (ML) consisting of a list of all the *measurement events* persisted in the order in which they are measured by the kernel. The ML is sent to the Verifier during the remote attestation phase and it is used to verify that the values contained in the PCRs indicate an integral state of the system. There are two versions of the ML file [6]:

- a binary version, usually located under, `/sys/kernel/security/ima/binary_runtime_measurements`;
- a human-readable version, usually located under `/sys/kernel/security/ima/ascii_runtime_measurements`.

3.2.2 Fields description

PCR	template-hash	template	filedata-hash	filename-hint
10	91f34b5c671d73504b274a919661cf80dab1e127	ima-ng	sha1:1801e1be3e65ef1eaa5c16617bec8f1274eaf6b3	boot_aggregate
10	8b1683287f61f96e5448f40bdef6df32be86486a	ima-ng	sha256:efdd249edec97caf9328a4a01baa99b7d66...	/init
10	ed893b1a0bc54ea5cd57014ca0a0f087ce71e4af	ima-ng	sha256:1fd312aa6e6417a4d8dcdb2693693c81892...	/usr/lib64/ld-2.16.so
10	9051e8eb6a07a2b10298f4dc2342671854ca432b	ima-ng	sha256:3d3553312ab91bb95ae7a1620fedcc69793...	/etc/ld.so.cache

Table 3.1. ASCII Measurement Log file example

The ASCII version is similar to Table 3.1. Each measurement event is a line in the ML. Each record contains a certain number of fields depending on the template used because each template can use different types of file measurements. To date, the standard provides 3 base templates [7] [6]:

- **ima**: only supports SHA-1 as a digest for filedata-hash;
- **ima-ng**: supports digests other than SHA-1 for filedata-hash;
- **ima-sig**: additionally includes the file signature in the file-signature field.

The main fields are described as follows:

- **PCR index**: always present, it contains the index of the PCR register that has been extended by this measurement;
- **template-name**: always present, it contains the name of the template used for the measurement;
- **filedata-hash**: always present, it contains the hash of the file calculated at the time of the measurement. It can support various formats depending on the template used. In case of "ima-ng" and "ima-sig" template, it is possible to specify which hash to use ("sha1", "md5", "sha256", "sha512", "wp512", ...) that would be inserted as a hash prefix in the data-hash file. The "ima" template, on the other hand, does not contain any prefix since it only supports SHA-1 and does not insert any prefix;
- **filename-hint**: always present, it contains the absolute path of the measured file, except for special cases, such as "boot-aggregate", where it represents a reference to the concatenation of the hashes of the post-boot PCR registers;
- **file-signature**: only in the case of an "ima-sig" template, signature of the file contained in the "security.ima" attribute of the file itself;
- **template-hash**: always present, it is normally a SHA-1 hash of the content of the "template-data", that is a binary structure containing all the measurement data specific to a particular template (file-name, file-hash, hash-size, signature etc...). In some special cases it can be a string containing all 0s. In the case of an "ima" template, on the other hand, it is calculated as the SHA-1 hash of the concatenation of the "filedata-hash", of the "filename-hint" plus a quantity of 0s to reach a total length of 256 bytes.

3.2.3 PCR Extend

Given a new measurement, there are currently two different ways of extending a PCR register. Since there seems not to exists a standard way of calling them, we just call them extends "*Type 1*" and extends "*Type 2*" [8].

Type 1 (zero-pad)

In case of SHA-1 PCR banks:

- if the "template-hash" is made up of all 0s: a digest made up of all 1s is used to extend the PCR register;
- otherwise, if it is not made up of all 0s: the "template-hash" is passed to the extend as it is.

In case of SHA-256 PCR banks, it behaves as in the PCR SHA-1 banks but, before passing the result to the extends, this is padded with 12 Bytes to 0s thus obtaining a 256-bit string at the ends.

- if the "template-hash" is made up of all 0s: a binary string made up of 20 bytes at 1 and 12 bytes at 0s is generated;
- otherwise, if it is not composed of all 0s: the "template-hash" concatenated with 12 Bytes is passed to 0s.

Type 2 (hash)

In case of SHA-1 PCR banks, we proceed with the same logic as Type 1.

- if the "template-hash" is made up of all 0s: a digest made up of all 1s is passed to the ends of the PCR register;
- otherwise, if it is not made up of all 0s: the "template-hash" is passed to the extends as it is.

In case of SHA-256 PCR banks:

- if the "template-hash" is made up of all 0s: a digest made up of all 1s is passed to the ends of the PCR register.
- otherwise, if it is not composed of all 0s: the value passed to the extend is a hash generated from the "template-data", with a process equal to the "template-hash" one, but this time using the hash of the PCR bank (SHA-256) instead of SHA-1.

It can be noticed that, in Type 2, the generation of the string to be extended in the PCR can be independent from SHA-1. In contrast, Type 1, always uses SHA-1 in the computation of the "template-hash". Type 2 on the other hand follows a generic process that is the same for all bank types. The only variation is the hash used for calculating "template-data" that matches that of the bank used.

3.2.4 IMA in Remote Attestation

Immediately after adding an element to the ML, the IMA extends the PCR registers in the TPM (usually only PCR 10 is used), according to the configuration of the IMA itself. In this way, the ML always contains more measurements than the ones that are actually extended in the PCR registers. This allows the Verifier to examine the ML and simulate the extension of the registers made by the system, by looking for that measurement for which the value of the registers corresponds to that of the TPM Quote [8].

Thanks to this approach, a misalignment of some measurements between the PCR registers and the ML can occur. This allows to verify that the PCR registers contain a value that can be calculated from the ML regardless of the exact moment in which the Quote takes place, which could be after updating the ML but before updating the TPM registers. At the same time, it detects any misalignment that must be extended in the following remote attestation cycles to prove that the system is healthy.

Chapter 4

Anonymous Digital Signature (ISO 20008:2013)

In conventional digital signature, it is possible to create a signature that cannot be traced back to the signer. Conceptually, if the private key is in the possession of more than one entity, it would not be possible to distinguish which owner created the signature. However, in a system that uses this method to hide the identity of the signer, it is sufficient that even one of the owners becomes malicious or discloses the key to make the whole system unreliable. In fact, there is no way to revoke the ability of a private key holder to issue new signatures. The only solution would be to create a new key pair and redistribute the keys.

The ISO 20008 standard [9] [10] proposes alternative ways to the conventional digital signatures to create Anonymous Digital Signature mechanisms. In particular, we will focus on the so called *group signatures*. As it is defined in the standard [9], an *Anonymous Digital Signature* is a signature which can be verified using a *group public key* (in the case of *group signatures*), or multiple public keys (for *ring signatures*), and which cannot be traced to the distinguishing identifier of its signer by any unauthorized entity, including the signature verifier.

4.1 Ring signature: multiple public keys

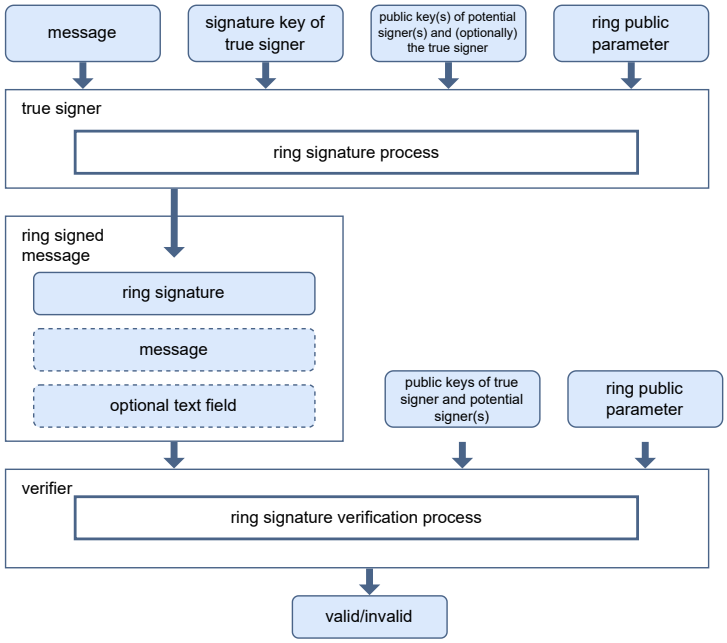

Figure 4.1. Signature and verification processes for a ring signature mechanism [9]

Ring signatures are characterized by a certain number of *ring members* whose public keys are known and allow to hide the identity of the signer (the *true signer*). The signature is based on the true signer's private key and the public keys of the other Members of the ring (the *potential signers*). The true signer can sign regardless of the willingness of the other Members of the ring and without even letting them know [9]. An overview of the processes is proposed in Figure 4.1.

In order to verify a signature, it is necessary to have all the public keys of the Members of the ring. In this way, it is possible to validate the signature and understand if the signer belongs to the ring or not but without the possibility of identifying him.

A disadvantage of ring signatures is that the complexity of the signature and its verification is proportional to the number of ring Members. In addition, in dynamic rings, where new Members can join or leave the ring, all ring Members and all Verifiers must be notified so that signatures can be generated and validated accordingly.

4.2 Group signature: group public key

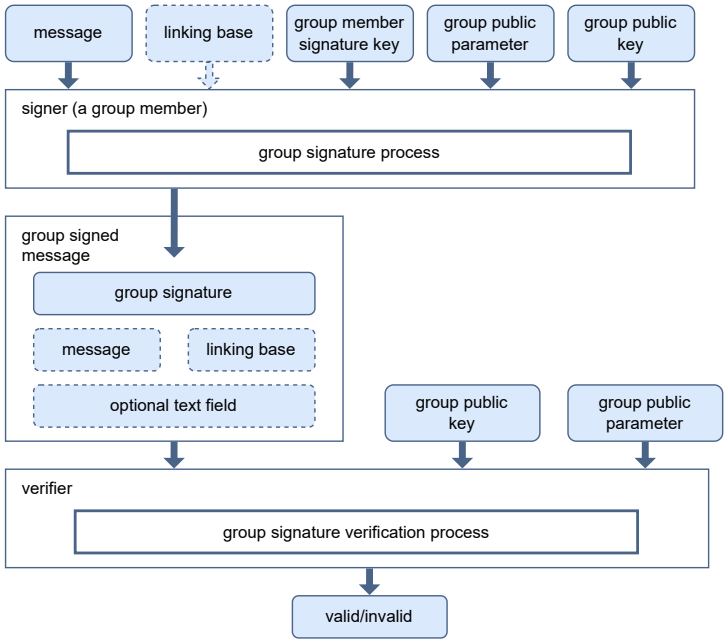

Figure 4.2. Signature verification processes in an anonymous signature mechanism using a group public key [9]

Group signatures, on the other hand, are based on a single public key, the *group public key*, which allows a Verifier to validate any signature issued by Members of the group. Creating and verifying a group signature is more complex and requires several steps [9], Figure 4.2 present an overview of the processes.

The standard [10] defines 7 implementation *mechanisms* for obtaining a group signature. Each mechanism has its own unique features but shares many common characteristics on which categorizations can be created. First of all, there are common actors and procedures that we are going to examine below.

4.2.1 Group membership issuing process

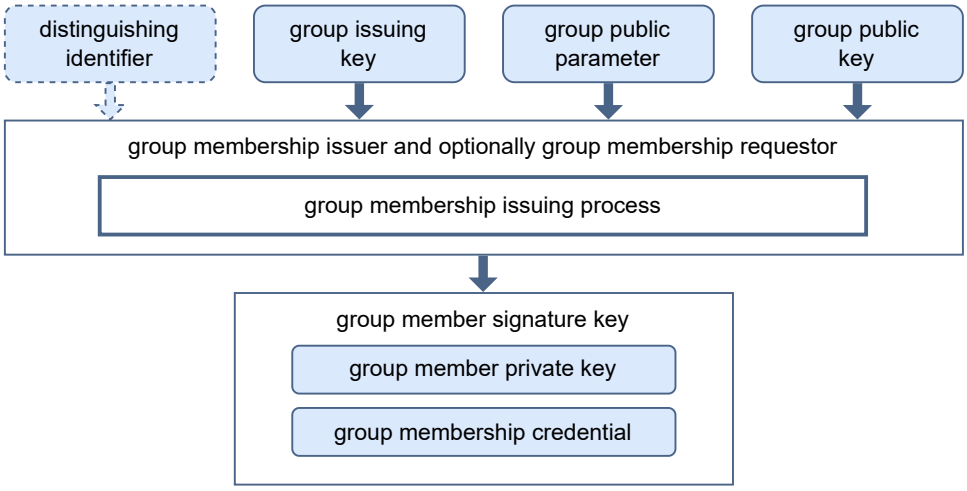

Figure 4.3. Group membership issuing process [9]

The possibility of introducing new Members into the group is entrusted to the *group membership issuer* (or *Issuer*) which keeps the *group issuing key* that allows the generation of the *group membership credential* [9].

The process, visible in Figure 4.3, can be done entirely by the Issuer or through a joint work of the candidate Member and the Issuer. In the second case, a Member who wants to join the group creates a *group member private key* and contacts the Issuer who optionally verifies the identity of the candidate Member through a *distinguishing identifier* and finally provides the *group membership credential* to the Member [9].

Only the combination of the *group member private key* and the *group membership credential* provided by the Issuer can generate a valid signature. This combination is called *group member signature key* [9] .

This means that the Issuer must be trusted by everyone because it could authorize any malicious Members, but also that if he does not know the *group member private key* of a Member he would still not be able to impersonate him. In this case, the mechanism would possess the "non-repudiation" property [9].

4.2.2 Linking capabilities

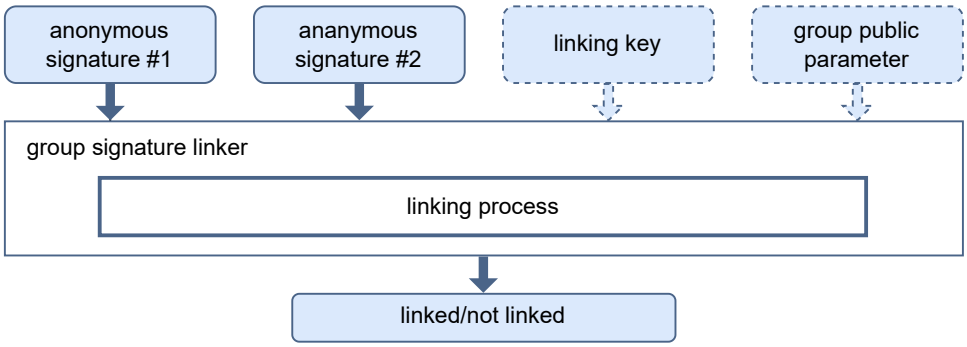

Figure 4.4. Group signature linking process [9]

Some mechanisms incorporate a *linking process*, described in Figure 4.4. They provide a way for a *group signature linker* to identify if two signatures have been produced by the same signer. Usually, the use of this property is optional and allows *user-controlled-linkability*, that consists in the choice by the signer to possibly generate a linkable signature or not [9].

Some mechanisms require the use of a *linking key* which is a private data element that permits the group signature linker to carry out the linking process.

The linking base (*bsn*), instead, represents the public element involved in the signature process, which, if used with the same value in two separate signatures, allows to link the two signatures.

4.2.3 Opening capabilities

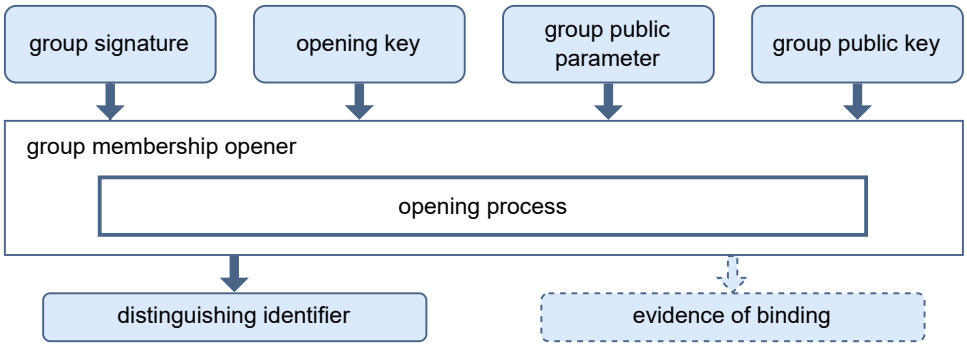

Figure 4.5. Group membership opening process [9]

The *opening process*, shown in Figure 4.5, is the process performed by the *group membership opener*, which allows to be able to trace the identity of the signer starting from the anonymous signature. Only a few mechanisms support it. Some

of them allow to provide, in addition to the identity of the signer, also a proof called *evidence of binding* [9]. This proof can be used by an *evidence evaluator* to make sure that the given signature belongs to the same identity that the opener has provided. There may be several reasons why it is useful to obtain evidence of binding. For example this becomes necessary if the result of the evaluation must be verified by a third party other than the opener.

4.2.4 Signature and verification process

As already mentioned, only group Members in possession of a signature key can create a signature. Unlike a conventional digital signature, in the *group signature process*, if the mechanism supports it, there can be an optional linking base that can possibly be delivered together with the signature, as it is a public parameter. As displayed in Figure 4.2, it is also interesting to note that the signature depends not only on the "group public parameter", but also on the "group public key" [9].

The verification is conceptually similar to the conventional signature: the "group public key" and the "group public parameters" are required to verify the validity of the signature.

4.2.5 Revocation mechanisms

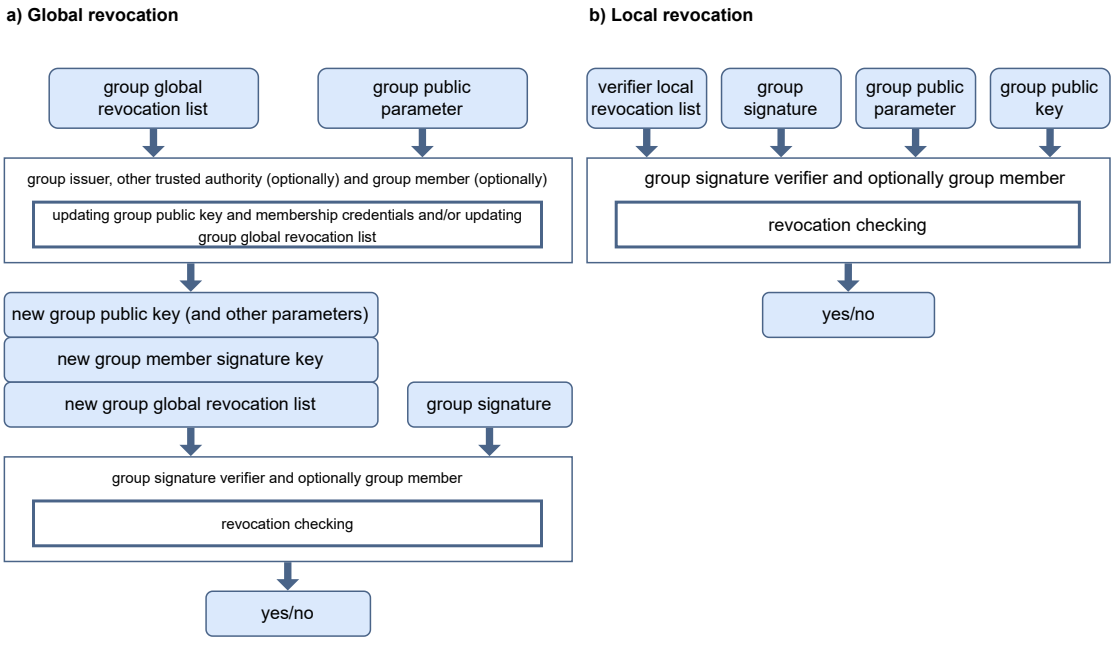

Figure 4.6. Group signature revocation processes [9]

An interesting topic about anonymous digital signatures is the ability to revoke a Member's right to create new signatures and/or notice that already created

signatures have been generated by revoked Members. The revocation mechanisms are dependent on the implementation mechanism, even if several characteristics are shared. An overview of the revocation process is proposed in Figure 4.6.

The standard defines 3 different levels of revocation for anonymous digital signatures using group public key [9]. Each level allows different types of authorizations to be revoked:

- **Level 1 revocation (Group revocation)**: The entire group is revoked. If the authorization of an entire group needs to be revoked, the appropriate group public key shall be added to a group public key revocation list. Any signature associated with a revoked group public key shall be rejected [9]. This revocation method is the same as the one used with a conventional digital signature scheme.
- **Level 2 revocation (Global Revocation)**: It consists of a global revocation of single *group membership credential*, so that signatures created with those credentials can be revalued as invalid by Verifiers [9]. Depending on the mechanisms, it can be achieved in 2 main ways:
  - *Credential Update*: it is an update of the credentials of each Member of the group and of the "group public key", carried out by the Issuer and possibly supported by the Members. It can be done cyclically at regular intervals or as soon as a malicious Member is detected [9].
  - *Group global revocation list*: when a malicious Member is detected, a trusted authority inserts data in the list. Each Verifier can judge the validity of the signature created by a potential malicious Member based on the data present in the list [9].
- **Level 3 revocation (Local Revocation)**: Level 3 revocation or *Verifier-local revocation* is the ability of a single Verifier to revoke a particular group membership credential [9]. This does not compromise the validity of the credential for all other Verifiers who will continue to consider the signature valid. The Verifier who wants to use a revocation of this type will make use of a personal *Verifier local revocation list* in which he will enter the data obtained from the signatures of the Member to be revoked as needed.

4.2.6 Types of revocation list

Depending on the type of revocation list, this enters the verification or signature processes in different ways. For example, some revocation mechanisms require the signer to prove at the time of signing that he is authorized to create the signature based on the revocation list present at the time of signing. Other revocation methods require, instead, that the Verifier consults the list during the *signature verification process* [9].

It is also possible that the list is "compressed" into a single parameter to increase the efficiency of the verification but without this parameter revealing

sensitive information regarding the privacy of the signer [11]. A mechanism that compresses a revocation list to a single parameter is also known as an *accumulator*.

The contents of the revocation lists can be of the following main types:

- In **private key revocation**, the private signature key of the revoked Member is inserted into the list [9]. In this way, a Verifier is able to understand if the signature to verify was created by a revoked Member, usually by repeating the signature operation or part of it for each revoked key. It is worth remembering that the Issuer or another trusted authority has access to the private signature key at the time of issuing the credential only in some specific situations. Such a list can be used both in global and local revocation.

- In **membership credential revocation**, the group membership credential of the revoked Member is added to the revocation list [9]. The signer might be required to provide proof that its credential is not on the list. Depending on the mechanism, such a list can be used in global revocation.

- In **verifier blacklist revocation**, the signature (or part of it) corresponding to a particular "signature linking base" is included in the list [9]. The Verifier can check, thanks to the linking capability, that the signature belongs to a signer whose signature is already present in the revocation list. Such a list can be used in local revocation.

- In **signature revocation**, a signature (or a partial signature) is included in the revocation list [9]. A Verifier, thanks to additional evidence provided by the signer or an evidence provided by the group membership opener, can verify that the signature belongs to a signer who has created a signature in the list. Depending on the mechanism, such a list can be used in either global or local revocation.

4.2.7 Mechanisms proprieties summary

To conclude, let us briefly summarize the most distinctive properties of the various mechanisms defined in the standard. For this purpose, we consider two fundamental aspects of the various implementations. Firstly the type of revocation mechanism that can be used (Figure 4.7), secondly the mathematical assumptions underlying them (Figure 4.8) [9]. It is noteworthy that the **mechanism 4**, which we will use in the implementation of this thesis, supports potentially all types of revocation seen above. Moreover, as we will see, the implementation acts as an *Static Diffie Hellman* oracle that weakens its security strength.

	Private key revocation	Verifier blacklist revocation	Signature revocation	Credential update
Mechanism 1	✓	✓		
Mechanism 2	✓	✓		
Mechanism 3	✓	✓	✓	
Mechanism 4	✓	✓	✓	✓
Mechanism 5				✓
Mechanism 6				✓
Mechanism 7				✓

Figure 4.7. Mechanisms: possible type of revocation [10]

	Strong RSA assumption	Decisional Diffie-Hellman assumption	Strong Diffie-Hellman assumption	Lysyanskaya-Rivest-Sahai-Wolf (LRSW) assumption	Static Diffie-Hellman
Mechanism 1	✓	✓			
Mechanism 2	✓	✓			✓
Mechanism 3		✓	✓		
Mechanism 4		✓		✓	✓
Mechanism 5	✓	✓			
Mechanism 6		✓	✓		
Mechanism 7		✓	✓		

Figure 4.8. Mechanisms: mathematical assumptions [10]

Chapter 5

Direct Anonymous Attestation

The *Direct Anonymous Attestation* (DAA) is a set of anonymous digital signature schemes performed using TPM. The generation of an anonymous signature with DAA is possible thanks to the collaboration between the TPM and the host device (Host). Due to the limited resources that the TPM possesses, the TPM must be used only for necessary operations, assuming the role of *principal signer*. By contrast, the Host, with a much higher computing capacity, as *assistant signer*, is responsible for the remaining calculations. The TPM and the Host together form the so called Platform. Such division of the signer role between TPM and Host enhances the complexity of the anonymous digital signature schemes as it is necessary to consider the Host as potentially untrusted [12]. As we will see, although the Host knows the credential, it will not be able to autonomously emit signatures.

5.1 Schemes and TPM specification

The first DAA scheme, RSA based, was proposed by Brickell, Camenisch, and Chen in 2004 and later standardized in the TPM1.2 specification [13]. Brickell, Chen, and Li proposed the first DAA scheme based on symmetric Pairings on Elliptic Curve (ECC-DAA) [1] which inspired the TPM 2.0 specification.

The specification provides a series of generic APIs that allow to perform a subset of the DAA-related operations which were thought to be necessarily performed by the TPM. According to the TPM specification writers, these APIs should be generic enough to allow new DAA schemes in the future without changing the specification. Indeed, over the years there have been various attempts to achieve DAA in different and increasingly efficient way [12] [14] [15]. However, many of the proposals turned out to be fallacious and required new developments [14].

Whether you want to attest an internal TPM object or sign a generic input, the specification provides a number of specific APIs for each purpose. In any case, any signature using DAA needs to perform a preliminary operation called *TPM2_Commit* that computes and returns some parameters necessary for anonymous signing. The TPM2_Commit accepts as input 2 points, then perform

the point multiplications on the provided points with the DAA secret key and a random key r_{cv}, then return intermediate signing values and a counter value cv. Through the counter value, the TPM, in the next stage of signing, will be able to derive r_{cv} which is necessary for its computation.

The signing of a generic input is done with the *TPM2_Sign*. In order for the TPM to validate the signature input, the specification forces to use as input a hash necessarily produced by the TPM itself. This mechanism is to prevent malicious input as discussed in subsection 2.4.4. For this reason it is necessary to call *TPM2_Hash* which will provide a ticket *tkt* to be presented later at the signing stage as proof that the hash being given as input has been generated and validated by the TPM. Once the ticket has been obtained from the TPM2_Hash and the counter from the TPM2_Commit, the call to the TPM2_Sign is made to complete the anonymous digital signature.

For the attestation of PCR registers, in a manner quite similar to non-anonymous attestation, *TPM2_Quote* is used. The API accepts as input the counter value of the TPM2_Commit and the set of PCR registers to be attested and returns the attestation.

It is also possible to certify a generic object (for example, a key) in the TPM, that is, to issue an attestation certifying that the object is present in the TPM and has specific public data. For this use case, *TPM2_Certify* is used, which accepts as input the references to the object to be attested and the counter value and returns the desired attestation.

5.1.1 Applications

There are many application that increasingly require the need to preserve user privacy. Particularly important is the topic of the Internet of Things, where very simple devices have low security capabilities. To increase their security and test their operating status, they could be sustained by a TPM and use remote attestation. Furthermore, applications such as vehicle-to-vehicle communication should prevent the possible tracking of users, at the same time guaranteeing the authenticity of the communication [16]. DAA constitute an adequate solution to the lack of user privacy characterizing traditional methods.

A DAA protocol is already integrated into the FIDO authentication framework [17], in which the TPM creates a signature using DAA for authentication purpose. The attempt to use the DAA in Cloud Services [18] is also very interesting to allow the user to choose which services can use its data and prevent disastrous data leaks, where the connection between the user's identity and his actions would seriously damage its privacy.

In addition, thanks to the ability to perform PCR registers attestation via TPM2_Quote, DAA can be used to perform remote attestation. DAA allows to certify the state of a system while preserving the user's privacy. First of all, it allows system attestations without the need to know the identity of the system and therefore of the user. Secondly, it minimizes the amount of information that

a malicious user would possess if he was able to access information regarding the claims (*data minimization principle*) [15].

5.1.2 Anonymity and unlinkability

As mentioned in subsection 4.2.2, the presence of a linking base (*bsn*) in the signature creation phase gives the possibility to link two signatures. If two signatures performed by the same signer use the same *bsn*, then it will be possible for a Verifier to find out and link them to the same author. Otherwise, in case it is not possible to link the two signatures made by the same signer, we say we have the property of *unlinkability*.

On other hand, the concept of *anonymity* can be defined as the impossibility for any actor, including the Issuer, to find the identity of the platform given a set of signatures.

We can say for the purposes of this paper that the *privacy* of the Platform is preserved when the proprieties of anonymity and unlinkability are simultaneously respected.

If a signer chooses to create a linkable anonymous digital signature using the same *bsn* several times, it would leave a trace that would convince a Verifier that the signatures were all created by the same entity, but would still not allow it to trace its identity. In this case we have the so called *pseudo-anonymity* or linkable anonymity.

The ability to relax the privacy property of a Platform and to use pseudo-anonymity allows a system with anonymous actors to create a classification of information not relating to an actor identity, but on its *pseudonym*. It therefore allows to have a history of the signing activity carried out by a Platform, while respecting its anonymity. It could also be possible to revoke the possibility of a Platform to issue other valid signatures by identifying it by its pseudonym.

If the Platform wants to generate a new pseudonym, it could start to create signatures with a new *bsn* and then it will be associated with the new pseudonym. It is responsibility of the service to prevent unacceptable behavior from the Platform, possibly rejecting signatures that are outside a well-constructed whitelist.

If an application needs to make sure that a certain Member of the group of possible signers can have only one pseudonym, then it could force all users to use the same *bsn* in all signatures. In this way, in fact, each signature produced by users is accepted only if the chosen *bsn* is present. As consequence, the requirement to use it results in the possibility, starting from a signature, to revoke Member ability to perform new signatures.

It is clear that the possibility of having adaptable anonymity and adaptable unlinkability allows, at the expenses of privacy, to obtain simpler systems and to extend possible applications. The more privacy guarantees we give to a system, the more difficult it will be to guarantee its security. The more the privacy properties are relaxed, the more it is possible to control the system with ease and efficiency.

This is the manifestation of the tradeoff between security and privacy in which the DAA offers new tools in favor of privacy.

5.1.3 Remote attestation: full and hybrid DAA

In the case a Platform is willing to execute a remote attestation using exclusively the DAA, it should first create a DAA key and obtain a credential from the Issuer. Once having obtained the credential, it is able to perform a Quote attestation through the DAA key and the received credential. Therefore, the Verifier validate the attestation signature by only knowing the public parameters of the signature group which the Platform is part of.

For simplicity we call this way of carrying out the remote attestation *full-DAA*, since all the various phases, including the Quote, are carried out with the use of operations related to the DAA.

However, carrying out a full-DAA remote attestation brings with it some disadvantages compared to the traditional one. First of all, the fact that it is a relatively recent technology with few ready implementations, also given the great variability of the existing schemes. Secondly, the DAA adds complexity and requires a higher workload for the TPM, which leads to longer signatures execution times. In contexts where the same TPM is shared among several actors who carry out the remote attestations, this could be a not negligible problem. Recently, new schemes have been created to narrow the gap with traditional signatures by increasing the performance of the DAA [15].

As discussed in subsection 5.1.2, it is possible to sacrifice unlinkability to obtain guarantees on the Host's pseudonym. This allows to create a hybrid protocol where Quotes are executed by means of an Attestation Key (AK) and a traditional attestation. After obtaining the credential from the Issuer, it is used to certify a traditional AK with DAA and exchange it with the Verifier. The latter will be able to verify that the AK has been signed by a member of the group. With a previous agreement between the parties on the use of a known, fixed, *bsn* and with the use of this *bsn* in the AK certification phase, it is ensured that in the future the Member cannot try to certify a new AK without being identified thanks to the linking properties. A correlation is then created between the Host's pseudonym and the AK that the Host will use to sign the Quote attestations.

Thanks to this *hybrid version* of the protocol, it is possible to obtain the privacy advantages guaranteed by the DAA used in the key certification and the advantages of traditional attestation methods. On the other hand, this solution can only be used where the pseudo-anonymity of the Platform is sufficient as each attestation would become linkable. The implementation proposed in this thesis uses this hybrid version of remote attestation.

5.2 Wesemeyer scheme

Let us now discuss the implementation details of the scheme proposed by Wesemeyer et al. [2] that will be used in the implementation proposed in this thesis. We have chosen this scheme as it presents a complete description of the protocol, including the TPM calls, and a C++ implementation of the various parts of the protocol that we have used as basis for this thesis project. Moreover, a formal analysis of the security of the protocol useful for future works is provided. The scheme is based on Mechanism 4 of ISO 20008-2 [10] from which it inherits the linking capabilities and the revocation mechanisms.

In this chapter we first define the used notation, and afterwards we will present the prerequisites for the execution of the protocol, i.e. the definition of the public and private parameters of the group. We will then proceed with the description of the *make credential* procedure, that is a procedure defined in the TPM2 specification fundamental for the exchange of a secret starting from the TPM Endorsement Key (EK). We will also describe the *join protocol* between the TPM, the Host and the Issuer that will allow the Platform to obtain a DAA credential. We will next analyze the various signature types and Host-TPM protocol phases in order to create attestations of the PCR registers (Quote), of specific object of the TPM (Certify) or generate signatures on generic inputs (message sign). We will then describe the verification mechanism of the various types of signatures. Lastly, we will make some considerations concerning the security of the scheme showing two criticalities nowadays present in its usage. The first is due to the elliptic curves that are currently used by the TPM, while the second one is due to the fact that, to date, the TPM behaves as a static Diffie-Hellman oracle.

The notation used in the following discussion is given below [2]:

t	A security parameter.
p, n	Prime numbers.
$[x, y]$	The set of integers from x to y inclusive, if x, y are integers satisfying $x \leq y$.
Z_p	The set of integers modulo p, i.e., $[0, p-1]$. These form a prime field, F_p. Also note that F_{p^m} is an extension finite field with p^m elements, where m is a positive integer.
Z_p*	The multiplicative group of invertible elements in Z_p , i.e. the set of integers in $[1, p-1]$.
G_1	An additive cyclic group of order n over an elliptic curve. The curve has points with co-ordinates in $F_p \times F_p$.
P_1	A generator of G_1.
$G2$	An additive cyclic group of order n over an elliptic curve. The curve has points with co-ordinates in $F_{p^2} \times F_{p^2}$.
P_2	A generator of G_2.
$[k]P$	Multiplication operation that takes a positive integer k and a point P on the elliptic curve E as input and produces as output another point Q on the curve E, where $Q = [k]P = P + P + ... + P$, i.e., the sum of k copies of P.

G_T	A multiplicative cyclic group of order n.
$\hat{h}$	A bilinear map $\hat{h} : G_1 \times G_2 \rightarrow G_T$ such that for all $P \in G_1, Q \in G_2$, and all positive integers a, b, the equation $\hat{h}([a]P, [b]Q) = \hat{h}(P, Q)^{ab}$ holds. This bilinear map is also called a pairing function.
H	A cryptographic hash-function.
H_p	$\{0,1\}^* \rightarrow Z_p$ is used when we want to hash to a co-ordinate on the elliptic curve.
H_n	$\{0,1\}^* \rightarrow Z_n$ is used when we want to hash to a multiplier for an elliptic curve point.
H_k	$\{0,1\}^* \rightarrow Z_{2^k}$ a general hashing function.
H_s	The "map to point" function, used to map a random string, rs, to a tuple $H_s(rs) = (s_2, y_2)$ such that $(H_p(s_2), y_2)$ is a point on the curve G_1. The function H_s is essentially the function I2P given in ISO/IEC standard document 11770-4 [19] using SHA256 as the Key Derivation Function (KDF).
$\perp$	The item is unspecified. To avoid unnecessary if–then–else constructs in the diagrams, this carries through. So, for group items, for example, if $A = \perp$, then $[b]A$ will also equal $\perp$.
$len16(x)$	The length of x expressed as a 16-bit integer (most significant bit first).
senc	A symmetric-key encryption function, $senc(data, key)$.
sdec	A symmetric-key decryption function, $sdec(cipher, key)$.
aenc	An asymmetric encryption function, $aenc(data, ekey)$.
adec	An asymmetric decryption function, $adec(cipher, dkey)$.
m	A message to be signed.

To use the scheme it is first necessary to establish which are the cryptographic parameters that will then be used by all the actors involved [2]. In particular:

- a security parameter t;
- an asymmetric bilinear group pair (G_1, G_2) of large prime numbers order n and a pairing function $\hat{h}$;
- two generators, P_1 and P_2.

At the time of group creation, the Issuer:

- chooses two integers x, y in Z_n which are the Issuer's private keys;
- chooses $X = [x]P_2$ and $Y = [y]P_2$;
- publishes *group public parameters* $= (G_1, G_2, G_T, \hat{h}, P_1, P_2, H_n, H_p, H_k)$;
- publishes *group public key* = (X, Y).

5.2.1 Make credential procedure

The *make credential procedure* is a procedure described in the TPM2 specification [3]. This allows an external actor that has the EK of the TPM and the public data of one of the keys created by that TPM to be able to create a message containing a secret that is readable only by the TPM to which the two keys belong [2]. In our case, given:

- ε, public data of the TPM RSA Endorsement Key,
- Q_{PD}, public data for the TPM DAA key,
- C the DAA attestation key credential,
- a random credential key K which is used to encrypt C.

We want to make sure that K can be used by a certain Host only if:

- ε comes from the Host's TPM,
- the DAA key was generated by the same TPM.

The make credential procedure is used twice in the join phase. The first time without the attestation key credential in input, as the intention is to verify that the Host has access to the TPM and therefore is able to unwrap the credential blob created. The second time, on the other hand, is complete with the signed credential with which the Host can ensure that it comes from the Issuer [2]. We proceed as follows:

- validate that ε belongs to a legitimate TPM and has the characteristics we expect. In our case ε must conform to the TCG endorsement [3] key profile, i.e. it is a 2048-bit RSA key, uses AES 128 as the symmetric encryption algorithm and SHA256 as hash algorithm;
- ensure that the properties of the DAA key is conform to what is expected. That is, the key must be *restricted*, *fixedTPM*, *fixedParent* and use SHA256 as hash algorithm. The *DAA key name* is calculated by deriving it from the public data of the key [3]:

  $$Q_N = nameAlgID_{16} || H_{name}(Q_{PD})$$

  $nameAlgID_{16}$ is the 16-bit identifier which represents the hash algorithm of the key, for the SHA256:

  $$H_{name}(Q_{PD}) = 0x000b$$

- generate a random seed s of t bits;

- generate an encryption key k_e and a HMAC key k_h through a key derivation function:

$$k_e = KDF(s, \text{“}STORAGE\text{”}, Q_N)$$

$$k_h = KDF(s, \text{“}INTEGRITY\text{”}, NULL)$$

- encrypt K using k_e via AES 128 in CFB mode and IV to zero:

$$\hat{K} = senc(K, k_e)$$

- generate the HMAC, H, of $\hat{K}$ using k_h:

$$H = hmac(k_h, len_{16}(\hat{K})||\hat{K}||Q_N)$$

- generate the *credential blob* as:

$$CB = H||len_{16}(\hat{K})||\hat{K}$$

- encrypt the seed s with ε to get $\hat{s}$, which is what in the TPM specification is called the secret. The encryption uses RSA-OAEP encryption with SHA256 as the hash function and MGF1 padding:

$$\hat{s} = aenc(s, \varepsilon)$$

- encrypt C using K:

$$\hat{C} = senc(C, K)$$

- output $\hat{C}$, CB and $\hat{s}$.

To derive the credential, the Host uses *TPM2_ActivateCredential* to unwrap the credential blob and retrieve the key K. At this point he can use K to decrypt the credential C.

5.2.2 Join

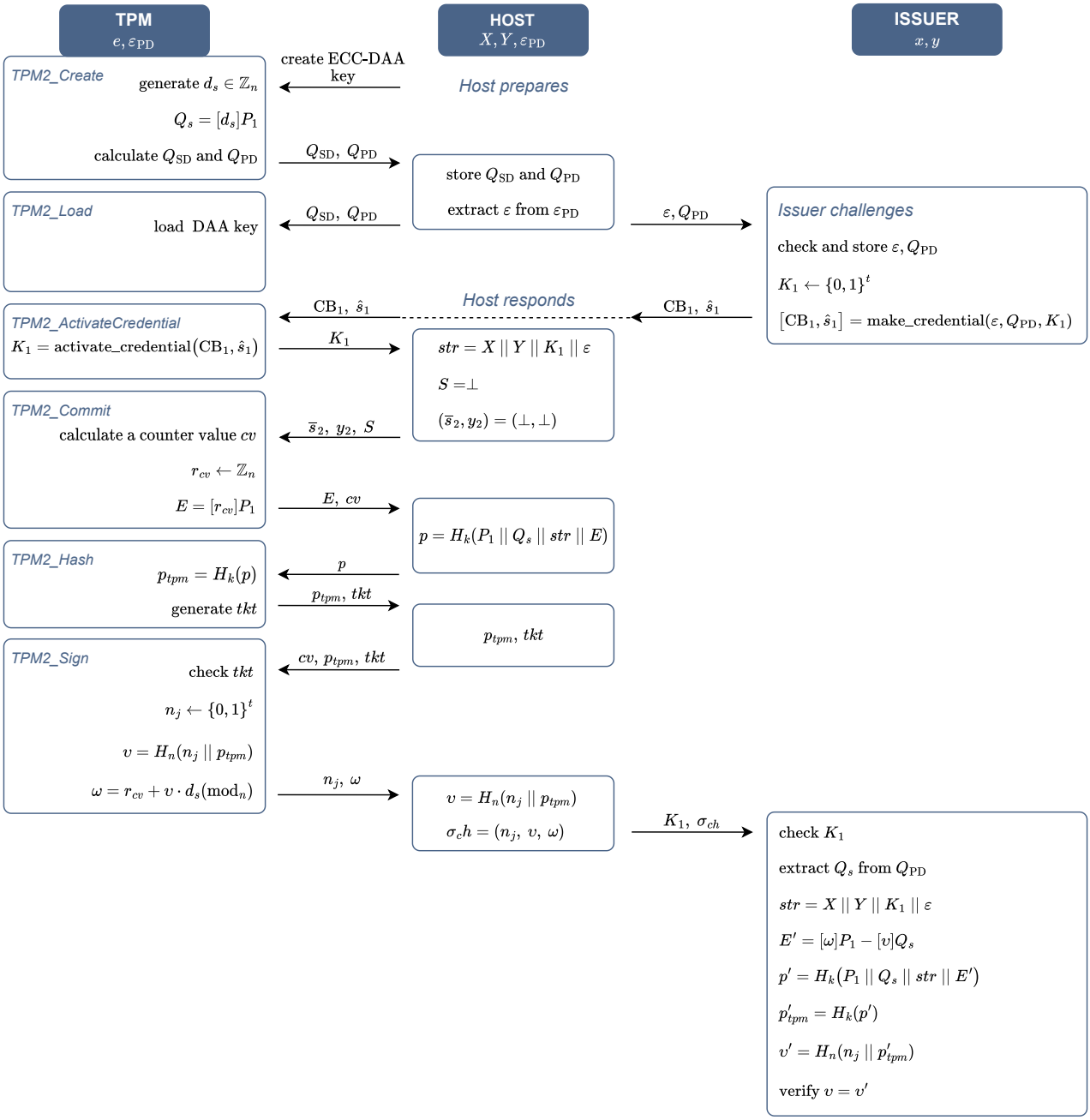

Figure 5.1. Initialing the join [2]

When creating a new membership credential, the candidate Member and Issuer perform the following operations as shown in Figure 5.1 [2]:

- the TPM creates the DAA key and the relative public key Qs. The Host receives both the public data of the Q_{PD} key and the secret Q_{SD} data. The Host then sends Q_{PD} and the ε public endorsement key to the Issuer;

NOTE: the key must be *fixedTPM*, *fixedParent* and *restricted* to allow attestations of internal objects of TPM.

- the Issuer creates the K_1 challenge and encapsulates it with ε through the *make_credential* procedure and then sends the result to the Host;
- the Host and the TPM through the *TPM2_ActivateCredential* extract the challenge and obtain K_1;
- $Str = X||T||K_1||\varepsilon$ is built. It will be used later in the hash. The presence of ε is very important due to the fact that it protects against attacks from possible malicious TPM [20];
- the Host calls the *TPM2_Commit* with S and $(s2, y2)$ unvalorized;
- the data is merged and passed to *TPM2_Hash* which, in addition to the hash, produces a *tkt* ticket used to confirm that the hash is generated by the TPM;
- the *TPM2_Sign* is called where the TPM checks *tkt* and eventually generates the signature;
- the signature is sent to the Issuer together with the value of the K_1 challenge; The Issuer verifies the validity of the challenge and the signature.

At this point of the join, the Issuer is convinced that the DAA key and the EK ε belong to the same TPM [2].

The second phase of the join begins, here the Issuer creates the attestation key credential (A, B, C, D) for the specific DAA key, sends it encapsulated to the platform, which, in turn, can check its validity and acquire it as shown in Figure 5.2 [2]:

- the Issuer generates the attestation key credential $cre := (A, B, C, D)$;
- the Issuer signs the *cre* credential and generates σ_{cre}. It is basically a double *Schnorr* signature that allows the Host to verify that D and B are correctly formed, that is, that B and P_1 have the same discrete logarithm of D and Q_s;
- the Issuer creates the K_2 encryption key with which it encrypts the attestation key credential, then encapsulates the cypher-text with the *make_credential* procedure and the result sent to the Host;
- the Host uses the *TPM2_ActivateCredential* to obtain the K_2 encryption key. It then uses the key to decrypt the C certificate and thus obtain the attestation key credential;
- the Host checks the signature applied by the Issuer;
- the Host confirms that the attestation key credential is valid using the bilinear map h.

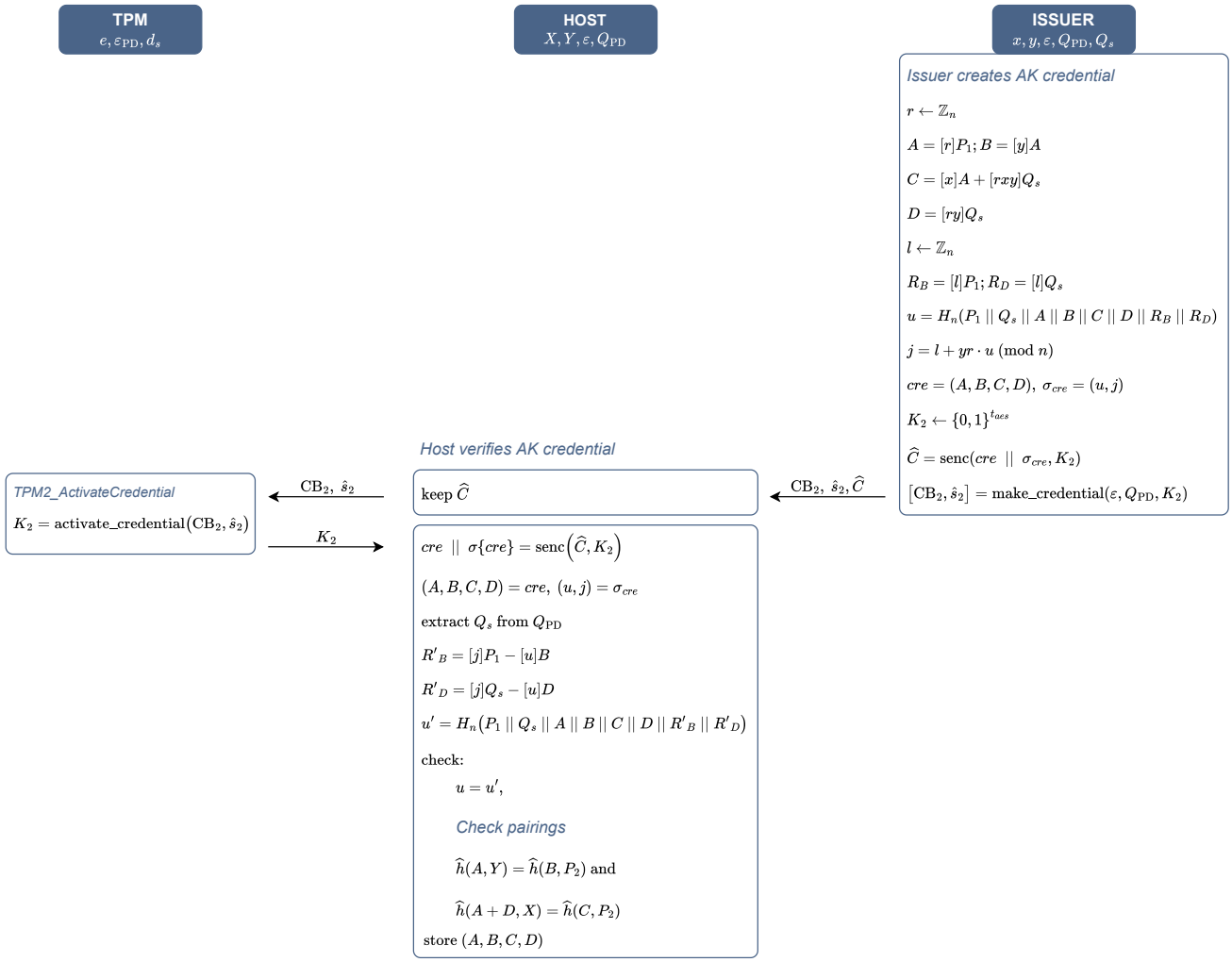

Figure 5.2. Completing the join [2]

5.2.3 Sign

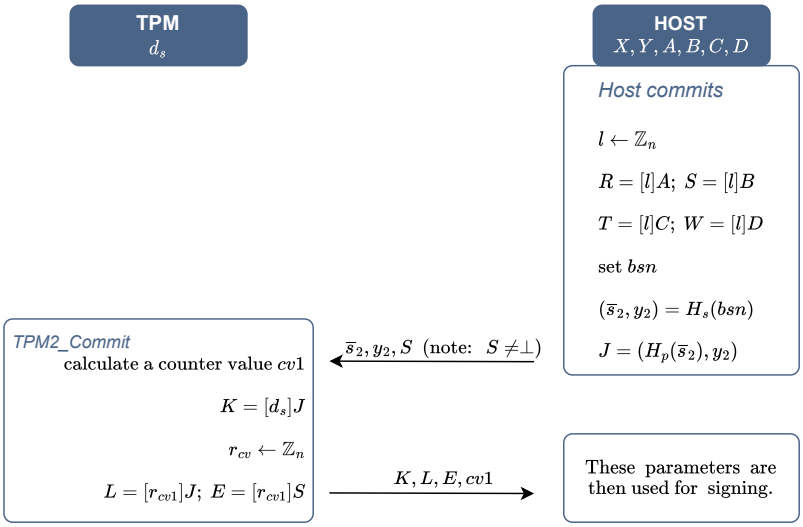

Figure 5.3. Preparing to use the DAA key [2]

Signing with an DAA key is a 2-step process [2]. In the first phase, as shown in Figure 5.3, the key credential certification obtained in the *join phase* by the Issuer is randomized, thus creating (R, S, T, W).

The *TPM2_Commit* is then carried out in preparation for the next signature phase. It is possible to create the signature with a *bsn* set or not. The *bsn* will make the signature linkable, but will require additional operations by the TPM, which will lengthen the execution times of the signature. In particular, if the *bsn* is not used, the *TPM2_Commit* will return only $E = [r_{cv1}]S$, while K L J will remain undefined and will not contribute to the hash H_k, which will be calculated in the second phase.

In the second phase, the DAA key will be used for the actual signature. There are 3 types of signature defined:

- the *Message Sign*, i.e. a signature of a given message passed as input;
- the *Certify*, i.e. an attestation of a TPM key by signing its public data;
- the *Quote*, i.e. an attestation of the public data of the PCR registers including of their values.

Each type has its own way of proceeding in generating the signature, as detailed below.

Message signature

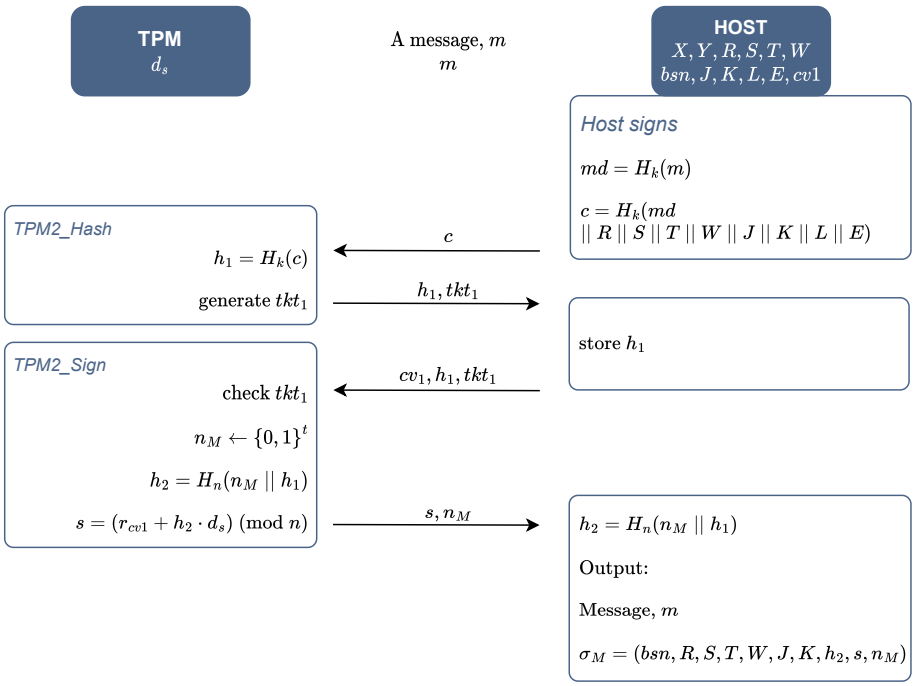

Figure 5.4. Signing a message m [2]

If the Host want to sign a generic message, as shown in Figure 5.4, it has to [2]:

- calculate $md = H_k(m)$;
- compute $c = H_k(md||R||S||T||W||J||K||L||E)$ with K, L and J used only if bns is present;
- call the *TPM2_Hash* passing as input c and obtain, together with the hash, the ticket tkt_1;
- call the *TPM2_Sign* passing the hash and the tkt_1 ticket;
- compute $h2 = H_n(n_M||h_1)$;
- return the signature $\sigma_M = (bsn, R, S, T, W, J, K, h_2, s, n_M)$.

Key certify

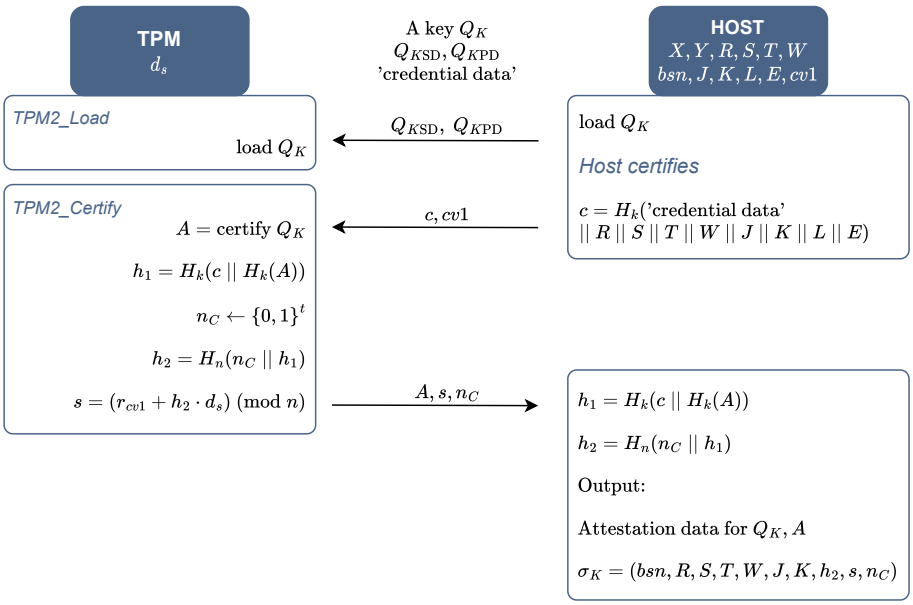

Figure 5.5. Certifying a key Q_K [2]

In case the Host wants to certify a TPM key, then, as shown in Figure 5.5, it will [2]:

- load the key to be certified into the TPM via *TPM2_Load*;
- compute $c = H_k(\text{“credential data”}||R||S||T||W||J||K||L||E)$ with K, L and J used only if *bns* is present;
- call *TPM2_Certify*;
- compute $h_1 = H_k(c||H_k(A))$ and $h_2 = H_n(n_C||h_1)$;
- return the attestation $\sigma_K = (bsn, R, S, T, W, J, K, h_2, s, n_C)$.

Quote

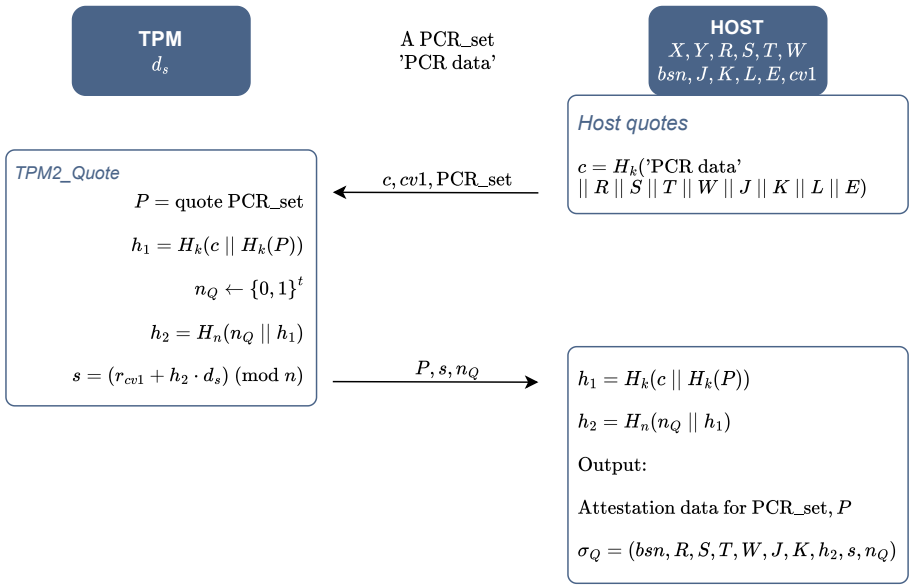

Figure 5.6. Quote a set of PCR values [2]

If, on the other hand, a Quote is needed, as shown in Figure 5.6, the Host has to [2]:

- compute $c = H_k(\text{“pcr data”}||R||S||T||W||J||K||L||E)$ with K, L and J used only if bns is present;
- call the *TPM2_Quote*;
- compute $h_1 = H_k(c||H_k(P))$ and $h_2 = H_n(n_C||h_1)$;
- return the attestation $\sigma_Q = (bsn, R, S, T, W, J, K, h_2, s, n_Q)$.

5.2.4 Verify

Verification of a DAA signature also consists of two stages. The first stage is independent of the signature type and concerns verification of the credential of the attestation key. The second stage depends on the type of signature and relates to the verification of the actual signature [2].

Verify of DAA credential

For verification of the DAA credential, the Verifier:

- verifies $\hat{h}(R, Y) = \hat{h}(S, P_2)$ and that $\hat{h}(R + W, X) = \hat{h}(T, P_2)$;

- computes $E' = [s]S - [h_2]W$.

If the *bsn* is set, in addition:

- computes $(\overline{s}_2, y_2) = H_s(bsn)$;
- computes $J' = (H_p(\overline{s}_2), y_2)$;
- verifies that $J = J'$;
- computes $L' = [s]J - [h_2]K$.

Verify a message signature

Given the message signature $\sigma_M = (bsn, R, S, T, W, J, K, h_2, s, n_M)$ and message m, the Verifier:

- computes $md' = H_k(m)$;
- computes $c' = H_k(md'||R||S||T||W||J||K||L'||E')$ with K, L' and J used only if *bns* is present;
- computes $h'_1 = H_k(c')$;
- computes $h'_2 = H_n(n_M||h'_1)$;
- checks that $h'_2 = h_2$.

Verify a key attestation

Given the attestation signature $\sigma_K = (bsn, R, S, T, W, J, K, h_2, s, n_C)$, the attestation A and the public data of the certificated key Q_{KPD}, the Verifier:

- extracts key name Q_N from Q_{KPD};
- checks Q_N corresponds to the given in A;
- computes $c' = H_k(\text{“credential data''}||R||S||T||W||J||K||L'||E')$ with K, L' and J used only if *bns* is present;
- computes $h'_1 = H_k(c'||H_k(A))$;
- computes $h'_2 = H_n(n_c||h'_1)$;
- verifies that $h'_2 = h_2$.

Verify a Quote attestation

Given $\sigma_Q = (bsn, R, S, T, W, J, K, P, h_2, s, n_Q)$ and the Quote attestation P, the Verifier:

- computes $c' = H_k(\text{“PCR data}''||R||S||T||W||J||K||L'||E')$;
- computes $h'_1 = H_k(c'||H_k(P))$;
- computes $h'_2 = H_n(n_c||h'_1)$;
- verifies that $h'_2 = h_2$.

5.2.5 Linkability

As reported in [10], it is possible to check if two signature are linked, that is, if they are created by the same signer, by verifying the equality of J and K in the two signatures.

If all signer sign with the same *bsn*, all signatures will have the same J, from which it will be possible to check that the user has complied with the obligation to use the specific *bsn*.

K on the other hand depends on both *bsn* and DAA key so it is dependent on the signer who generate the signature. If we had two signatures having the same K, then we could conclude that the signatures were issued by the same signer. In case the protocol requires always using a particular *bsn*, then K can be used to discern a revoked Member. In fact, checking that K is not in a list of revoked users would ensure under these conditions that the Member credential is not revoked.

5.2.6 Parameter selection and BN_P256 (in)security

As highlighted, the proposed implementation requires pairing friendly EC curves. To date, the TPM has only 2 standardized curves of this type: BN_P256 and BN_P638.

Recent developments [21] show that the BN_P256 curve is not as safe as expected and therefore it is necessary to use the BN_P638 but its implementation in the TPM is optional and therefore hardly implemented in physical TPMs. For this reason, the BN_P256 has been chosen until the BN_P638 become usable. The choice of curve specifications comes as a consequence as:

- $t = 256, G1, G2$ set by BN_P256;
- P_1 is set by the TPM specification;
- P_2 is set by the AMCL [22], a cryptographic library used for the implementation;

- H_k is fixed SHA256;
- H_p is the result of SHA256 $mod(p)$;
- H_n is the result of SHA256 $mod(n)$.

5.2.7 Static Diffie-Hellman Oracle

Like Acar et al. have shown [23], to date, the APIs present in TPM 2.0 can be used as Static Diffie-Hellman Oracle. With the introduction of the ECC-DAA and support for Schnorr signatures, the *TPM_Commit* API has been added to be used in the DAA as a preliminary step before signing. The operation takes an input h and returns $W := h^w$ where $w \in Z_p$ is chosen randomly by the TPM. The *TPM_Sign* accepts a c as input and returns $r := cx + w$, where x is the private key. It is therefore possible to calculate $h^x = (h^r/W)^{1/c}$. Therefore, this algorithm provides the possibility, given a choice h in input, to obtain h^x, that is, it behaves like a Static Diffie-Hellman Oracle. This weakens the security of a signature on a BN_P256 curve from 2^{128} to 2^{118}. Fortunately, due to the slow processing speed of the TPM and to the limits that can be imposed on the maximum number of signatures in a given time frame, the attack is difficult to make.

Chapter 6

Design

The goal of this thesis is the implementation of a remote attestation framework using DAA in a distributed context. We can imagine a context where the various attestation actors, i.e. Members, Issuers and Verifiers can also reside on different machines and communicate through REST calls. It should be possible that there is some kind of orchestrator that can generate distributed objects, which, in their first startup will perform an automatic configuration phase that will lead them to become Members through interaction with the Issuer who must be able to verify their eligibility. Verification of the identity of Members by the Issuer is beyond the scope of this thesis. In addition, it is assumed that the Member and Verifier have trusts with respect to the Issuer.

For this purpose we have chosen to build an infrastructure to enable Direct Anonymous Attestation based on the hybrid scheme previously outlined in subsection 5.1.3. In this way we can distinguish the initial configuration phase, in which each distributed object becomes Member and exchanges an AK with a Verifier, with the remote attestation phase, which stays equivalent to that of traditional remote attestation. To simplify management in a distributed environment, we chose to develop attestation in *push mode*, that is, where the attester (Member) periodically contacts a verification server (Verifier) providing a Quote and the BIOS and IMA event log. This way, the Verifier does not have to have the authorizations to contact each Member and for that reason may also be outside the infrastructure where Member reside.

Six modules have been created to carry out the various phases of a hybrid DAA as show in Figure 6.1:

- the **Issuer** is a webserver that Manages the creation of credentials to allow new members access to the group;
- the **Verifier** is a web server that allows the exchange of an AK with the Member and then handles push remote attestations by verifying and logging any problems related to signatures or attestations;
- the **Provision_tpm** is an application that manage the creation and persistence in the TPM of the Endorsement Key (EK);

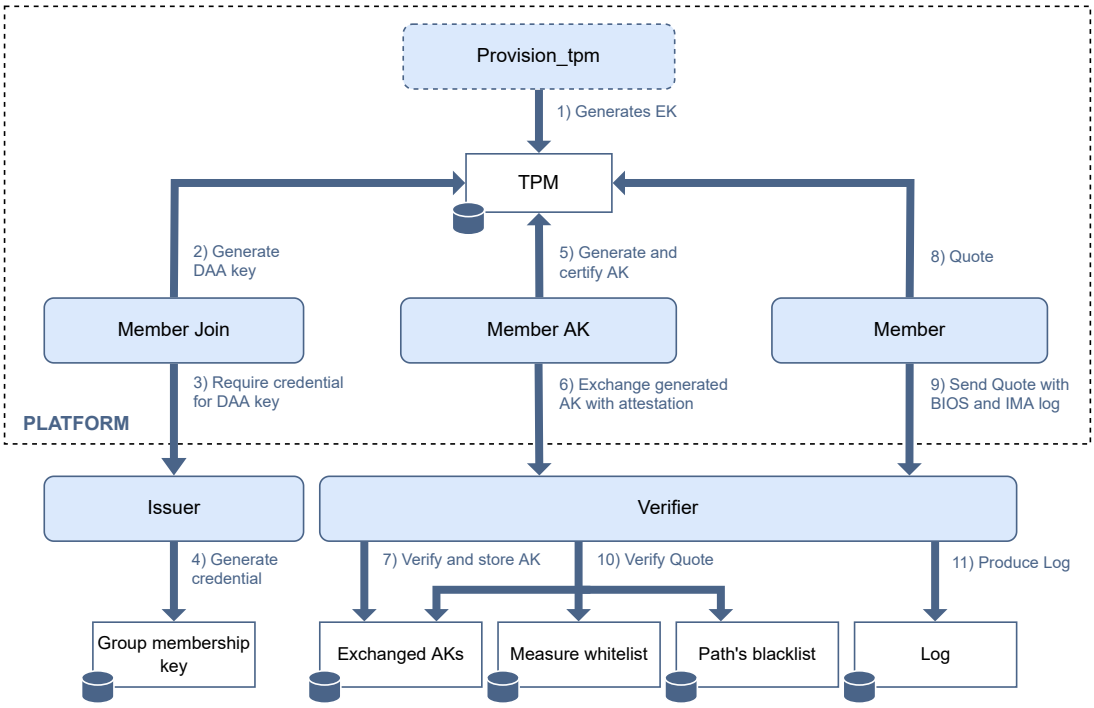

Figure 6.1. Implementation overview

- the **Member_join** creates the DAA key, executes the join with the Issuer and obtains and persists the generated credential;

- the **Member_ak** is responsible for creating an Attestation Key (AK), certifying it using the DAA, and, finally, exchange the key with a Verifier;

- the **Member** uses the AK for the purpose of traditional remote attestation with the Verifier.

With reference to the Figure 6.1, the Platform configures its TPM, creating the EK (1) (Provision_tpm). It then generates the DAA key (2) and requests a credential from the Issuer (3-4) (Member_join). The possession of such credential renders it a Member of the group, allowing to generate (5), exchange (6) and register (7) the AK with the Verifier (Member_ak). At this point, it is ready to iterate the execution of the remote attestation with the Verifier in push mode making a Quote (8) and than sending that with BIOS and IMA log (9) (Member). Finally, the Verifier validates the AK, signature, and attestations (10) and produces an attestation log (11) containing any validation problems.

Let us now describe the applications and their interactions in more detail, focusing first on how they work and then describing the protocols.

6.1 Applications

Issuer

The Issuer application acts as the Issuer from DAA scheme by handling group secret keys and the possibility to admit new members into the signature group through credentials release. It deals with the verification of the candidate identity and ensuring the compliance of an authorization policy for the purpose of credential release. As previously mentioned, the Issuer is a HTTP webserver waiting for requests by candidates Member willing to join the group through the "*Join protocol*". The user manual can be found in section A.1.

Provision_tpm

Provision_tpm has the aim of setting all TPM configurations so that the TPM is then usable by the following applications. Its main function is to generate the EK which will be used as parent key during the DAA generation and to force its persistence in the TPM.

This is a non-fundamental step, as it could already exist an EK which is persistent in the TPM for various reasons. The generated EK uses handle=0x810100c0 as well as all the following modules. To date, the only way to change the handle value is directly modifying the source code and recompiling the application. Therefore, it is necessary to possibly launch it at most once for each machine to be attested. The user manual can be found in section A.2.

Member_join

The Member_join creates or reads from a file a DAA key and executes "Join protocol" together with the Issuer to obtain a DAA credential which will be stored in an output file. The user manual can be found in section A.3.

Verifier

The Verifier has a double role in this hybrid version of DAA scheme. It allows Members to register an AK through a key certification via DAA and, meanwhile, it manages the verification of the measurements regularly sent by the Member. For the AK registration, the Verifier executes with the Member the "*AK exchange protocol*", receiving an AK certificate from the Member, forcing, during the signature phase, the use of a fix *bsn* chosen by the Verifier. The Verifier then verifies that the DAA signature is valid and it also ensures that the Member has never before exchanged an AK with it by guaranteeing that it can exchange at most a single AK with each Member. To do this, as discussed in subsection 5.1.2 and subsection 5.2.5, it exploits linking capabilities of the DAA that, through the use

of the fixed *bsn*, will produce a fingerprint in the signature by which we can distinguish whether that Member has already made an attempt to exchange the AK or not.

In order to verify the measurements, the Verifier receives from the Member through the "*Push measurement protocol*" the attestation of the TPM and of the state of the PCR registers as well as the logs of BIOS and IMA events. It then checks the signature of the attestation with the AK and verifies that the declared state of the PCRs matches the one obtained from the received BIOS and IMA events. It also checks that each event is expected, i.e. present in a list of expected measurements. It thus produces a measurement log in which any problems in validations or any unexpected BIOS and IMA measurements are persisted. The user manual can be found in section A.4.

Member_ak

The Member_ak has the aim to generate an AK in the TPM and register it to a specific Verifier through the "*AK exchange protocol*". To do that, it needs both the DAA credential and the DAA key, with which it will perform the certification of the AK. If successful, the data of the generated AK will be persisted into a specific file. The user manual can be found in section A.5.

Member

The Member is an application meant to be regularly called to perform a traditional remote attestation with the Verifier through the "*Push measurement protocol*". The Member needs an AK previously exchanged with the Verifier with which it intends performing the attestation.

The Member asks TPM to make a Quote of the configured PCR registers and afterwards reads the BIOS and IMA measurements thus composing the payload to forward to the Verifier. The user manual can be found in section A.6.

6.2 Protocols

Join protocol

The aim of the Join protocol is to agree the Issuer and the candidate Member on a DAA credential.

The candidate first requests the public data of the group. He then sends his EK and DAA key to the Issuer so that the Issuer can verify his identity via the EK. In case the Issuer reputes that the candidate can join the group, it creates a challenge to the candidate, via the make credential procedure, with the purpose of proving that he is the holder of the EK and DAA key received. In case the candidate succeeds in solving the challenge through the activate credential procedure, he

will ask the Issuer for the issuance of a credential for his DAA key through the use of the obtained credential key. The Issuer then verifies the validity of the response to the challenge and if it is valid, produces an encrypted DAA credential with a new credential key that can be derived from a new challenge that it will pose to the candidate. Finally, the candidate extracts the new credential key and decrypts the DAA credential. Implementation details can be found in section B.1.

AK exchange protocol

The exchange of the AK between the Member and the Verifier is handled through the AK exchange protocol.

The Member after creating an AK makes a DAA attestation on the key with a static *bsn* derived from a previous interaction with the Verifier. In this way he can prove that he is a Member of the signing group, and through the fixed *bsn* he can prove to the Verifier that he has never before attempted to exchange an AK with him. In fact, the Verifier at the end of the AK exchange, stores the Member's signature point K (PT_K) so that it can detect future uses of the same DAA key for key certification. Implementation details can be found in section B.2.

Push measurement protocol

Through the Push measurement protocol, a Member performs a remote attestation with the Verifier.

The Member request a Quote to the TPM signed with the AK first exchanged with the Verifier obtaining an attestation of the PCR registers. In addition, obtains the BIOS and IMA measurements logs and sends everything to the Verifier. The Verifier checks the validity of the used AK and validates the signature of the Quote attestation. It then examines the BIOS and IMA logs for unexpected events by incrementally calculating the expected value of the PCR registers. It then compares the values of the PCR registers in the attestation with those derived. If it encounters validation problems or unexpected events it writes the result in a measurement log. Implementation details can be found in section B.3.

Chapter 7

Implementation

In this chapter we describe the technologies and dependencies used to successfully achieve an implementation that respected the design proposed in the chapter chapter 6. The first implementation choice was the programming language since it was necessary to create an implementation as low-level as possible for the purpose of integrations in future projects. As mentioned previously, our implementation is heavily based on the implementation proposed by [2]. It is a C++ project divided into several applications that we will present in the next chapter. As described earlier, the motivation behind this choice is the presence of an accurate description of the protocol and the presence of a security proof of the protocol itself.

7.1 ecc-daa project

The development of the thesis is strongly based on the ecc-daa project [24] from which it takes the implementation of cryptographic primitives DAA. ecc-daa is a C++ project composed of various sub-projects, one for each program, that perform the various functions related to DAA. The implementation developed in this thesis inherits various code blocks from the ecc-daa project and readjusts them for our purpose. The same structure of the ecc-daa project has been maintained for backwards compatibility and future use.

We see below the list of programs in the ecc-daa project along with a brief description of how they work:

- **provision_tpm**: has the aim of preparing the TPM. It takes care of the creation of an RSA endorsement key and makes it persistent. This avoids that the creation of the key is performed at each step as it is a very slow operation on hardware TPMs.

- **make_daa_credential**: generates the DAA key and performs the join step. It performs both the part of the Member and the Issuer by generating the credential that the Issuer issues to the member after the join phase.

- **daa_sign_message**: starting from the credential and the DAA key, it signs an input message and returns the signature together with some associated information, including the randomized credential created for the signature. It can also be configured to use a random *bsn* which will affect the presence of the points J and K in the signature output.

- **verify_daa_signature**: reads a signature and the associated randomized credential and checks the signature and credential.

- **daa_certify_key**: starting from a credential and a DAA key, it creates loads and signs an ECDSA key, generating a certificate for it associated with a randomized credential. It can also be configured to use a random basename which will affect the presence of the points J and K in the signature output.

- **daa_quote_pcr**: starting with a credential and a DAA key, it uses TPM2_Quote to read and sign the values of a PCR set. It then returns the signature with the associated randomized credential. It can also be configured to use a random *bsn* which will affect the presence of the points J and K in the signature output.

- **verify_daa_attest**: is used to verify the claims produced by the daa_certify_key or the daa_quote_pcr. The signature is verified together with the associated randomized credential.

7.1.1 Dependencies

The ecc-daa project has some dependencies that will also be used in the implementation of this thesis. Let us look at them briefly giving a brief description and an overview of their use in the project.

ibmswtpm2 is a software implementation of the TCG TPM 2.0 specification created by IBM based on a previous Microsoft implementation. It allows to launch a TPM simulator which you can connected to though a socket interface [25].

ibmtss is a user space TSS for TPM 2.0. It implements the functionality equivalent to (but not API compatible with) the TCG TSS working group's ESAPI, SAPI, and TCTI API's but with a different interface [26]. It comes with over 110 *TPM tools* samples that can be used for scripted apps, rapid prototyping, education, and debugging. It also has a web based TPM interface, suitable for practicing when unfamiliar with TCG technology and also useful for basic TPM management.

Apache Milagro Cryptographic Library (AMCL), is a multi-lingual and architecturally agnostic cryptographic library that supports elliptic curve cryptography, pairing-friendly curve cryptography, RSA, AES symmetric encryption and hash functions. It is designed from the ground up with side-channel attack resistance in mind. AMCL has now been extended and is being re-released as *MIRACL Core* [22]. The library is used for elliptic curves calculations, either in credential creation, signature creation or verification activities.

OpenSSL is a general purpose cryptography library that provides an open source implementation of the *Secure Sockets Layer* (SSL) and *Transport Layer Security* (TLS) protocols [27]. It also includes tools for handling big numbers, generating RSA private keys and Certificate Signing Requests (CSRs), checksums, managing certificates and performing encryption/decryption. The library is used as the basis for cryptographic functions, especially big numbers, hashes and for ECDSA signatures. Utilities have been built to reproduce the functions useful to the protocol.

7.2 Structure and implementation choices

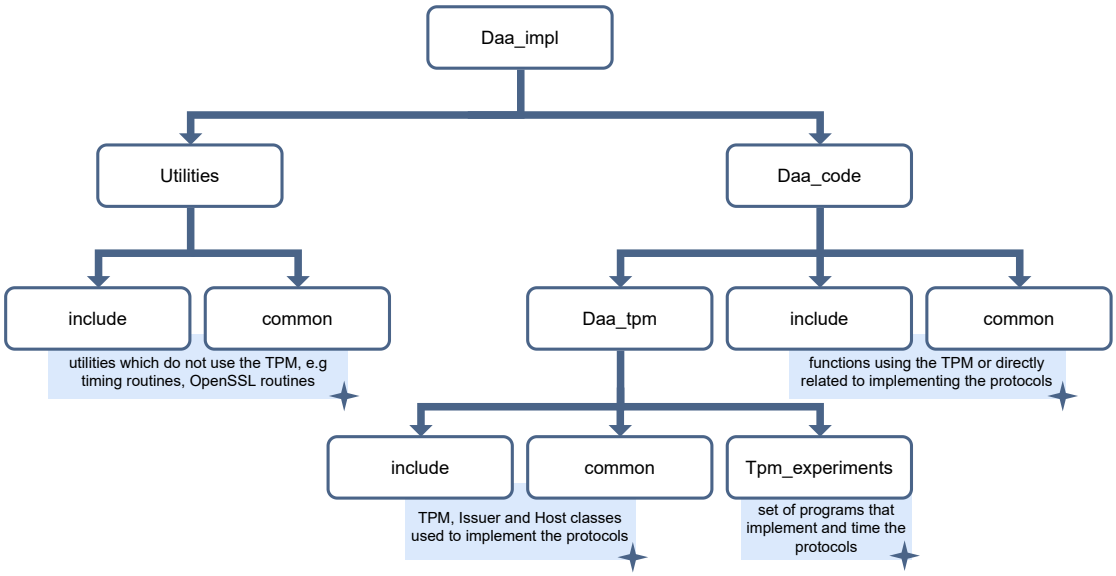

Figure 7.1. Project overview

In Figure 7.1 we can have a look at the code structure of the thesis project. It is a single C++ project divided into several folders with common code for all applications and application-specific folders. The "Utilities" folder contains wrappers to dependencies that do not use TPM, for example timing routines and OpenSSL routines. In the "Daa_code" folder, on the other hand, we can find wrappers for direct calls to TPM and support classes for the implementation of DAA protocols. All software main functions are placed under "Tpm_experiments".

All program inputs and all *Data Transfer Objects* (DTOs) used for REST calls in the various application protocols are intermediated by a tool called *ByteBuffer*, which allows standardization of the way C and C++ objects and structures are serialized and then sent in the protocols. It is possible to find more details in section B.4.

7.2.1 Dependencies and other useful project

For the development of the Issuer and Verifier, it was necessary to find a library that would allow us to be able to create a simple webserver that would smoothly support REST calls and the HTTPS protocol. To this end, **libhttpserver** was chosen. It is a C++ library for building high performance RESTful web servers with the aim to directly support all possible HTTP features and, thanks to its simple semantic, allow the user to mainly focus on his application and not on HTTP request handling details [28].

As for applications running on the Platform side, they need to contact the webservers to make REST calls. For this purpose, we chose **httplib** [29], a header-only library for making simple and expressive HTTP calls.

The other fundamental project to carry out the verification of the BIOS and the IMA measurements log is the project **IBM TPM Attestation Client Server (ACS)** from *kgoldman* (IBM) [8]. kgoldman is also one of the author of ibmtss e ibmswtpm2 which, as said, are the basis of the entire project. The ACS project has been fundamental to fully understand how the IMA manages measurement log files (ML). Unfortunately, current Linux kernel (5.15.0-33-generic) is using a different technique for the calculation of the hashes to be extended in the PCR than the one currently present in the ACS project.

7.3 Verification and revocation details

In this section we describe the Verifier configurations and behavior details regarding remote attestation verification by showing some examples and expected results. Finally, we will discuss how to obtain the revocation of a Member's signing credential and prevent that Member from using the AK again or may attempt to exchange a new one.

7.3.1 File measure whitelist and blacklist paths

During the *Push measurement protocol* the Verifier validated each BIOS and IMA measurement by using a set of expected measurements, the *whitelist database*, and a collection of paths not to be checked, called *blacklist path database*. Each entry in the whitelist database is composed of a pair of *file_name* and hash. The file_name refers to the absolute path (terminated with a '\0') of a possible file having a certain hash. Multiple records for each file_name can exist as they represent the possible expected versions of a file in a specific file path. To date, the database is a JSON file which can be configured as startup parameter. Each record have the same scheme as that shown in Listing 7.1.

Listing 7.1. Example of a whitelist database entry

```
1 {
2     "file_name":"/usr/lib/x86_64-linux-gnu/libgcc_s.so.1\u0000",
```

```
3     "hash":"63eb91b5d726e401c5affb2a035e175c374cae75cdf25a1f..."
4 }
```

In contrast, the *blacklist path* is a database containing an set of absolute paths that should not alerts in case a file that matches the path has validation problems. In detail, it is a list of regex that validate the file_name to decide whether to emit alert. To date, they are implemented with a JSON array persisted in a configurable file as startup parameter. Each entry corresponds to a regex that, in case of match to file_name, does not emit alert even if the measure is invalid. In Listing 7.2 can be found an example showing the insertion in the path blacklist of any file whose absolute path begins with `/home/user/daa_test/Verifier/` or `/home/user/daa_test/Member/`:

Listing 7.2. Example of a path blacklist database

```
1 [
2     "^\/home\/user\/daa_test\/Verifier\/.+$",
3     "^\/home\/user\/daa_test\/Member\/.+$"
4 ]
```

Register mode

To easily enable the generation of the measurement whitelist database from a trusted Member, it is possible to start the Verifier in *Register Mode* (as reported in user manual in section A.4) and proceed with a remote attestation with member application. By doing so, the Verifier, whenever it detects a measurement absent in the whitelist, will proceed to add it to the database by adding the file_name and hash pair. In any case, within the measurement log there will still be an alert for each measurement not found.

7.3.2 Measuration warnings and errors

The results of each measurement performed with the *Push measurement protocol* are saved by the Verifier in a specific directory called `measurement_logs` in the base directory set in startup configurations. We can find an example of measurement log in Listing 7.3.

Listing 7.3. Example of measurement log without alert

```
1 [INFO]: Challenge:
      9da708ad90cd490193c9bf2f3246b6f9e29b4a406750745e118cba5639b9d829
2 [INFO]: AK public data:
      00580023000b000504720000001000180000b00030010002096b488e
      8006507e5c41a557ce61dcafd8eb29e0a1d8fa27822a07cf48e8977de0020899411cf3363c0a757
      87d6f19852130627674900951111b064235ff842a4cef9c
3 [INFO]: Attestation:
      ff54434780180022000b12113ef24b72de8ed6ed9855bc1e2a75aa1eb9
      8ce14ef06f68acc433aa34866d00209da708ad90cd490193c9bf2f3246b6f9e29b4a406750745e1
      18cba5639b9d82900000000014b3fdc000000000000000001201910230016363600000001000b03
      ff07000020efbfcff7383329c244b54a97d147627305b1a2c35f67d20c9f890ad7afb80e21
```

```
4 [INFO]: nc: d24b334e683396d962d76e984736b61f2abe519b4d9bfa8112f4f0f6037994d2
5 [INFO]: sig_s:
      bc9072003a269b99f6a098a0db1bdf47f390191ae43261179cfeb96978702beb
6 [INFO]: h2: 26a7633d389fa253fd84ccd6a92c49d903a2fdad08b23e9d813fcd362803c671
7 [INFO]: isPaddedBanks: 0
8 [INFO]: Verification ok
```

In the header of each measurement log it is possible to find the various input received by the Member for that measurement: the challenge, the AK used in the signature, the attestation produced by the TPM in the quote and the related signature. In addition, if the application is started with *log_level=2*, also the entire BIOS and IMA log will be logged in the same file.

Warnings

In case a BIOS or IMA measurement is invalid, ie. that is not present in the whitelist measurements and not matching the blacklist paths, this will be reported in the log as WARN. The result will be a log similar to the one in Listing 7.4.

Listing 7.4. Example of measurement log with alerts

```
1  ...
2  [WARN]: Alert: time between two measurement is greater than the timeout. 542
       seconds passed from last measurement.
3  [WARN]: The reset counter was changed
4  [WARN]: The restart counter was changed
5  [WARN]: The safe flag is false
6
7  [WARN]: Unmatched Bios Measurements, 9 found:
8  [WARN]: Hash:
       194ed60de92dd28d096e142b6b68b8da78cd9012ea40beff9aa2be92e8491738 Pcr: 9
9  [WARN]: Hash:
       cc6aa04ba7cefc2fb8c7b479369c23618ac9d0e03c256d3113044dd4e2dcf2c5 Pcr: 8
10 [WARN]: Hash:
       3ba11d87f4450f0b92bd53676d88a3622220a7d53f0338bf387badc31cf3c025 Pcr: 4
11 ...
12 [WARN]: Unmatched Ima Measurements, 523 found:
13 [WARN]: Hash:
       54c863325bb96debab77e600bf76806085d2cecf3f16297bfc921f336e724133 File
       name: boot_aggregate
14 [WARN]: Hash:
       83db13d7730b34530fc27161ed9d9c9c54e274d45d0e3e56db766ff2966d6b05 File
       name: /usr/lib/modprobe.d/blacklist_linux_5.15.0-52-generic.conf
15 [WARN]: Hash:
       f4877bdf7d58fb0481713bcad5e8e9d5408bf730eaf914ef3948c5f9989b67ec File
       name: /usr/lib/modules/5.15.0-52-generic/modules.softdep
16 [WARN]: Hash:
       4151e0b3705f33575e86cc2196491f2b91d7fd25fc31f1bef807dfafc29ea4b7 File
       name: /usr/lib/modules/5.15.0-52-generic/modules.dep.bin
17 ...
18 [INFO]: Verification ok
```

The log contains a row per any invalid measurement. In the case of BIOS measurements, they contain the PCR register and the hash of the measurement.

Instead, in case of IMA measurements, they contain the absolute paths and the hash of the measured file. If the time between two following `POST/measurement` exceeds the set time, a warning containing the number of seconds elapsed between the two measurements by will be emitted. Moreover, a warning will be also emitted if the two measurements reports different *reset* or *restart counters* as essential for detecting TPM or platform restarts.

Errors

In case of absence of errors, the log will contain a row showing *"Verification OK"*. Conversely, as shown in Listing 7.5, if problems concerning the signature verification, challenge verification or AK verification arise, the log will contain one or more ERRORs with the relative description of the identified problem.

Listing 7.5. Example of errors in the measurements log

```
1  ...
2  [ERROR]: Verification error: Error in challenge checking: challenge not
       registered.
3
4  [ERROR]: Verification error: Error in challenge checking: registered ak_pd
       != quote ak_pd
5
6  [ERROR]: Verification error: Attestation extra data != challenge
7
8  [ERROR]: Verification error: Measurement's digest not compatible with
       attestation's one
9
10 [ERROR]: Verification error: Signature check failed
```

7.3.3 AK revocation

As previously mentioned, during the *AK exchange protocol* the Verifier, after having verified the Member's AK, inserts it in a database of the active keys together with the Member pseudonym PT_K. In the proposed implementation, the database is a JSON file passed as a parameter to the application and contains an array of records structured as in Listing 7.6. Details can be found in the user manual in section A.4.

Listing 7.6. Example active AKs database

```
1 [
2   {
3     "ak_pd": "00580023000b000504720000001000180000b00030010002096b488e80...",
4     "l_m": "ef417945a3d1adf85d4c90a8414e764136f4316d5d06ef2ec5a652b1...",
5     "lmt": 1665345806,
6     "pt_k": "000200200e796d3118420d02e95b252de4c499e2d34f607b2ea41e2cbd..."
7   },
8   ...
9 ]
```

In case a Member turns out to be malicious, or in the general case one wants to revoke of the AK exchanged with a particular Verifier, the Verifier must proceed to edit the entry for that AK in the database. To give a practical example, let us imagine the case where we have a Member whose attestation Quote makes us infer it underwent an attack and whose DAA credentials may have been stolen together with the AK exchanged. By looking at the measurement_log related to the suspected measurement, we can retrieve the logged AK public data *ak_pd*. Once retrieved we modify the database of active AKs by searching for the entry containing as ak_pd the value sought. In particular, we can modify the ak_pd value by inserting any value different from the initial one. In this way, future attestations using that AK will not be accepted and a warning/error will be issued alerting to the attempted use of an nonexistent key. This is regardless of the validity of the created attestation. By doing so, the AK is revoked and, due to the presence of the *PT_K* in the database, the Member's DAA credential cannot be used to certify new AKs without the Verifier blocking and warning of the attempt.

The proposed method can be considered a *Verifier blacklist revocation* of the DAA credential. To date, this is the only revocation method supported.

7.4 Considerations

We conclude this chapter with consequential considerations for the proposed implementation. In particular, we spend a few words about the possibility of the Member's anonymity being lost despite the DAA due to other factors that may cause the identity to be discovered. We will also discuss the Issuer, its ability to issue signatures, and the need for trusts against it.

7.4.1 Member anonymity

The DAA is a tool that helps lay the groundwork for remote attestation that safeguards the signer's anonymity, however, may not be sufficient to guarantee it. As described in subsection 5.1.3 and implemented in this thesis, the use of a pseudonym allows us to have a way of identifying a Member who has already exchanged an AK but it is achieved at the expense of the privacy of the Platform. This compromise enables us to obtain a protocol similar to that of traditional remote attestation and being able to have a history of the attestations received allowing us to do a temporal analysis of them, detecting changes in system state, such as resets, or measuring the time between two subsequent attestations.

In addition, another important aspects for Platform anonymity are the BIOS log and IMA logs sent to the Verifier at each remote attestation. In these logs there is much information concerning the Platform that could, in specific contexts, be sufficient not only to characterize the Platform and collecting data about it, but also to discover its identity. For instance, if considering a well-known configuration file logged in the IMA log and containing a Host identifier, it would be possible retrieve its content, thus Platform identity, through a dictionary attack. Another

example could be a software in execution uniquely by a specific host whose identity is known: the presence of the particular software hash in the IMA log would directly correlate to that Platform identity. On the other hand, in case from the attestation history or/and form the BIOS and IMA log the Platform could be identified, it would make the solution from a privacy point of view equivalent to traditional remote attestation. It is therefore important to consider the context in which the devices operate in order to determine the impacts regarding Platform privacy.

7.4.2 Trust of the Issuer

The Issuer can autonomously emit new credentials, indeed it could generate a DAA key and autonomously perform the Join protocol to obtain any number of DAA credentials. No group Members or Verifier could notice either the creation or the use of these credentials. To the Verifier, their usage would seem similar to the one by any other legit Member. Nonetheless, the Issuer could not impersonate another Members nor use their pseudonyms, since it does not possess the Member's DAA key. Indeed, each Verifier verifies the Member pseudonym PT_K, which directly depends on the DAA key. We can find in the role of the Issuer some parallels with that of Certification Authority (CA) has in the traditional remote attestation. Indeed, they both generate transitive trust toward the signer through a certification of its signature key. In the case of traditional remote attestation, the CA implicitly or explicitly certifies the correlation between the public key and the signer identity. On the other hand, in the case of DAA attestation, through the credential release, the Issuer certifies that a specific signer is reliable group Member capable of producing signatures that can be associated with the identity of the group itself.

Chapter 8

Test and validation

In this chapter we propose some of functional and performance tests on the developed applications. We initially describe the testbed used in test execution and its configuration, than, for each protocol used by the applications we will see some tests that are intended, on one hand, to show in practice how to perform certain operations, and on the other hand, to provide an overview of the execution times of the applications and their analysis.

8.1 Testbed

The test environment provides the execution of the various applications in separated *virtual machines* (VM) in order to simulate remote attestations in a distributed environment. The test environment consists of 3 VMs:

- the **Issuer VM** simulates the machine in which the Issuer is run. This is configured with the group secret key and executes the "Issuer" application waiting for candidates for the group;
- the **Verifier VM** simulates a generic Verifier listening for requests from various platforms. It executes the "Verifier" application with a preloaded IMA whitelist to make the result of attestations more realistic;
- the **Platform VM** simulates the Platform to be attested through a virtual TPM instance mounted as a device and managed by the hypervisor. This is configured via "Provision_tpm" for the generation of an EK that will then be used in the various tests.

All VMs uses a fresh installation of Ubuntu attempting to simulate an environment as standard as possible. The VM were deployed in the same host, based on *Proxmox 7,2.11* [30] a Linux open-source distribution based on Debian which allows graphical management of VMs and virtual TPMs, as well as allowing for easy backups and restores of these. Proxmox 7,2.11 virtualization is based on *qemu 7.0.0-3*. Moreover, each VM uses *Kernel-based Virtual Machine* (KVM). The host has the following configuration:

- CPU: AMD Ryzen 3900X (x86, 12 core, 24 thread, 3.8GHz) supporting hardware-assisted virtualization;
- RAM: 16x2GB DDR4 3200Mhz CAS 16;
- SSD: NVME 80K IOPS.

8.1.1 VMs setup

Each of the VMs presented was configured from a fresh installation of Ubuntu 22.04.1 LTS based on Linux 5.15.0-52-generic kernel. Following the OS installation, the environment setup procedure proposed under section B.5 was performed and then the various applications were installed in `/etc/{appname}`. In addition, we create `/home/thesis/daa_test/{actorname}/` as the base directory for application inputs and outputs. Now each VM is ready and it is possible to proceed with application execution.

Issuer VM

For the Issuer configuration, was used as *group issuing key* the one proposed in the Listing 8.2. From this we derive the *group public key* reported in Listing 8.3. Finally, the verifier application is started with the parameters shown in Listing 8.1.

Listing 8.1. Issuer execution command

```
1 /etc/issuer \
2 -d /home/thesis/daa_test/Issuer \
3 -fisk fisk.data \
4 -fipk fipk.data
```

Listing 8.2. `fisk.data` containing group issuing key

```
1 0002002065a9bf91ac8832379ff04dd2c6def16d48a56be244f6e19274e97881a776543c0020
2 126f74258bb0ceca2ae7522c51825f980549ec1ef24f81d189d17e38f1773b56
```

Listing 8.3. `fipk.data` containing group public keys

```
1 000200920002004600020020c824b17d4f4e845eebfdcaabc1eccef8afdc3ef2f8e2eabdc230
2 4a20e6b0b1e90020b0fc6dba0bda080e2f4a7965b2fdbf5fc6b2678683ae35d4004d1ac483f6
3 12920046000200206e20706db66d3abce4a8a4b5fb9d87e624a770fe835518bfadf449a6e65f
4 7c6c0020a48aa8741b05553289a2424d0a5ed85f5e77ca139428f22c88e8346cb863307e0092
5 00020046000200204e705fe26bf2918ce1d22cc0c956e570c7260cae27113adbf61e3b9f1e9a
6 5dce002087a097c489d8cb8f570ea621e6c60f858be3abf11de858e2202d579c1d7a22430046
7 00020020c09a8b38bc9bf70580e23904633c63655fc61f28a04cab527596c5d8b690d7e60020
8 54bed983371e5af0d4ac6e80af66ee5b2d5fbfe006220ac4f7384e601083739c
```

Verifier VM

To make performance tests more complex and truthful and to evaluate the results of functional tests more cleanly, the Verifier was initialized with a whitelist database of BIOS and IMA measurements preloaded with about 70000 records. Instead, the database of already exchanged AKs is initially empty. Verifier startup configurations are shown in Listing 8.4. The fixed *bsn* to be used in AK certification by the Member is indicated in Listing 8.5. Finally the group public key is naturally the same as that reported in Listing 8.3.

Listing 8.4. Verifier execution command

```
/etc/verifier \
-d /home/thesis/daa_test/Verifier \
-fipk fipk.data \
-fbsn bsn.data \
-fakdb aks_database.json \
-fbiosw bios_whitelist.json \
-fimaw ima_whitelist.json
```

Listing 8.5. `bsn.data` containing the fixed bsn to be used in the AK certify

```
000200920002004600020020c824b17d4f4e845eebfdcaabc1eccef8afdc3ef2
```

Platform VM

The Platform VM is the one that simulates the Platform to be attested, so it will contain a virtual TPM2 revision 1.64 that it will use after an initial phase of configuration and creation of the EK. The configuration is accomplished by running "Provision_tpm" application with the command shown in Listing 8.6. In addition we prepare the configuration file `pcr_configuration.json`, as in Listing 8.7, which will later be used in the Push measurement protocol. Functional and performance tests will use this VM to initiate execution of the various protocols and evaluate outcomes.

Listing 8.6. Provision_tpm execution command

```
/etc/provision_tpm -t
```

Listing 8.7. `pcr_configuration.json` - PCRs Quote configuration file

```
{
 "algorithm": "SHA256",
 "selectedPcrs": [0,1,2,3,4,5,6,7,8,9,10],
 "imaMeasurementType": "FULL_SIZE_BANKS"
}
```

8.2 Functional Test

Functional tests are intended to show the operation and expected results of the designed protocols so that the functioning of the protocols can be better understood. The tests were run in the testbed described above, specifically we are going to run them on the Platform VM since Issuer and Verifier are servers that handle requests without the need for restart.

8.2.1 Join protocol

The purpose of the test is to verify the Join protocol and the release of a credential through the interaction between the Member_join and the Verifier. Checks on EK are disabled to allow multiple iterations without being blocked by the fact that EK is already being used. To run the test, we start the Member_join as in Listing 8.8 and we expect the DAA key and credential will be generated. In particular, we expect the credential to be generated in `/home/thesis/daa_test/Member/credential.data` containing data similar to that in Listing 8.9. The DAA key, on the other hand, is created and saved in `/home/thesis/daa_test/Member/daa_key.data` and has a similar structure to Listing 8.10.

Listing 8.8. Member_join execution

```
/etc/member_join \
-t \
-d /home/thesis/daa_test/Member \
-fipk fipk.data \
-fdk daa_key.data \
-fc credential.data \
-iu https://issuer:8080 \
-ll 2
```

Listing 8.9. Credential file

```
00040046000200203a2c3c92deab24382674ff4896c425f441ae852dac453f29b88bddb803438
54b002068b24ad56b59a09bcb562fdaf8a486cf2bf64fa8d190b3c630399fb8afccfea9004600
020020453910c08803010b9f0009965146d55fbabd501c188fd79d8753e2d09db623560020de4
20f20eb7b18f5917a222f93ac6b872ea7deefdab1d2e5147ba713cd6b540900460002002023ec
579ae6bd2c98846b449bd2039e7a3ab77583fe799ab0d835f00f41b955690020269c5169ca59f
36c08a51a906e8cef7e7082e3080887b42c30e29c3dde7a0dd3004600020020908541a7e7d935
bb300e795fd2e33f9ce7d9a9d2fc24ae61d71c66110e31d1770020e1019d66a0b7f02e08d564d
975dadedb66c5ccb98779368fe5a37ac967bf763b
```

Listing 8.10. DAA key file

```
00020080007e00203f03bf8ba678739201d6c616edd3a4e80a13e4ca9c81bfd5bc2c054b6d417
24500103b8700c846539a5d34aa45da1f568053075545bb36e6fa71096c5d34221e6e6ab09484
d860eee6b3ff9c7f39830a0b0b5e6d83070eae8a40a938ea7aeb803a62f95236ea68eb5246c6e
f620ce7a406ad9a2f1f0fab5f1dd3a354005c005a0023000b000504720000010001a000b0001
0010001000204417086f07c4845bd21adf56d6eab443e8f4c6a106bb3a575814d41073f270c70
020e55bb5ea5db9c035f4224f9074e350947a02dbda87500adfc4fa171593bb04e8
```

Instead, in the case of reusing the same challenge retrieved from the `POST join/issuercredential` or in case the request to the `POST join/makefullcredential` is invalid the Issuer returns an error preventing the credential from being released.

8.2.2 AK exchange protocol

Let us now try to verify the behavior of the AK exchange protocol and the interaction between the Member_ak and the Verifier. We run Listing 8.11 and expect to get in `/home/thesis/daa_test/Member/ak.data` a file valued similarly to Listing 8.12.

Listing 8.11. Member_ak execution

```
/etc/member_ak \
-t \
-d /home/thesis/daa_test/Member \
-fak ak.data \
-fipk fipk.data \
-fdk daa_key.data \
-fc credential.data \
-vu https://verifier:8081
-ll 2
```

Listing 8.12. AK data file

```
007e00202a1c0344077be5ea73c4deca5489a7eefddac04afb12fedcf8ab947a296c6d6a00107
2b535402556129cd12e2375bf9ba8900dbdb38261370f7dfff4781784d1ec456b6837141d9bc0
766eb5abdbc556ebc50a894ae81bd7509dbef25002846f401620b783ccda5a952b17b6602292e
98285d6088f9b7f9096935907\r\n
00580023000b000504720000000100018000b0003001000202e51b951e86999c526aac9a49a71b
d616ed8248789accf51f90b78d0eb389ea600207d93a49380ef8687e264bbc3fbd8238f6823c6
4eccc2ac8a56dd4754005677b2
```

In case we try to use the same credential a second time we expect the Verifier to detect the attempt in the `POST /akRegister` due to the verification of the PT_K and not accept the AK.

8.2.3 Push measurement protocol

Finally with regard to the Push measurement protocol, the execution of the Member is expected to result in the creation of a log in `/home/thesis/daa_test/Verifier/measurement_logs` that would later allow us to obtain the information needed to identify problems in the Platform attestation. In particular, we test the behavior of an attestation subsequent to the creation, modification, and execution of a binary with root privileges. This will trigger measurement by the IMA that will be saved in the IMA log and consequently we expect the Verifier to take this into account during validation.

In order to do this, we let the IMA measure an executable twice, first for its creation and execution, then for its modification and re-execution. We obtain this

result by proceeding as in Listing 8.13, by making a temporary copy of `/bin/cat` and then replacing the copy with `/bin/cp`. Running the Member as in Listing 8.14 we expect, as in Listing 8.15, that the report contain the two executions both reported as WARN because even if the hash of the two applications (`/bin/cat` e `/bin/cp`) are present in the whitelist database, these were run in a path that is not the standard, whitelisted path (`/bin`), but were instead performed in `/home/thesis/test`.

Listing 8.13. Force double IMA measurement

```
1 cd /home/thesis/test
2 cp /bin/cat ./cat #Copy /bin/cat binary to current directory
3 ./cat justexec #Execute the cloned copy of /bin/cat to let IMA measure it
4 cp /bin/cp ./cat #Overwrite the cloned /bin/cat with /bin/cp
5 ./cat justexec #Re execute to let IMA re-measure it because data changed
```

Listing 8.14. Member execution

```
1 /etc/member \
2 -t \
3 -d /home/thesis/daa_test/Member \
4 -fcfg pcr_configuration.json \
5 -fak ak.data \
6 -vu https://verifier:8081 \
7 -fbm /sys/kernel/security/tpm0/binary_bios_measurements \
8 -fim /sys/kernel/security/ima/binary_runtime_measurements
```

Listing 8.15. Verifier measurement report

```
1 [WARN]: Unmatched Ima Measurements, 2 found:
2 [WARN]: Hash:
      dd5526c5872cce104a80f4d4e7f787c56ab7686a5b8dedda0ba4e8b36a3c084c File
      name: /home/thesis/test/cat
3 [WARN]: Hash:
      5cbd6e8d5aaa714a0910f5ee7710004cbf0a22c940eda1abec6f0194f430b94c File
      name: /home/thesis/test/cat
```

If instead we add to the blacklist paths the entire directory `/home/thesis/test/` and all its sub files, we expect to no longer find WARNs in the report. So let's proceed to configure the blacklist path database as in Listing 8.16 and, by re-running the Member, we get a report without the 2 WARNs as expected.

Listing 8.16. Verifier blacklist path file

```
1 ["^\/home\/thesis\/test\/.+$"]
```

8.3 Performance Test

In this section can be found performance tests of the applications, the collected measurements and their analyses. In order to obtain detailed information on the

execution timings of the various parts of each application, timers were inserted within the code. They measure execution time for each part and report it in the log file. From there they are extracted, aggregated and finally collected into tables showing the mean and standard deviation of the most impactful measures.

8.3.1 Join protocol

For measuring the timing of the Join protocol, it was performed a total of 500 times through re-execution of Listing 8.17. In each iteration the Issuer instance is the same and executes the protocol without being subject to restart. To make sure that each test uses the TPM in a state that is as clean as possible, at each iteration, the *tpm2_clear* is called and then the EK is recreated via the Provision_tpm. Results are summarized in Table 8.1.

Listing 8.17. Join protocol execution cycle

```
1  tpm2_clear
2  provision_tpm -t
3
4  rm /home/thesis/daa_test/Member/credential.data
5  rm /home/thesis/daa_test/Member/daa_key.data
6
7  /etc/member_join \
8  -t \
9  -d /home/thesis/daa_test/Member \
10 -fipk fipk.data \
11 -fdk daa_key.data \
12 -fc credential.data \
13 -iu https://issuer:8080
```

Table 8.1. Join protocol time report

Time (ms)	Mean	SD
Create and load DAA key (TPM2_Create, TPM2_Load)	4,67	0,35
GET /public/groupdata	4,19	0,21
POST /join/issuercredential	2,03	0,14
TPM2_ActivateCredential	3,08	0,23
TPM2_Commit	1,45	0,18
TPM2_Hash	1,22	0,12
TPM2_Sign (DAA)	1,53	0,22
POST /join/makefullcredential	14,2	0,52
Total execution time	56,32	1,18

8.3.2 AK exchange protocol

In the case of the AK exchange protocol as well, 500 executions of the protocol were carried out through iterations showed in Listing 8.18. In each iteration the

Verifier instance is the same and executes the protocol without being subject to restart. For this reason, re-executing the Join protocol is critical to requesting a new DAA credential that would otherwise lead to receiving an error message due to the PT_k already being in the database. Results are summarized in Table 8.2.

Listing 8.18. Ak exchange protocol execution cycle

```
tpm2_clear
provision_tpm -t

rm /home/thesis/daa_test/Member/credential.data
rm /home/thesis/daa_test/Member/daa_key.data
rm /home/thesis/daa_test/Member/ak.data

/etc/member_join \
-t \
-d /home/thesis/daa_test/Member \
-fipk fipk.data \
-fdk daa_key.data \
-fc credential.data \
-iu https://issuer:8080

/etc/member_ak \
-t \
-d /home/thesis/daa_test/Member \
-fak ak.data \
-fipk fipk.data \
-fdk daa_key.data \
-fc credential.data \
-vu https://verifier:8081
```

Table 8.2. Ak exchange protocol time report

Time (ms)	Mean	SD
Create and load AK (TPM2_Create, TPM2_Load)	4,23	0,30
TPM2_Commit	2,25	0,23
TPM2_Certify (DAA)	1,83	0,11
GET /akRegister/publicData	3,18	0,19
POST /akRegister	36,87	5,36
Total execution time	52,13	5,49

8.3.3 Push measurement protocol

Finally, regarding the Push measurement protocol once again it was run for 500 iterations. For each run, as shown in Listing 8.19, an execution of Member were performed. In each iteration the Verifier instance is the same and executes the protocol without being subject to restart. The results of the times concerning the Platform are summarized in Table 8.3. On the Verifier side, on the other hand, the measured times are shown in Table 8.4.

Listing 8.19. Push measurement protocol execution cycle

```
/etc/member \
-t \
-d /home/thesis/daa_test/Member \
-fcfg pcr_configuration.json \
-fak ak.data \
-vu https://verifier:8081 \
-fbm /sys/kernel/security/tpm0/binary_bios_measurements \
-fim /sys/kernel/security/ima/binary_runtime_measurements
```

Table 8.3. Push measurement protocol: Platform time report

Time (ms)	Mean	SD
TPM2_Load (AK)	2,6	0,1
POST /measurement/challenge	3,25	0,15
Read BIOS events	1,86	0,06
Read IMA events	116,8	2
TPM2_Quote	1,9	0,1
Data processing for POST /measurement	343	5
POST /measurement	6813	46,5
Total execution time	7280	47

Table 8.4. Push measurement protocol: Verifier time report

Time (ms)	Mean	SD
Request parse	536	6,5
Process BIOS measurements	9,6	0,3
Process IMA measurements	6143	20

8.3.4 Considerations

It can be seen that the execution times of the TPM functions are very low, on the order of 1-4ms. This is due to the use of a virtual TPM in the tests, which if it were replaced by a physical TPM would make the processing times substantially longer. In fact, the TPM was born with the idea that it could be an inexpensive chip, and this causes hardware implementations to sacrifice processing speed in favor of a cheaper implementation. Today the market offers various hardware implementations of the TPM each with very different performances as well. To give an idea of the possible processing times of a typical physical TPM we propose here the Table 8.5, extrapolated from [2] showing the execution times of various APIs using a Raspberry Pi 3 and a TPM evaluation module created by Infineon [31].

Another important consideration is the timing of sending, parsing and validating the IMA log in the Push measurement protocol. These take up most of the protocol execution time as they require having to process tens of thousands of IMA log entries. To minimize the impact of the IMA log, it is convenient to redesign the protocol by not sending the entire IMA log each cycle and revalidating it each time but instead incrementally sending only the new entries thus greatly reducing the size of the requests and the protocol processing time.

Table 8.5. Hardware TPM reference times

Time (ms)	Mean	SD
TPM2_Create	219.2	2.2
TPM2_Load	36.4	2.6
TPM2_ActivateCredential	219.4	3.2
TPM2_Commit	224.0	1.0
TPM2_Hash	26.2	2.0
TPM2_Sign (DAA)	65.5	3.7
TPM2_Certify (DAA)	53.9	3.0
TPM2_Quote (DAA)	53.6	2.0

Chapter 9

Conclusions

The purpose of the thesis project was to implement a system that allows remote attestation using DAA in a manner as similar as possible to traditional remote attestation so that it can be easily integrated into existing attestation frameworks. We therefore made an implementation of a hybrid attestation scheme that uses DAA to perform an anonymous authenticated exchange of an Attestation Key with a Verifier and then uses it to perform traditional remote attestations. In this way, we achieved the benefits of performance and architectural similarity with existing solutions and, at the same time, increased the privacy guarantees of the Platform if it is used in appropriate application contexts.

The implementation provides a valuable baseline for future developments and, due to the architectural similarity and modularity of the design, for integration into any existing attestation frameworks. However, there are some open issues and criticalities that require further development and are beyond the scope of this thesis. The first open issue concerns the privacy of the Platform. In fact, the pseudo-anonymity used as a compromise for obtaining this hybrid solution, combined with the presence of possible privacy-sensitive data in the BIOS and IMA measurement logs could, in particular contexts, provide sufficient elements to identify the attester. There are also technical improvements that can be made to the proposed implementation. Primarily, as visible from the performance tests, the heaviest component of the entire process is the sending and verification of the IMA log, which can be significantly improved by creating an incremental attestation mechanism that sends only the new measured events between attestation cycles instead of sending and reverifying the entire log each cycle. Secondly, it is relevant for future developments to implement mechanisms to verify the inputs of the various protocol APIs and applications in order to prevent attacks and make implementations more robust. Finally, it would be very valuable to extend the current implementations also to application contexts where it is necessary to share the same TPM among various attesters, such as in the case of OS-level virtualization, which is becoming the standard solution of today's distributed systems.

Appendix A

User manual

The following sections present the user manuals of the various applications, in particular the inputs, the outputs and environment variables parameters of each application. For each parameter, we will specify in the following order: the parameter's short name and full name, any input it requires, any default values for each input and finally a description of them. In addition, in case an input requires a particular structure or encoding, this will be specified in the description. In the encodings, there is extensive use of serialized *ByteBuffers*, which are described in detail in section B.4. When we say that we expect input in ByteBuffer format we mean that we want a ByteBuffer encoded in hexadecimal as input.

For example, if we want to say that the parameter with short name "-fipk" and full name "--fileissuerpublickeys" accepts as input a file path that has inside encoded in hexadecimal a container ByteBuffer with two *G2_points* representing the X and Y points of the group public key, we write:

- -fipk, --fileissuerpublickeys `<file>` - `<file>:=ipks.data` - File containing group public keys X and Y encoded as ByteBuffer(G2_point, G2_point).

All applications have the following parameters in common that allow them to control the log level and obtain the version or help menu:

- -ll, --loglevel `<level>` - `<level>:=0` - Log level can be one of the following values [0: WARN, 1: INFO, 2: DEBUG];
- -h, --help – Show help menu;
- -v, --version – Show code version.

A.1 Issuer

[**Input**]:

- -d, --datadir `<directory>` - `<directory>:=.` - Data directory where are located other files;
- -fipk, --fileissuerpublickeys `<file>` - `<file>:=ipks.data` - File containing group public keys X and Y encoded as ByteBuffer(G2_point, G2_point);
- -fisk, --fileissuersecretkeys `<file>` - `<file>:=isks.data` - File containing group secret keys data x and y encoded as ByteBuffer(BIG, BIG);
- -fhcert, --filehttpscertificate `<file>` - File used as https certificate, if not present it will use only http;
- -fhkey --filehttpskey `<file>` - File used as https key, if not present it will use only http.

A.2 Provision_tpm

[**Input**]:

- -t, --dev - Use the TPM device;
- -s, --sim - Use the TPM simulator.

[**Environment variables**]

- TPM_DATA_HOME - `TPM_DATA_HOME=/var/TPM_data` - Base directory for IBM TSS files.

A.3 Member_join

[**Input**]:

- -d, --datadir `<directory>` - `<directory>:=.` - Data directory where are located other files;
- -fipk, --fileissuerpublickeys `<file>` - `<file>:=ipks.data` - File containing group public keys X and Y encoded as ByteBuffer(G2_point, G2_point);
- -iu --issuerurl `<url>` - Issuer URL to be used in API call;
- -t, --dev - Use the TPM device;
- -s, --sim - Use the TPM simulator.

[**Input/Output**]:

- -fdk, --filedaakey `<file>` - `<file>:=daa_key.data` - If the file exists, it will be used as data source by the DAA key. Otherwise, it will be created a DAA key whose data will be inserted in the file during execution. The DAA key file is encoded as ByteBuffer(TPM2B_PRIVATE).

[**Output**] :

- -fc, --filecredential `<file>` - `<file>:=credential.data` - Output file for the generated credential encoded as ByteBuffer(Daa_credential).

[**Environment variables**]

- TPM_DATA_HOME - `TPM_DATA_HOME=/var/TPM_data` - Base directory for IBM TSS files.

A.4 Verifier

[**Input**]

- -d, --datadir `<directory>` - `<directory>:=.` - Data directory where are located other files;
- -reg, --registermode – By default, the register mode is deactivated, but it is activated if the flag is present;
- -fipk, --fileissuerpublickeys `<file>` - `<file>:=ipks.data` - File containing group public keys X and Y encoded as ByteBuffer(G2_point, G2_point);
- -fimab, --fileimapathblacklist `<file>` - `<file>:=ima_path_blacklist.json` - File with a JSON array of regex of blacklist paths;
- -fbsn, --filebsn `<file>` - `<file>:=bsn.data` - File containing the fixed *bsn* of this verifier encoded as Bytebuffer;
- -fhcert, --filehttpscertificate `<file>` - File used as https certificate, if not present it will use only http;
- -fhkey --filehttpskey `<file>` - File used as https key, if not present it will use only http.

[**Input/Output**]

- -fbiosw, --filebioswhitelist `<file>` - `<file>:=bios_whitelist.json` - File with a JSON array of "hash"-"pcr_index" JSON objects;
- -fimaw, --fileimawhitelist `<file>` - `<file>:=ima_whitelist.json` - File with a JSON array of "file_name"-"hash" JSON objects;
- -fakdb, --fileaksdatabse `<file>` - `<file>:=aks_database.json` - File with a JSON array of "ak_pd"-"pt_k" JSON objects.

A.5 Member_ak

[**Input**]

- -d, --datadir `<directory>` - `<directory>:=.` - Data directory where are located other files;
- -t, --dev - Use the TPM device;
- -s, --sim - Use the TPM simulator;
- -vu --verifierurl `<url>` - Verifier URL to be used in API call;
- -fipk, --fileissuerpublickeys `<file>` - `<file>:=ipks.data` - File containing group public keys X and Y encoded as ByteBuffer(G2_point, G2_point);
- -fdk, --filedaakey `<file>` - `<file>:=daa_key.data` - File containing DAA key's private data encoded as ByteBuffer(TPM2B_PRIVATE);
- -fc, --filecredential `<file>` - `<file>:=credential.data` - File containing DAA credentail encoded as ByteBuffer(Daa_credential).

[**Environment variables**]

- TPM_DATA_HOME - `TPM_DATA_HOME=/var/TPM_data` - Base directory for IBM TSS files.

[**Output**]

- -fak, --fileattestationkey `<file>` - `<file>:=ak.data` - Output file containing AK's public data encoded as ByteBuffer(TPM2B_PUBLIC).

A.6 Member

[**Input**]

- -d, --datadir `<directory>` - `<directory>:=.` - Data directory where are located other files;
- -t, --dev - Use the TPM device;
- -s, --sim - Use the TPM simulator;
- -vu --verifierurl `<url>` - Verifier URL to be used in API call;
- -fak, --fileattestationkey `<file>` - `<file>:=ak.data` - File containing AK's public data encoded as ByteBuffer(TPM2B_PUBLIC);

- -fbm, --filebiosmeasurement `<file>` - `<file>:=/sys/kernel/security/tpm0/binary_bios_measurements` - File containing binary BIOS measurements log;
- -fim, --fileimameasurement `<file>` - `<file>:=/sys/kernel/security/ima/binary_runtime_measurements` - File containing binary IMA measurement log;
- -fcfg, --fileconfiguration `<file>` - `<file>:=configuration.json` - File containing configuration about PCRs to be attested.

[**Environment variables**]

- TPM_DATA_HOME - `TPM_DATA_HOME=/var/TPM_data` - Base directory for IBM TSS files.

Appendix B

Developer manual

This chapter presents the developer manual useful for getting details on protocols, serialization of objects and structures, and finally the development environment setup in order to better understand and reproduce the proposed implementation. First, the designed protocols will be presented and for each one the APIs and their purpose, as well as their requests and responses, will be described. All APIs use a JSON body to pass input parameters. For each API, we will specify the parameter name, the encoding used and a description of them. Many of the communication parameter are encoded as hexadecimal serialized ByteBuffer, which will be described in detail in section B.4.

For example, if we want to say that the parameter named "ipk" contains the group public keys data encoded in hexadecimal derived from a container ByteBuffer with two *G2_points* representing the X and Y points of the group public key, we write:

- **ipk**: ByteBuffer(G2_point, G2_point) - Group public keys G2_points X and Y.

B.1 Join protocol

The aim of the Join protocol is to agree with the client on a DAA credential, verify the candidate identity by its EK and allow the candidate to verify the Issuer identity through the group public key. An overview is shown in Figure B.1.

Figure B.1. Join protocol overview

The protocol is implemented with three endpoints:

- GET /public/groupdata
- POST /join/issuercredential
- POST /join/makefullcredential

GET /public/groupdata

It returns the public keys of the group managed by the Issuer. In the flow it is called by the Member_join application.

[**Request**]: empty.

[**Response**]:

```
{
 "ipk": "000200920002004600020020c824b17d4f4e845eebfdcaab..."
}
```

- **ipk**: ByteBuffer(G2_point, G2_point) - Group public keys G2_points X and Y.

POST /join/issuercredential

It takes as input the public data from EK and from candidate's DAA key and performs make_credential obtaining (cb_1, s_1), which is returned to the candidate. It then inserts the credential key K_1 used in make_credential in a database waiting for the candidate to later return it in /join/makefullcredential.

[**Request**]:

```
{
 "e": "ea25d5bade63c4096e396ab18e8d6dc337b9cb2ffd60662bf04906...",
 "q_pd": "005a0023000b000504720000010001a000b000100100010002..."
}
```

- **e**: ByteBuffer(TPM2B_PUBLIC) - Candidate's EK public data;
- **q_pd**: ByteBuffer(TPM2B_PUBLIC) - Candidate's DAA key's public data.

[**Response**]:

```
{
 "cb1": "0020db43715b6a7620e112e2362cdf717d3e67f5cbaf06f2a74b835...",
 "s1": "e507d4034ce26ac7dabdeb7343f9b38a79b5c1d0f635eb363e3fa1e8..."
}
```

- **cb1**: ByteBuffer(RSA2048 signature) - Value of cb_1 generated from make_credential procedure;
- **s1**: ByteBuffer(BIGNUM) - Value of s_1 generated from make_credential procedure.

POST /join/makefullcredential

It takes as input the signature (n_J, v, w) and the result from activate_credential K_1, checks both the signature and K_1 validity by searching in the database. It then performs another activate_credential obtaining (cb_2, s_2). It creates the DAA credential for the client and encrypts it with K_2 used in the new activate_credential. Finally, it returns (cb_2, s_2) and the credential.

[**Request**]:

```
{
 "k1": "a6df5ca78d0506de4cb1db5f6b0d4ba8",
 "nj": "e1a72d610396f428f563b273fc0d4d416f65d19d01d4c925874534cab55603fa",
 "v": "c3d6279b81b7fd90df127e72719b762e41085609da10de6ec24d6b8876a15045",
 "w": "0be74c09c59dad46d7197e1b389e7d37125a12561633d573ea099f9dc36a4690"
}
```

- **k1**: ByteBuffer(DIGEST_2B) - Credential key K_1 used in the first activate_credential;
- **nj**: ByteBuffer(ECC_PARAMETER_2B) - n_j returned by the TPM signing procedure;
- **v**: ByteBuffer(BIGNUM) - v returned by the TPM signing procedure.
- **w**: ByteBuffer(ECC_PARAMETER_2B) - w returned by the TPM signing procedure.

[**Response**]:

```
{
 "cb2": "0020517ae241397354f96e082f972c3f1820de83bfcf4b3043015e4...",
 "s2": "aa99097d3f3a3caebc70bd192fe3152cc72ac505e54fa1d594e39473...",
 "daa_cre": "047a31add8091d8fc0c4c143a5a411033a908e5d61e22bccfe7...",
}
```

- **cb2**: ByteBuffer(RSA2048 signature) - cb_2 generated from make_credential procedure;
- **s2**: ByteBuffer(BIGNUM) - s_2 generated from make_credential procedure;
- **daa_cre**: ByteBuffer(AES-128-CTR(Daa_credential)) - The DAA credential encrypted with AES-128-CTR using K_2 as key.

B.2 AK exchange protocol

The exchange of the AK is handled through the AK exchange protocol, as shown in Figure B.2, during which the Verifier checks, through the linking property of the DAA signature, whether the Member has not previously tried to register other AK. Afterwards, it verifies the group signature authenticity and associates the AK to the Member's pseudonym.

Figure B.2. AK exchange overview

GET /akRegister/publicData

It returns the bsn that characterizes the Verifier. The returned bsn will be used during AK attestation as signature bsn.

[**Request**]: empty.

[**Response**]:

```
{
 "bsn": "0002009200020046000200 20c824b17d4f4e845eebfdcaabc1eccef8afdc3ef2"
}
```

- **bsn**: ByteByffer(256 bytes random payload) - Verifier specific basename to be used in AK's certification signature.

POST /akRegister

It takes as input the randomized credential (R, S, T, W) used for the signature, the signature result $(bsn, J, K, h2, s, nc)$, AK public data (qps_pd) and the attestation (A). It checks that J is related to Verifier specific bsn, it evaluates the validity of the attestation relative to AK public data, thus it validates the signature. Moreover, it checks that K has never been previously used in any registration, to avoid the same Member to register two different AK. If the validation is successful, it registers in the database the AK together with K.

[**Request**]:

```
{
 "bsn": "000200920002004600020020c824b17d4f4e845eebfdcaabc1eccef8afdc3ef2",
 "certificate": "ff544347801700000000000000000002d17c000000000000000...",
 "daa_credential": "0004004600020020cdf9494862e52809b6929ec49c467ce8ba...",
 "h2": "fab85de6a7a50e8a82339b42a588476a0a1cd163b6384773330fc7ee5a517988",
 "ipks": "000200920002004600020020c824b17d4f4e845eebfdcaabc1eccef8afdc...",
 "label": "63726564656e7469616c2064617461",
 "nt": "39383af69bbe0f0c2c9ba0fb7de88467f6fa932b4042fb5c8317bf73a7d43240",
 "pt_j": "00020020f8ef5b8feb3b448e6e74b56eaea90d68db556b1eb7eae7e5597e...",
 "pt_k": "00020020f4d7ccb6263b1ba847dd0c2a73b0b2ec27d8e4faab2ac997ddb3...",
 "qps_pd": "00580023000b000504720000001000180000b0003001000209ba8db1278...",
 "sig_s": "94434fb252e6f423d321559cee5e9a03ca23f2c00fd49f9fbfdde679184f9fbc"
}
```


- **ipks**: ByteBuffer(G2_point, G2_point) - Group public keys G2_points X and Y;
- **bsn**: ByteBuffer(256 bytes random payload) - Basename selected by Verifier;
- **daa_credential**: ByteBuffer(Daa_credential) - Randomized DAA credential used for signature;
- **h2**: ByteBuffer(BIGNUM) - $h2$ signture's output parameter: $h2 = hash(nc \| h1)$;
- **label**: constant string = "credential data";
- **pt_j**: ByteBuffer(G1_point) - Point J, signature output, depend on bsn;
- **pt_k**: ByteBuffer(G1_point) - Point K, signature output, depend on point J and DAA private key;
- **qps_pd**: ByteBuffer(TPM2B_PUBLIC) - AK's public data;
- **nt**: ByteBuffer(ECC_PARAMETER_2B) - nc random value generated during signature;
- **sig_s**: ByteBuffer(TPM2B_ECC_PARAMETER) - s result of signature process;
- **certificate**: ByteBuffer(TPMS_ATTEST) - AK key's certificate (attestation).

[**Response**]: "true" in case of success.

B.3 Push measurement protocol

Though the Push measurement protocol, a Member sends to the Verifier the BIOS and IMA measurements together with the involved PCR registers quotes. The Verifier checks the validity of the used AK, the quote signature and, finally, the measurements validity relative to the values received as input from the Member. If

a validation fails, it is logged in WARN in a Log file relative to the measurement. An overview of the protocol is shown in Figure B.3.

Figure B.3. Push measurement protocol overview

POST /measurement/challenge

It takes as input the public data of the AK with which the measurement is to be made. It returns a string to be used by the attester as optional data in the quote it will carry out. The challenge and its association with the AK are stored in the database so that they can be verified later.

[**Request**]:

```
1 {
2  "ak_pd": "00580023000b000504720000001000180000b0003001000209ba8db1278..."
3 }
```

- **ak_pd**: ByteBuffer(TPM2B_PUBLIC) - AK's public data.

[**Response**]:

```
1 {
2  "challenge":
       "83310137672cadec01cdc880f6978f70d6bcf08131757c6814abd8ac6c0d909d"
3 }
```

- **challenge**: ByteBuffer(256bytes random payload) - Value to be added as extra parameter in the Quote attestation.

POST /measurement

It takes as input the randomized credential (R, S, T, W) used for the signature, the signature result $(bsn, h2, s, nc)$, AK public data (qps_pd) and the attestation (A). It also retrieves BIOS and IMA measurements lists.

[**Request**]:

```
{
 "ak_pd": "00580023000b000504720000001000180000b0003001000209ba8db1278...",
 "attestation": "ff54434780180022000bc944a5df23e39231b002ec7ef569f0c...",
 "challenge":
     "39e815c61efb79efe01483bd47a601c5c5151ef672b66c29ce9bbfe7e0b7c7ec",
 "h2": "91f20b0a7fc12564b7fdad2f498a70c07274e9b49b72b28f29bf90f9fc46efff",
 "nt": "870e3e9a474daf5a3a71f41e195c4a518dd81309a760ada87f3b463fcdc20a63",
 "sig_s": "560b8f325bb1596c2f50e7f69745544bb7baa357b8e46e08122f693988dd12f6",

 "bios_measurement_events": [
     "00000000000000030000000000000000000000000000000000000000000000000...",
     "00000001000000060000001000b6918630adf88621acc953b4b5869e653d0...",
     "00000002000000050000001000b9dbd87163112e5670378abe4510491259a...",
     "00000002000000060000001000bac08c54b65d8474bfa9dce0ca99f9886dc...",
     "00000002000000060000001000bceff39f4eda92a15d96dd54f2236b44c61...",
     "..."
 ],
 "ima_measurement_events": [
     "0000000ab1cd730ee01e7137e7881fcb42fd8462c5c8de1900000006696d61...",
     "0000000a0b5a1ee5112b581fc74040146ce5bdb607c1e1ac00000006696d61...",
     "0000000adbc038c03d50fed2ac6f7f8eb210f7e670d1c72000000006696d61...",
     "0000000ab6a0d6a27247a1af40cd7101130c688f771425130000000006696d61...",
     "0000000afa5df98ea572ec24c06f0ef9452a9ab23a1a41a700000006696d61...",
     "0000000ab0594fefde8e4981dfded3099cf0c45968e64c8800000006696d61...",
     "0000000a0f8b2256c6159adfec34fc26b0f70f62b189e9a600000006696d61...",
    "..."
 ]
}
```


- **ak_pd**: ByteBuffer(TPM2B_PUBLIC) - AK's public data;
- **attestation**: ByteBuffer(TPMS_ATTEST) - PCR's Quote attestation;
- **challenge**: ByteBuffer(256bytes random payload) - Given by verifier in the /measurement/challenge;
- **h2**: ByteBuffer(BIGNUM) - $h2$ signture's output parameter: $h2 = hash(nc \| h1)$;
- **nt**: ByteBuffer(ECC_PARAMETER_2B) - nc random value generated during signature;
- **sig_s**: ByteBuffer(TPM2B_ECC_PARAMETER) - s result of signature process;
- **bios_measurement_events**: ByteBuffer(TCG_PCR_EVENT) - Binary values of BIOS measurement events;
- **ima_measurement_events**: ByteBuffer(ImaEvent) - Binary values of IMA measurement events.

[**Response**]: true

B.4 ByteBuffer

ByteBuffer is a fundamental class acting as generic dynamic array of bytes, used as data structure in all operations on binary data such binary operations, hash, signatures, etc. In addition, it is used as a conversion class to and from other object types and as a standard for encoding and decoding objects.

ByteBuffer serialization

The first use of the ByteBuffer is as a tool to support serialization of objects to allow persistence on files or to use them on communication channels. Given a ByteBuffer as input, serializing it means creating a new ByteBuffer with 2 extra bytes that are used as a header to specify the size of the payload, that is, the length of the input to be serialized. The two bytes of the header are calculated as:

$$firstByte = payloadSize/256$$
$$secondByte = payloadSize\%256$$

In this way, it is possible to re-caluclate the payloadSize during the deserialization phase as:

$$payloadSize = firstByte * 256 + secondByte.$$

For instance, if receiving as input a ByteBuffer of 800 bytes payload, its serialization would be a new ByteBuffer of 802 bytes, whose first bytes, i.e. the header, would be equal to:

$$firstByte = 800/256 = 3$$
$$secondByte = 800\%256 = 32$$

The remaining 800 bytes are equivalent to the ones of the ByteBuffer in input. The ByteBuffer thus created is ready to be persisted or sent as a communication DTO since it has the information to be reconstructed later by deserialization.

ByteBuffer as container

The second important use of the ByteBuffer is as a container for several, and even diverse, objects. Indeed, through the `serialise_byte_buffers` function it is possible to take multiple ByteBuffers, serialize them one by one, and merge them into a new ByteBuffer, thus allowing to serialize complex objects. The resulting ByteBuffer is composed by a payload made up of the concatenation of the serialization of the input ByteBuffers plus a header composed of 2 bytes. The header bytes are calculated from the number of ByteBuffer to be serialized (numBBs) as follows:

$$firstByte = numBBs/256$$
$$secondByte = numBBs\%256$$

In this way, when deserializing, it is possible to go back to numBBs computing:

$$numBBs = firstByte * 256 + secondByte$$

For example, if taking as input 530 different ByteBuffers, the output will be made up by a header having:

$$firstByte = 530/256 = 2$$
$$secondByte = 530\%256 = 18$$

The payload, instead, is the concatenation of the serialization of the single ByteBuffers in input.

Hex_string

Hex_string allows to assess whether a string in input is codified in hexadecimal and to normalize it to have an even number of characters. Hex_string is easily converted to ByteBuffer and vice versa allowing a simple transformation of ByteBuffers into hexadecimal strings. All complex objects requiring to be saved on file or sent to the network are converted to ByteBuffer and then codified into hexadecimal through Hex_string.

B.4.1 ByteBuffer conversion utils

A number of utility classes have been created to simplify and centralize the conversion of the various object types to ByteBuffer and vice versa. We do below some examples that may help new developers better master the tool.

The serialization of a generic C/C++ struct is simply derived from the serialization of the entire memory section composing the struct. For example a widely used structs are those regarding TSS like *TPM2B_PUBLIC* which is defined in the TPM specification and describes the public data of a generic TPM object. Another is the *ImaEvent* that is defined in TSS utils library and allows to easily manipulate the IMA binary measurement events through several utilities from the library. Concerning the objects related to IBM TSS, a function called `TSS_Structure_Marshal()` takes as input the TSS struct and convert in standard manner to an output buffer easily converted and serialized with the ByteBuffer. Another example are the BIG and BIGNUM structs, representing BigNumber respectively of AMCL and OpenSSL. These have utility functions for conversion to ByteBuffer, `big_to_bb` and `bn2bb` respectively.

Some utilities define the serialization process of objects used as cryptographic parameters and as points of elliptic curves. Lets have a look as an example at the serialization of *G1_point* and *G2_point*. A G1_point consists of a pair of values (x, y) indicating coordinates on a curve, each 32 bytes long. Serialization can be performed with `g1_point_serialise` which is equivalent to `serialise_byte_buffers(x,y)` of its coordinates. The *Daa_credential* is the composition of four

G1_point representing the four points of the DAA credential. A G2_point, on the other hand, is composed of a couple of coordinates, *G2_coord*, which in turn is composed of 2 values (x, y), each of 32 bytes. Serialization is obtained through `g2_point_serialise` which is actually `serialise_byte_buffers(serialise_byte_buffers(x1,y1),serialise_byte_buffers(x2,y2))` where $(x1, y1)$ e $(x2, y2)$ are the values of the first and second coordinate of the two G2_points, respectively.

B.5 Setup environment

In this section it is shown how to configure the development environment in order to properly compile and run the applications. We begin by configuring the TPM simulator and compiling the TSS libraries, then compiling the support libraries, and finally the thesis project applications.

ibmswtpm2

Download `ibmtpm1661.tar.gz` from https://sourceforge.net/projects/ibmswtpm2/ and extract it in `/opt/ibmtpm1661`. Then compile with:

```
1  cd /opt/ibmtpm1661/src
2  make
```

create a symbolic link in `/opt/ibmtpm`:

```
1  cd /opt
2  ln s ibmtpm1661 ibmtpm
3  chmod 755 -R /opt/ibmtpm
```

(Follow `ibmtpm.doc` for more compilation details)

ibmtss

Download `ibmtss1.6.0.tar.gz` from https://sourceforge.net/projects/ibmtpm20tss/ and extract it in `/opt/ibmss1.6.0`. Then compile it with:

```
1  /configure
2  make
3  make install
```

and create a symbolic link in `/opt/ibmtss`:

```
1  cd /opt
2  ln s ibmss1.6.0 ibmtss
3  chmod 755 -R /opt/ibmtss
```

(Follow `ibmtss.doc` for more compilation details)

To launch the TPM simulator run:

```
1  sudo /opt/ibmtpm/src/tpm\_server
```

finally, to launch all tests and verify operation, run:

```
1 /opt/ibmtss/utils/reg.sh -a
```

libhttpserver

To configure *libhttpserver* it is necessary to clone the project from the repository https://github.com/etr/libhttpserver, configure, compile and install it:

```
1 git clone https://github.com/etr/libhttpserver
2 cd libhttpserver
3 ./configure
4 make
5 make install
```

thesis applications

Each application has its own directory in this path: `/Daa_code/Daa_tpm/Tpm_experiments`. The environment variable must be set to allow linking to *ibtss utils* library:

```
1 export LD_LIBRARY_PATH=/opt/ibmtss/utils
```

To compile any application, in the path of the specific application, just do:

```
1 make
```

Bibliography

[1] E. Brickell, L. Chen, and J. Li, "A new direct anonymous attestation scheme from bilinear maps", Trusted Computing - Challenges and Applications (P. Lipp, A.-R. Sadeghi, and K.-M. Koch, eds.), Berlin, Heidelberg, 2008, pp. 166–178, DOI 10.1007/978-3-540-68979-9_13

[2] S. Wesemeyer, C. J. Newton, H. Treharne, L. Chen, R. Sasse, and J. Whitefield, "Formal analysis and implementation of a tpm 2.0-based direct anonymous attestation scheme", Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, New York, NY, USA, 10 2020, p. 784–798, DOI 10.1145/3320269.3372197

[3] ISO Central Secretary, "Information technology – trusted platform module library", Standard ISO/IEC, 11889:2015, International Organization for Standardization, 2015

[4] W. Arthur, "A practical guide to tpm 2.0", 2015, Hinweis: Open-Access-Publikation

[5] Trusted Computing Group, "TCG TSS 2.0 Overview and Common Structures Specification." https://trustedcomputinggroup.org/resource/tss-overview-common-structures-specification/, Accessed: 2022-06-22

[6] "Integrity Measurement Architecture (IMA)." https://sourceforge.net/p/linux-ima/wiki/Home/, Accessed: 2022-06-22

[7] "SDB:Ima evm." https://en.opensuse.org/SDB:Ima_evm, Accessed: 2022-06-22

[8] "IBM TPM Attestation Client Server." https://sourceforge.net/projects/ibmtpm20acs/, Accessed: 2022-06-22

[9] ISO Central Secretary, "Information technology – security techniques – anonymous digital signatures – part 1: General", Standard ISO/IEC, 20008-1, International Organization for Standardization, 2013

[10] ISO Central Secretary, "Information technology – security techniques – anonymous digital signatures – part 2: Mechanisms using a group public key", Standard ISO/IEC, 20008-2, International Organization for Standardization, 2013

[11] N. R. Luther, "Implementing direct anonymous attestation on tpm 2.0", Master's thesis, 2017. https://vtechworks.lib.vt.edu/handle/10919/86349

[12] J. Camenisch, M. Drijvers, and A. Lehmann, "Universally composable direct anonymous attestation." Cryptology ePrint Archive, Paper 2015/1246, 2015, https://eprint.iacr.org/2015/1246

[13] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation", Proceedings of the 11th ACM Conference on Computer and Communications Security, New York, NY, USA, 2004, p. 132–145, DOI 10.1145/1030083.1030103

[14] J. Camenisch, L. Chen, M. Drijvers, A. Lehmann, D. Novick, and R. Urian, "One tpm to bind them all: Fixing tpm 2.0 for provably secure anonymous attestation", 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 901–920, DOI 10.1109/SP.2017.22

[15] K. Yang, L. Chen, Z. Zhang, C. J. P. Newton, B. Yang, and L. Xi, "Direct anonymous attestation with optimal tpm signing efficiency", IEEE Transactions on Information Forensics and Security, vol. 16, 2021, pp. 2260–2275, DOI 10.1109/TIFS.2021.3051801

[16] J. Whitefield, L. Chen, T. Giannetsos, S. Schneider, and H. Treharne, "Privacy-enhanced capabilities for vanets using direct anonymous attestation", Turin, Italy, 2017, pp. 123–130, DOI 10.1109/VNC.2017.8275615

[17] FIDO Alliances, "FIDO alliance universal authentication framework (UAF) 1.1 specifications." https://fidoalliance.org/download/, 2017, Accessed: 2022-06-22

[18] U. Greveler, B. Justus, and D. Loehr, "Direct anonymous attestation: Enhancing cloud service user privacy", On the Move to Meaningful Internet Systems: OTM 2011 (R. Meersman, T. Dillon, P. Herrero, A. Kumar, M. Reichert, L. Qing, B.-C. Ooi, E. Damiani, D. C. Schmidt, J. White, M. Hauswirth, P. Hitzler, and M. Mohania, eds.), Berlin, Heidelberg, 2011, pp. 577–587, DOI 10.1007/978-3-642-25106-1_11

[19] ISO Central Secretary, "Information technology – security techniques – key management – part 4: Mechanisms based on weak secrets", Standard ISO/IEC, 11770-4:2017, International Organization for Standardization, 2017

[20] J. Whitefield, L. Chen, R. Sasse, S. A. Schneider, H. Treharne, and S. Wesemeyer, "A symbolic analysis of ecc-based direct anonymous attestation", IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019, 2019, pp. 127–141, DOI 10.1109/EuroSP.2019.00019

[21] R. Barbulescu and S. Duquesne, "Updating key size estimations for pairings." Cryptology ePrint Archive, Paper 2017/334, 2017, DOI 10.1007/s00145-018-9280-5, https://eprint.iacr.org/2017/334

[22] "The MIRACL Core Cryptographic Library." https://github.com/miracl/core, Accessed: 2022-06-22

[23] T. Acar, L. Nguyen, and G. Zaverucha, "A tpm diffie-hellman oracle." Cryptology ePrint Archive, Paper 2013/667, 2013, https://eprint.iacr.org/2013/667

[24] "ecc-daa Github repository." https://github.com/UoS-SCCS/ecc-daa, Accessed: 2022-06-22

[25] "ibmswtpm2." https://ibmswtpm.sourceforge.net/ibmswtpm2.html, Accessed: 2022-06-22

[26] "IBM's TPM 2.0 TSS." https://sourceforge.net/projects/ibmtpm20tss/, Accessed: 2022-06-22

[27] "OpenSSL." https://www.openssl.org/, Accessed: 2022-06-22

[28] "libhttpserver Github repository." https://github.com/etr/libhttpserver, Accessed: 2022-06-22

[29] "cpp-httplib." https://github.com/yhirose/cpp-httplib, Accessed: 2022-06-22

[30] "Proxmox." https://www.proxmox.com/en/, Accessed: 2022-06-22

[31] "Infineon Technologies AG. 2017. Iridium SLB 9670 TPM2.0 Linux." https://www.infineon.com/cms/en/product/evaluation-boards/iridium9670-tpm2.0-linux/, Accessed: 2022-11-01