POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Distributed Trusted Computing Base

Supervisor
prof. Antonio Lioy

Candidate
Davide Scovotto

Tutors
LINKS FOUNDATION
Andrea Vesco, Ph.D.

Academic year 2022-2023

Summary

The widespread adoption of Distributed Ledger technologies (DLTs), specifically blockchain, has generated great interest in the economic and financial sectors mainly due to its decentralized approach. An implementation of DLT is the blockchain, which is the technology created for the Bitcoin cryptocurrency. The blockchain is the underlying structure that enables the exchange of cryptocurrency, through transactions among different peers in a completely decentralized manner. Due to its nature, the DLT provides advantages such as implicit integrity and increased transparency of stored information. However, to exploit such benefits, an existing Trust relationship among the transacting peers is required. Trusted Computing can leverage the advantages provided by DLTs and combine them with the concepts of hardware Root-of-Trust (RoT), as well as Trusted Computing Base (TCB). Moreover, Remote Attestation (RA) protocol can be appropriately improved with the aid of the DLT properties to verify the integrity of a TCB belonging to a device. Conversely from common RA protocols involving centralized entities, the adoption of DLT in such a context, enables a decentralized attestation model. This work of thesis aims to build a completely decentralized model that allows a group of peers, interacting through a DLT, to establish trust without the intervention of any central entity. This work focuses on the design of a custom RA protocol, which leverages a TPM and the IMA kernel module, to implement a model for Distributed Trusted Computing Base (DTCB). It leverages the IOTA DAG-based DLT, called Tangle, as a secure means for storing and exchanging information. The RA protocol features also a distributed group-consensus protocol. This allows a set of nodes to maintain a distributed state of trust among themselves and also to detect and exclude non-trusted peers from the group. This work details the implementation of a DTCB by building a Proof of Concept (PoC), whose implementation has been deployed and tested to a set of constrained devices (e.g. RaspberryPi), demonstrating that the nodes participating in the formation of the DTCB can autonomously maintain a distributed state of trust.

Acknowledgements

I would like to thank Professor Antonio Lioy for his supervision. I sincerely thank Ing. Andrea Vesco, Ing. Alberto Carelli and Ing. Davide Margaria, the LINKS Foundation Cybersecurity team, for their precious and continuous support during the development of this work.

I thank my family, who sustained me in all the difficulties that have arisen during my university journey. I especially thank my father and my mother, without whom I would not have had the opportunity to live this academic experience. I would like to give a special thanks to Francesca, who always stood next to me giving me encouragement and love during these years. Finally, I thank all my friends, that have helped me throughout this journey.

Contents

Chapter 1

Introduction

Trusted computing has become more and more important over the years due to the increasing number of computer system owners, which are constantly exposed to severe threats. Trusted Computing refers to those computer systems that can identify their hardware and software components to establish trust between themselves and an external entity that wants to verify if the computer system is behaving as expected. The Trusted Computing Group [1] provides standard-based Trusted Computing specifications, solutions and technologies that enable a reporting mechanism about the internal state of a device that can be used to state whether the device is in a trusted state. One of the major Trusted Computing technologies developed by the TCG is the Trusted Platform Module (TPM). It is a hardware module that can be embedded into end systems like PCs, or even IoT devices, and acts as a "Root of Trust (RoT)" for the platform. The TPM offers the possibility to store cryptographic hash values, called measurements, inside the TPM's Platform Configuration Registers (PCRs). Those measurements describe the software state and configuration of the system, and when digitally signed with a valid key, an Attestation report is generated. The key used to sign a set of PCRs is often called Attestation Key (AK) and has to reside inside the platform's TPM. A third party (Verifier) can then request an attestation report to a given system (Attester) and verify the integrity of the digital signature and also the trustworthiness of the received measurements by comparing them with a database of known values, often called golden values; then, based on the verifier's policies, conclusions can be made about the level of trust of the system under inspection.

Those generally described operations give birth to a new security service called *Remote Attestation*. Remote Attestation is often used in conjunction with Trusted Computing Base (TCB) to provide an additional layer of security. The TCB can be used to generate the attestation report, and the other party can use the attestation report to verify that the TCB is functioning correctly and has not been compromised. In TCG's terms, Verifiers can use Remote Attestation to check whether the underlying Trusted Computing Base of the targeted device and the data arriving from the TCB are trustworthy. The TCB is typically made up of low-level components such as the operating system kernel, device drivers, and firmware. These components are responsible for managing the system's resources, including access to hardware and software, and enforcing security policies such as access controls and permissions.

In recent years, many Remote Attestation protocols have been implemented, but most rely on nodes having to put trust in centralized verifiers. However, today there is a growing interest, especially in the financial sector, in managing various types of applications using decentralized technologies like the blockchain or, more generally, Distributed Ledger Technologies (DLTs). DLTs offer the creation and the management of an **immutable and append-only** distributed storage, not allowing the deletion or the update of any information once stored on the DLT. These technologies introduce the possibility to build real-world applications in a completely decentralized manner, which means that there is no more need for central authorities to act as trusted intermediaries between two or more peers. Hence, DLTs are **trustless** systems where all their participants rely on a single source of truth, the ledger state. Active nodes on the distributed network execute a consensus mechanism allowing them to agree on a single source of truth, without having to put trust in third-party authorities. However, computer devices can be vulnerable and if attacked can lie at any time; in such cases about the state of the ledger. When a set of distributed nodes is unknowingly compromised, the consensus outcome could not reflect the true state of the ledger, thus compromising the whole network and making possible the approval of malicious transactions. The just described scenario is just an example, but the same issue can be applied to multiple real-world distributed applications, hence it is important to ensure that nodes in a distributed system are behaving as expected, possibly without the need of putting trust in central entities.

This works aims at answering the above issues by extending the concepts of Trusted Computing, especially of TCB and Remote Attestation, to work in decentralized applications in such a way that peers, or devices, can trust each other without needing the support of external trusted entities. Hence, this work focuses on providing a DLT-based solution that enables a set of devices to **autonomously** maintain a distributed state of trust among them. The design of a Distributed Trusted Computing Base (DTCB) model, together with the implementation of a Proof of Concept, is proposed to show how a set of devices can autonomously monitor and evaluate the trust status of all the DTCB participants by leveraging a mutual Remote Attestation protocol utilizing a DLT to store and exchange the protocol's messages.

In this document, the core concepts and technologies of Trusted Computing are introduced, such as TCB and Remote Attestation, as well as an overview of the core aspects and functionalities offered by DLTs. The chosen DLT properties and the communication library used for interfacing with the DLT are as well presented. Finally, a DTCB model is presented, highlighting the properties and the requirements for a DTCB to be practical, accompanied by an overview of the implemented Proof of Concept.

Chapter 2

Trusted Computing Base

In December 1985 the U.S. Department of Defense (DoD) released an important book that marked a turning point for Trusted Computing: the Trusted Computer System Evaluation Criteria (TCSEC)[2], which defined the concept of Trusted Computing Base (TCB):

"The totality of protection mechanisms within a computer system – including hardware, firmware, and software – the combination of which is responsible for enforcing a security policy."

The TCB concept is important in trusted computing, as it helps to identify the components that are responsible for enforcing security and allows for technical innovations to be focused on these components. The TCB also serves as a reference point for subsequent discussions of trusted computing and security policy, with their impact and relevance being evaluated in relation to the TCB. This concept became particularly influential in the field of trusted computing over the past two decades. When creating and executing a Trusted Computing Base, several important factors must be taken into account. One crucial element to consider is reducing the size of the TCB as much as feasible since a smaller TCB is usually considered more secure. Additionally, it is necessary to make sure the TCB is designed to resist attacks and can recover from potential failures or attacks without compromising the security of the system. For a system's TCB to meet the trust requirements set by the TCG, it must fulfill the following TCB properties [3]:

1. The functions executed by the TCB must not compromise its integrity and must be computationally limited to prevent excessive resource utilization.

2. For a function implementation to be effective in a TCB, it must be able to run uninterrupted and not be impacted by any restrictions (such as unavailable resources) or influenced in any manner.

3. TCB instances must be unique and distinguishable from one another. Cryptographic identity confirms that each TCB node has a unique identity, which it can prove to other devices.

4. The TCB should have a flexible architecture that allows for the integration of new layers (or removal of existing ones) to respond to evolving hardware and computational demands.

However, while the TCSEC criteria primarily focused on defining the security domain of operating systems, it is important to recognize that the hardware of a computing platform also plays a significant role in the TCB. Hardware components contribute to the system's overall security, but from the operating system's (OS) point of view, the hardware is considered trusted because the OS has no way to verify that the underlying hardware is behaving correctly. However, this does not mean that hardware is invulnerable to compromise. The risk of hardware vulnerabilities prompted the formation of the Trusted Computing Group (TCG)[1], which focused on the concept of a hardware *Root-of-Trust* (RoT) to identify the security-relevant parts of a hardware platform. A hardware RoT should be able to check its integrity and then establish trust by confirming the authenticity of the operating environment, including the boot software, operating system, and other software and hardware components within the system.

A specific hardware component that implements this concept is the Trusted Platform Module (TPM), which is defined by the specifications of the TCG. These specifications are based on the concepts of Trust and Trusted Platforms (TP). A TP is a system that is designed to have both hardware and software capabilities that allow it to collect and provide integrity measurements to third parties. These measurements can be used to remotely determine if the TP is behaving as expected, a process known as *Remote Attestation* (RA). A Linux-based TP has the option to enable a kernel module called *Integrity Measurement Architecture* (IMA), which allows for dynamic integrity measurements of all executables, kernel modules, and configuration files loaded at runtime.

2.1 Trust and Trusted Platforms

The concept of "Trust" has been interpreted in various ways in the past years, but in the context of the TCG specifications, trust is conveyed to represent a system that behaves as expected. Trust in a platform is based on the concept of *Roots of Trust* (RoTs), which are system components that are secure by design, cannot be compromised, and any misbehavior cannot be detected during runtime, as defined by TCG[4].

2.1.1 Roots of Trust

The TCG specifies a minimum set of RoTs required to ensure the trustworthiness of a platform. These RoTs are defined based on the characteristics that are essential for platform security. The TCG's specifications list the following minimum RoTs required in a TP:

- Root of Trust for Storage (RTS)
- Root of Trust for Measurement (RTM)
- Root of Trust for Reporting (RTR)

The RTS is responsible for securely storing critical system information, such as encryption keys and platform configuration data, in memory locations known as *Shielded Locations*. These locations can only be accessed externally through dedicated commands known as *Protected Capabilities*. Shielded Locations can store sensitive information, like the private part of asymmetric keys, which are protected from unauthorized access. Other memory locations, known as Platform Configuration Registers (PCRs), are used to store integrity measurements of the platform components. These integrity measurements are calculated by applying a cryptographic hash function on the software and configurations of the platform, and their values can only be modified through reset or Extend operations.

The RTM establishes a secure foundation for measuring the system's configuration, including the firmware and software that is running on it. The RTM relies on a fixed piece of code called the Core Root of Trust for Measurement (CRTM). The CRTM is the foundation for measurement and sets the initial measurements of the platform. These initial measurements are then sent to the RTS which stores them in a Shielded Location, the PCRs, by leveraging the Extend operation. To ensure that measurements are accurate, the code being executed must have control over the environment in which it is running so that the values recorded by the RTS reflect the initial trust state of the platform. When a power-on reset is performed, the platform is returned to a known initial state, with the main CPU executing code from a specific location. This code has full control over the platform, allowing it to take measurements of the firmware. These initial measurements can be used to establish a chain of trust. This chain of trust is created only once when the platform is reset and cannot be altered, so it is referred to as a static Root of Trust for Measurement (S-RTM). An alternate approach to initializing the platform, available on certain processor architectures, is to allow the CPU to function as the CRTM and enforce protections on specific areas of memory that it measures. This process enables the creation of a new chain of trust without requiring a reboot of the platform. Since the Root of Trust for Measurement can be established dynamically, this method is referred to as the dynamic Root of Trust for Measurement (D-RTM). Both S-RTM and D-RTM can bring a system from an unknown state to a known state. D-RTM has the benefit of not needing to reboot the system.

The RTR is responsible for constructing reports about the contents of the RTS. An RTR report typically consists of a digital signature on a digest of selected values that reside in the RTS. It is important to remember that not all Shielded Locations are directly accessible, and as a result, the RTR report will not contain sensitive information such as the private part of keys.

2.2 The Trusted Platform Module

A Trusted Platform Module (TPM) is a specialized microcontroller integrated into a computing device that provides hardware-based security features. The TPM v1.1b was released back in 2003 by the TCG. The TPM specification has undergone two major revisions. The first generation, TPM 1.1b to 1.2, gradually added new capabilities as they were identified by the specification committee, resulting in a complex final specification. In response to cryptographic weaknesses in SHA-1,

TPM 2.0 was completely redesigned, resulting in a more streamlined and cohesive design [5]. In this work, the TCG specifications being referred to are those of the "Family 2.0", thus the TPM version 2.0 will be used as the standard of reference.

A TPM must provide RTS and RTR functionality, which is achieved through the inclusion of various functional components such as:

- PCRs that enable recording the platform state
- A mechanism for reporting platform state to external parties
- Secure non-volatile and volatile storage
- Hardware random number generator
- Symmetric and asymmetric key generation and management
- Encryption and decryption capabilities
- Hashing capabilities
- Digital signature creation and verification

The TPM is designed to allow for the creation of new security services that build upon its capabilities, thereby increasing the security level of existing software-based solutions. A prime example of this is Remote Attestation (RA): TPM-based RA allows a remote entity to cryptographically verify the trustworthiness of the platform's state; whereas software-based RA does not allow a remote entity to accurately determine the state of the platform because compromised software can easily lie and provide false information to the remote party.

Before going into deeper details about the process of RA, let us first give a brief presentation about the TPM's internal architecture, as well as all the key functionalities that are needed to better understand how a RA Protocol can be designed and implemented.

2.2.1 Architecture

In figure 2.1 the major elements of the TPM architecture are shown. To ensure that TPM hardware components are built correctly, manufacturers must adhere to the specifications set forth by the TCG. Let's take a closer look at the various components of a TPM's architecture and their respective roles:

- I/O buffer: it serves as the intermediary between a TPM and the host system, allowing for communication to take place. The host system writes command data to the I/O buffer, and then retrieves response data from the buffer.
- RNG: it is a protected capability without any access control, and it is used by the TPM as the source of randomness. The RNG is used for nonces, key generation, and signatures.

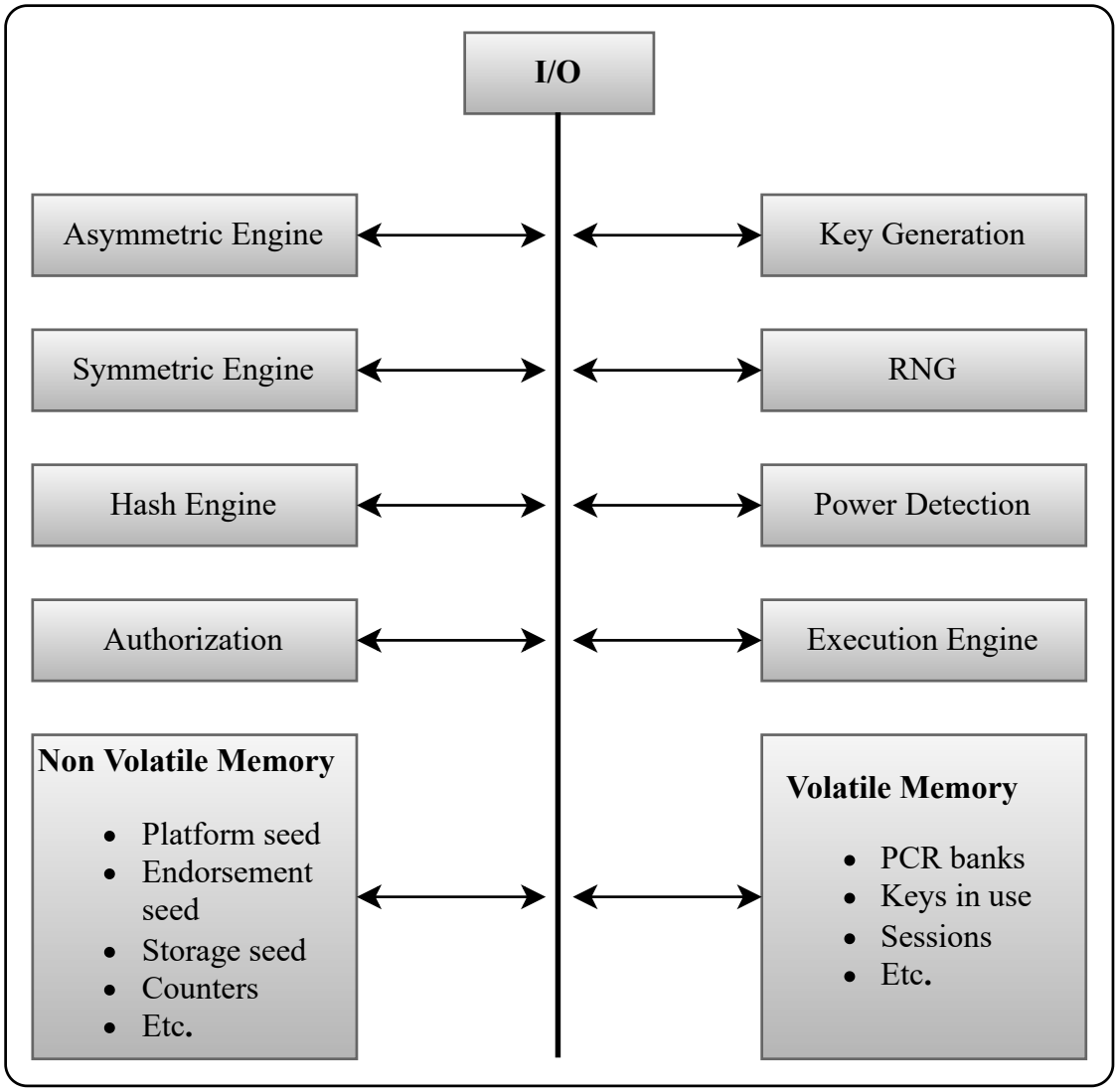

Figure 2.1. TPM v2.0 internal architecture [4]

- Key generation: it allows the creation of two types of keys. The first, a standard key, is generated using the RNG as the seed for the calculation. The output of the calculation is a secret key value stored in a Shielded Location. The second type of key, called a Primary Key, is created using a seed value, not the RNG directly. The seed is typically generated by the RNG and is permanently stored on the TPM.

- Hash engine: it provides the set of SHA family hash functions that can be utilized independently by external programs or as a result of various TPM actions. The TPM employs hashing for integrity verification, authentication, and one-way functions, such as KDFs, as necessary.

- Symmetric and asymmetric engines: they both provide the corresponding functionalities needed for encryption/decryption and for signing and signature

verification.

- Authorization: before a command can be carried out, the Authorization Subsystem verifies that the necessary permissions have been granted to access each Shielded Location.
- Power detection: it manages the power states of the TPM, taking also into consideration the power state of the hosting platform.
- Volatile memory (RAM): it is responsible to store TPM transient data. Data in TPM RAM may be lost when the TPM power is removed. It stores the PCR banks, the active sessions, and loadable entities, like keys in use.
- Non-volatile memory: it provides a way to store persistent objects, which remain intact even when power is removed, through the use of Shielded Locations that can only be accessed using Protected Capabilities. Some of this memory is reserved for the storage of primary seeds, which are provided by TPM vendors, and potentially an Endorsement Key (EK). Some space is also left to let the TPM user define and store objects, like keys, persistently.

2.2.2 Hierarchies

The TCG uses the concept of *hierarchy* to represent a collection of entities that are related and managed as a group[5]. A hierarchy can be persistent or volatile, allowing the TPM users to choose whether an entity has to be retained after a reboot or not. The TPM 2.0 specifications enhance this concept by including three persistent hierarchies and one volatile hierarchy:

- Platform hierarchy
- Storage hierarchy
- Endorsement hierarchy
- NULL hierarchy (volatile)

Each persistent hierarchy has been designed to perform specific tasks in different use case contexts, but they also share some common properties:

- An authorization value and a policy. These authorization controls are independently managed between the three hierarchies, allowing to have separate administrators for each one of them. This allows for a more granular level of security management, where different levels of access and control can be granted to different users or roles within the organization.
- Each one has an enable/disable flag. Disabling one hierarchy does not affect the other hierarchies.

- Each hierarchy has a different cryptographic root, the *seed*. A seed is a large, randomly generated number that is created by the TPM or injected by the TPM manufacturers and is never shared or exposed outside the TPM's secure boundaries. The seed is persistent within its hierarchy and can be used to derive data objects and keys.
- Each hierarchy can store a tree of keys, where the parent key is a key created starting from the seed of the corresponding hierarchy, called the primary key. Starting from the primary key of a hierarchy, descendant keys can be created.

Despite the TPM having a restricted amount of NV memory, it is capable of creating multiple keys, thanks to the capability of recreating the same keys by using a cryptographic algorithm on a given hierarchy seed, as it is deterministic. As a result, the keys do not have to be saved in the file system, as they can always be regenerated. Thus, users can easily create hierarchies of keys for their specific needs.

Platform Hierarchy

The *platform hierarchy* is intended to be under the control of the platform manufacturers and provides a secure location for them to keep objects related to the platform, including cryptographic material that safeguards the update process of the CRTM (the early boot code, the BIOS). The seed located in this hierarchy, known as the Platform Primary Seed (PPS), is generated by the TPM every time it is turned on and no PPS is present or it can be injected only by the TPM manufacturer.

Storage Hierarchy

The *storage hierarchy* facilitates the secure storage of objects that are relevant to TPM users, including the encryption of any data using the root key of the storage hierarchy (SRK) and the safekeeping of encryption keys in the non-volatile memory of the TPM. As for the platform hierarchy, the Storage Primary Seed (SPS) is created whenever the TPM is powered on and no SPS is present.

Endorsement Hierarchy

The *endorsement hierarchy* is the hierarchy that is privacy sensitive and is the preferred option when the user is concerned about privacy. The seed belonging to this hierarchy takes the name of the Endorsement Primary Seed (EPS). The EPS forms the foundation of the RTR's identity and is used to create an asymmetric key, the Endorsement Keys (EKs)[4]. To ensure that the primary keys in this hierarchy are only associated with an authentic TPM that is connected to an authentic platform, manufacturers inject the EPS before the device is shipped and then, optionally, generate one or more EKs and store them in the TPM. A Certification Authority (CA) can then issue a digital certificate for the generated public part of the EK;

the EK certificate. Generating and certifying the EK enables a correlation to be made back to a TPM, so that remote or local parties are ensured that received data are coming from a single and authentic TPM.

NULL Hierarchy

Unlike persistent hierarchies, the *NULL hierarchy* is the only volatile hierarchy available on a TPM. The NULL seed is a random value that is set on every TPM reset (analogous to a platform's reboot). It can be used to generate primary objects and also children of primary keys. However, unlike persistent hierarchies, objects in this hierarchy cannot be made persistent and their existence is limited to the next TPM reset, as the seed value changes on this event.

2.2.3 Endorsement and Attestation Keys

Due to the properties of the endorsement hierarchy, it is the preferred place where to store keys or objects that have to be used for privacy-sensitive processes; such as Remote Attestation. As mentioned in the previous section, the EK is the primary key of the endorsement hierarchy and is calculated starting from the EPS. The manufacturers can inject an EK together with its certificate signed by the manufacturer before the device shipment. The end user can then use the provided key and certificate to verify that the EK public key is associated with a *genuine* hardware TPM produced by the right manufacturer.

The TCG defines an *Endorsement Key* (EK) as an asymmetric key pair consisting of a public and private key stored in a Shielded Location on the TPM[6], that can be generated using the supported algorithms provided by the TPM. The public part of the EK can be read either from the TPM or, if existent, from the corresponding certificate; while the private part of the EK MUST be never exposed outside the TPM's boundaries. In TPM 1.2 specifications the EK could only be created as a decryption key, while the TPM 2.0 gives the possibility to generate the key either as a decryption key or as a signing key, providing more flexibility to the end users or the manufacturers. However, the TCG recommends generating the EK as a decryption key and restricting it to certain operations. That is because the EK and the EK certificate are considered privacy-sensitive when used in a cryptographic protocol, for example in a Remote Attestation Protocol. In such cases, a signing key has to be generated as well to be able to sign TPM internal data when needed.

The TCG generally defines an *Attestation Key* (AK) as a non-duplicable asymmetric restricted signing key[7], which means that it can be used to sign only TPM-generated values to prevent the signing of external data but with the same format as TPM-generated values. An AK that is secured by a TPM can be trusted to provide accurate reports on Shielded Location content, such as PCRs content, and not sign any externally provided data that appears to be valid but is not TPM-produced. However, an identity certification process is needed to link an AK with the platform it represents, without which it would hold little value for a remote

challenger that wants to verify any report signed with that AK. This process takes the name of *Credential activation*.

Credential Activation

A TPM user can create an AK and request a certificate for it from a third-party attestation Certificate Authority (CA). The CA may require proof that the key is resident on a TPM, which can be provided using a previously generated key, like an EK, or a Platform Certificate from the same TPM.

The Credential Activation process is a challenge-response protocol that involves two actors: the Credential Provider (CP) and the TPM. It can be described with the following steps[5]:

1. The CP receives the AK's public area and the EK certificate. The corresponding EK key is an encryption/decryption key and its certificate is typically issued by the TPM manufacturer.
2. The CP validates the EK certificate by verifying its chain up to the issuer's root. The CP also makes sure that the EK is fixed to a compliant TPM.
3. The CP examines the AK's public area and decides if it is a certifiable key. It also makes sure that the AK is a restricted key that is fixed to the TPM.
4. The CP generates a *credential* for the AK.
5. The CP generates a *secret* that is used to protect the credential. Typically, this is a symmetric encryption key.
6. The CP generates a *seed* to a key derivation function (KDF).
7. This seed is encrypted with the EK public key. Thus, it can later only be decrypted by the TPM.
8. The seed is used in a TCG-specified KDF to generate a symmetric encryption key and an HMAC key. The symmetric key is used to encrypt the secret, and the HMAC key provides integrity. It is important to know that the KDF also uses the AK's *name*, which is the digest of the AK's public key and its attributes and it also identifies the key.
9. The encrypted secret and its integrity value are sent to the TPM, as well as the encrypted seed.

On the TPM's side the following steps are needed to recover the credential generated from the CP:

1. The seed is recovered using the EK's private key, and the TPM retains the seed.
2. By using the public part of the AK and its attributes, the TPM computes the AK's name.

3. By using the same TCG KDF and the AK's name and seed as input, the TPM can reproduce the same symmetric encryption key and the HMAC key.

4. Now the TPM can recover the secret and returns it to the user.

The protocol ensures that the recovery of the credential is dependent on the possession of the private key linked to the EK certificate, and the TPM having a key (the AK) that is identical to the one presented to the credential provider.

The user can now request a certificate for AK by handing the obtained secret to the CP:

- The CP, after receiving the secret from the requester, checks it is equal to the previously generated secret. If this verification fails, the CP will terminate the certification issuing process.

- The CP issued a certificate for the AK and sends it back to the requester.

In conclusion, a descendant certificate (the AK certificate) has been issued, with the EK certificate being its parent. This allows for remote parties to verify the authenticity and the provenance of any data signed by the AK certificate, as it is linked to a trustworthy parent certificate resident in a genuine TPM.

2.2.4 Platform Configuration Register

The TCG defines a *Platofrm Configuration Register* (PCR) as a mean to provide a method to cryptographically record software state: both the software running on a platform and configuration data used by that software[5]. When a computer turns on or when a TPM Reset command is thrown, the TPM sets all PCRs to their initial value, either all zeroes or all ones, as defined in the TPM platform specification. It is not possible to change a PCR value directly; instead, the TPM uses the *Extend* operation to modify a PCR value. The Extend operation is a cryptographic process that guarantees that the PCR value is unique based on the specific sequence and combination of digest values that were extended, and it is defined as follows[4]:

$$PCR_{new} := H_{alg}(PCR_{old}||digest)$$

The || symbol is intended to specify a concatenation between the value of the PCR (before the Extend operation) and the data to be extended. Finally, the new PCR value is obtained by applying one of the hash algorithms offered by the TPM. Since the TPM 2.0 can offer a variety of hash algorithms, PCRs are organized in *banks* that differ only in the algorithm they are referring to. The same PCR can then be allocated to multiple banks, allowing any combination based on the various hash algorithms. The TPM is typically shipped with two banks available, the SHA-1 and SHA-256 banks. A user can activate and allocate a new bank by using the *TPM2_PCR_Allocate* command. The typical number of PCR registers that can be allocated in each bank is a minimum of 24.

PCR Index	***PCR Usage***
0	CRTM, Host Platform Code (BIOS)
1	BIOS Configuration
2	UEFI driver code
3	UEFI driver configuration
4	MBR (Master Boot Record)
5	MBR Configuration
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Static Operating System
16	Debug
23	Application Support

Table 2.1. PCR Allocation [5]

The expected results of the PCR can be determined if the sequence of steps used in the PCR called is known. However, if the sequence of steps is not predictable, it is not possible to know the expected results beforehand. To address this, the system keeps a record of all changes made, known as the Measurement Log (ML). Each entry in the ML represents a change in the system state, known as a Measurement Event (ME). The PCR results can be used to verify the integrity of the ML and ensure that each change made was acceptable.

2.2.5 Trusted Boot

The TPM uses multiple PCRs for better evaluation of the boot to run-time process. This is done by dedicating each PCR to store measurements of different modules, rather than using a single PCR for all the digests of the boot sequence. Even though it is technically possible to use a single PCR for this purpose, using multiple PCRs simplifies the evaluation process. Table 2.1 shows an example of how PCRs can be allocated.

Usually, PCRs in the range of 0 up to 9 are dedicated to the measurement of events that happen during the system booting sequence; while PCRs from 10 upwards are dedicated to the measurement of events related after the kernel booted. As mentioned before, the verification of PCR values can be made to ensure the state of the platform's software state. The *Trusted Boot* process involves taking measurements of all software components and configuration files involved in starting up the system and storing their unique digest values in a specific PCR. This mechanism enables an independent entity to confirm the secure boot of a system by validating the PCRs associated with the boot process.

The Trusted Boot process must ensure the following three key points[8]:

- The chain of trust must be established in a sequential manner. Before granting control rights, the executable entity must be measured by the TCB and can

only gain control rights after its integrity has been verified, completing the chain of trust establishment process.

- The TPM must be responsible for completing all metrics and calls involved in establishing the chain of trust.
- During the chain of trust establishment, all important secret data, including keys, pre-measurement data, and verification data, must be stored and sealed within the TPM for integrity and confidentiality.

A TPM-based Trusted Boot process in a Linux environment is shown in figure 2.2. The mechanism can be roughly described in the following steps[8]:

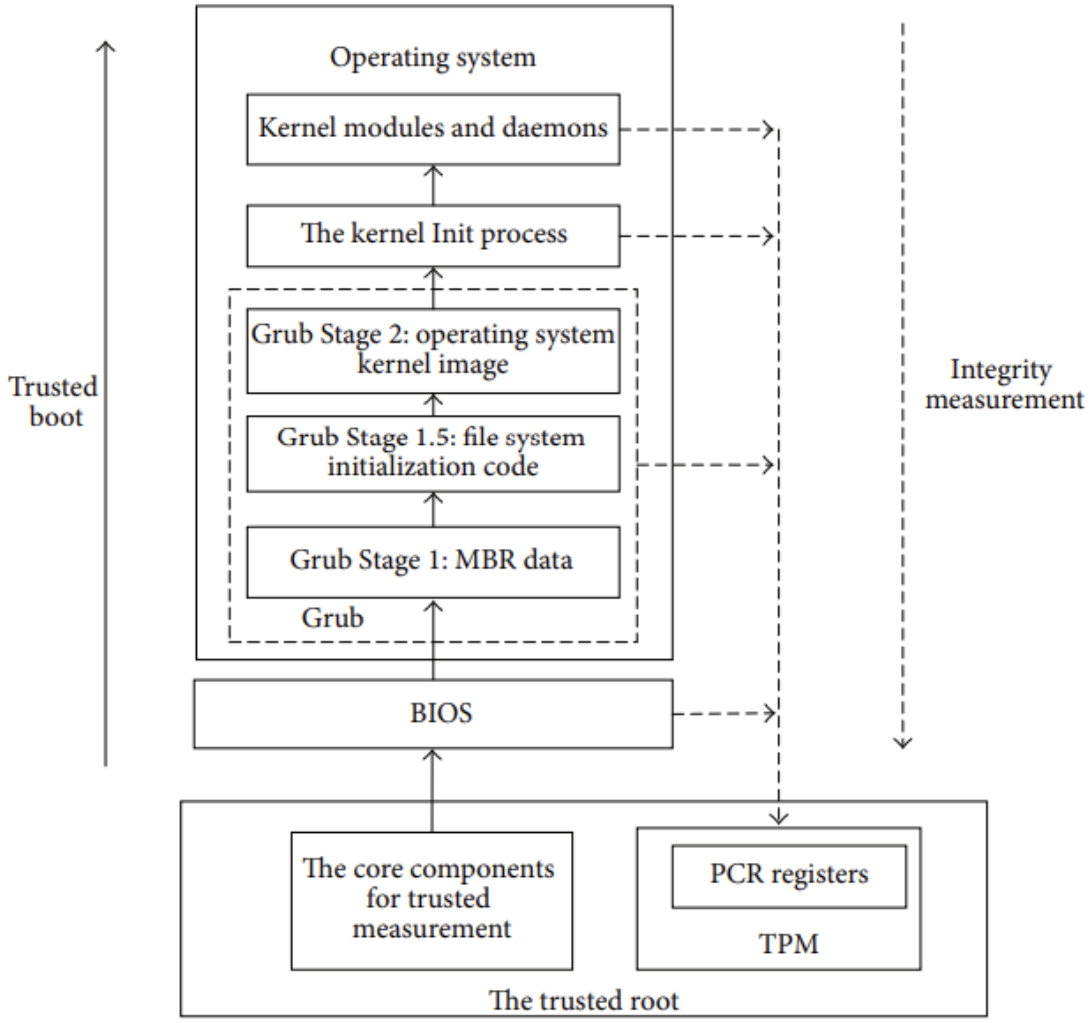

Figure 2.2. Trusted Boot process (Linux) [8]

1. During the Trusted BIOS process, the boot loader stored in the boot sector is loaded and sent to the TPM for measurement and verification. After the

TPM confirms its integrity, the boot program is loaded into memory, then the BIOS transfers control rights to the CPU to run the boot program and load the operating system.

2. The TPM checks the integrity of the operating system loader program, such as Grub in Linux. If the validation is successful, the Grub Stage 1 code in the master boot sector is loaded into memory and takes control of the trusted boot process to proceed with loading the operating system kernel.

3. The trusted boot process continues with Grub Stage 1, which verifies the integrity of the Grub Stage 1.5 code using TPM. If the validation is successful, the code for the Stage 1.5 phase is loaded and executed. Upon completion of this stage, the file system is mounted.

4. The Grub Stage 2 code is first verified by TPM and then loaded by the trusted Grub Stage 1.5. Once it gains control, it verifies the integrity of the configuration file "/boot/Grub/Grub.conf", which holds the details of disk partitions, the kernel image, and the virtual RAM disk file initrd.

5. The Grub Stage 2 code opens the configuration file, reads the operating system kernel image then uses the TPM to verify its integrity. Upon successful validation, the operating system kernel image is loaded and given control.

6. After the operating system kernel image is successfully loaded, TPM will evaluate and confirm the integrity of the Init process. If the process is validated, the kernel key data structures will be established and the kernel Init process will take control.

7. The Init process starts by identifying the list of kernel modules and daemons required to be loaded based on the system configuration. Then, it uses TPM to verify the integrity of each kernel module and daemon before they are loaded. Only trusted kernel modules and daemons are executed sequentially to ensure a secure computing environment. Finally, the Init process begins to accept user input, and the trusted computer is ready for use.

2.2.6 TPM 2.0 Implementations

The TCG offers different types of TPM implementations to accommodate various use cases. Each implementation has its own strengths and limitations, making it suitable for specific applications. Currently, the most widely used TPM implementations are[9]:

1. ***Discrete TPM*** offers the most robust level of security. To achieve this, a discrete chip is designed, manufactured, and evaluated to withstand tampering, including probing and freezing with advanced attacks. Thus it is recommended for critical systems.

2. ***Integrated TPM*** is the second level of security. This level also includes a hardware TPM, but it is integrated into a chip that offers features other than

security. The hardware implementation makes it resistant to software errors, However, this level is not intended to be tamper-proof.

3. ***Firmware TPM*** is a software-based implementation that utilizes a protected execution environment, known as a Trusted Execution Environment (TEE), to safeguard sensitive information such as private keys. This type of TPM does not require a separate hardware chip, as it runs on the main CPU, but it offers a more secure environment for TPM operations as it is separated from the rest of the programs running on the CPU. The disadvantage of using a firmware TPM is that it relies on multiple elements to maintain security, such as the TEE operating system and the application code running within it. This makes it more vulnerable to security issues compared to a discrete or integrated TPM, which has built-in tamper resistance.

4. ***Software TPM*** can simulate the functionality of a TPM using software, but it is susceptible to various vulnerabilities such as tampering and operating system bugs. However, it is useful for testing and prototyping TPM-based systems.

2.3 TPM 2.0 Software Stack

The TSS is a software stack created to shield TPM application developers from the intricate details of interacting with TPM. The TSS comprises several layers, as shown in figure 2.3, making it possible to customize scalable TSS implementations for high-end and resource-constrained low-end systems[10].

The lowest layer is the *TPM Device Driver* which is a software component that is specific to the operating system and is responsible for managing communication with the TPM, as well as reading and writing data to the TPM.

The *Resource Manager* manages the movement of objects, sessions, and sequences in and out of the limited TPM memory as required. This is done by executing context swapping.

The *TPM Access Broker* (TAB) manages access to the TPM by multiple processes, ensuring that each process can complete its TPM command without interruption from other processes.

The next layer is the *TPM Command Transmission Interface* (TCTI) is responsible for managing the communication between the TPM and the higher layers of the TSS stack. It provides different interfaces depending on the type of TPM being used (see Sect. 2.2.6 for additional details). Additionally, it supports both legacy TIS and command/response buffer (CBR) interfaces for communicating with the TPM.

The *System API* (SAPI) provides comprehensive access to the features of a TPM 2.0 implementation and is intended for use in various contexts such as firmware, BIOS, applications, and operating systems. It serves as a low-level interface that is best suited for advanced applications.

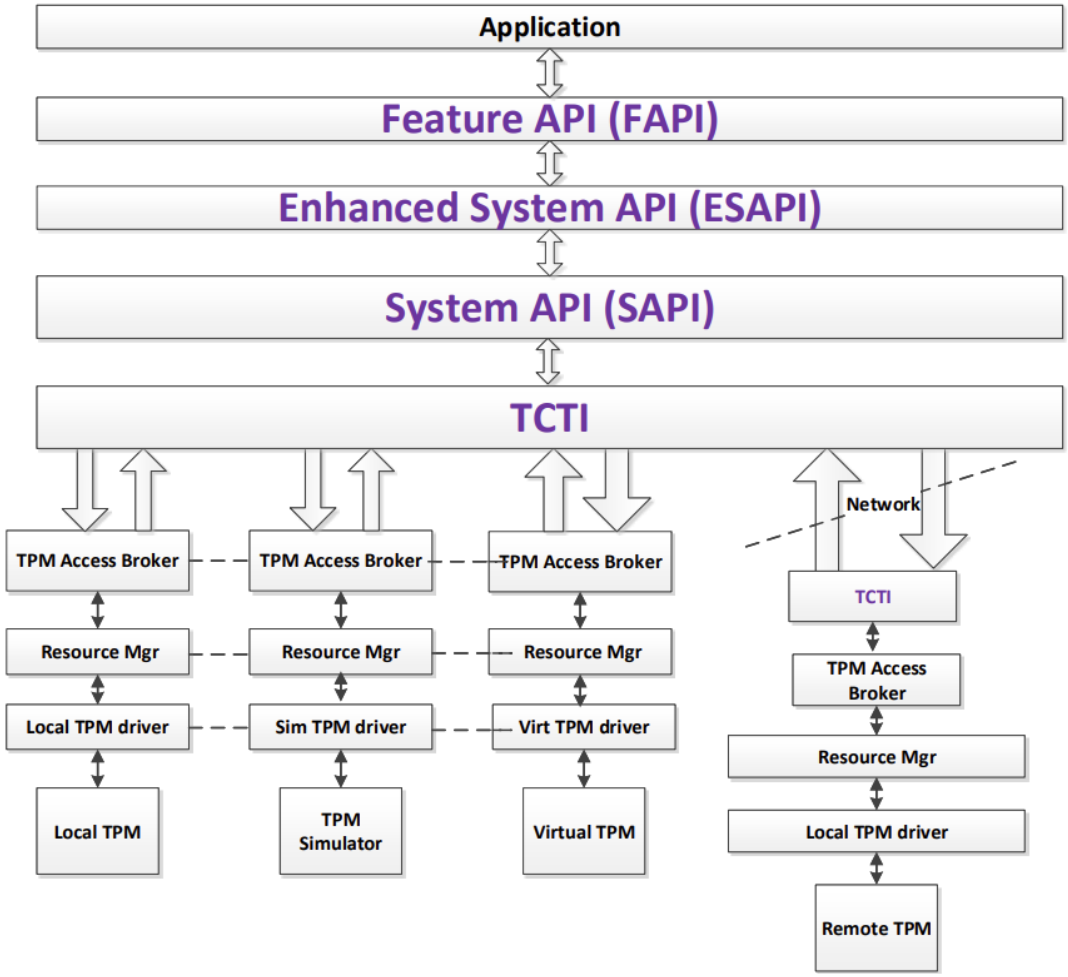

Figure 2.3. TPM 2.0 Software Stack [10]

The *Enhanced System API* (ESAPI) aims to simplify the process of accessing the TPM for applications by providing an interface that abstracts away the complexities of low-level TPM calls. However, a deep understanding of the interface to a TPM is required because the use of this layer still requires cryptographic operations on the data that are being sent and received from the TPM.

The *Marshalling/Unmarshalling API* is responsible for converting TPM commands and responses into a format that can be transmitted over a communication channel, and vice versa. It is used by both the SAPI and ESAPI.

The *Feature API* (FAPI) offers a simplified interface for application developers, allowing them to perform tasks without the need for knowledge of low-level details. This higher-level software abstraction is intended to make it easier for developers to work with the TPM. The downside of this high abstraction layer is that it only provides 80% of the actual TPM's functionalities[11].

Chapter 3

Remote Attestation

The *Remote Attestation* process involves a remote entity, known as the Verifier or Challenger, assessing the trustworthiness of a computational node, referred to as the Attester or Prover, through a challenge/response protocol. RA and the TCB are closely related concepts in the field of trusted computing. The TCB is part of a system that is responsible for providing security guarantees, such as confidentiality, integrity, and authenticity. RA allows a remote entity to verify the authenticity and integrity of the TCB. For RA to be effective, the TCB must be designed and implemented in a way that allows it to be accurately measured and attested to. The TCB provides the measurements and evidence that are used in the RA process, such as cryptographic hashes of firmware and software components, trusted boot measurements, and other system states.

In recent years, RA has been increasingly used in a variety of applications and also in different scenarios. Based on the implementation, RA can be classified into three different approaches[12]:

- ***Software-based RA*** does not rely on specialized hardware, making it inexpensive and easy to implement. However, the absence of specialized hardware also limits the level of security that can be achieved. Software-based RA uses checksum computation time as a means of verifying the integrity of a device. However, this method is vulnerable to attacks from adversaries with more powerful computational resources. To counter this, software-based RA protocols assume that adversaries cannot collaborate with other devices, that the checksum and attestation code cannot be optimized further, and that the attestation cannot be parallelized. While these assumptions provide some defense against attacks, a local adversary with faster network latency or stronger computational power could potentially bypass the attestation.
- ***Hardware-based RA*** offers the highest level of security and can be applied in critical situations. This type of RA typically uses TPMs to protect cryptographic keys and record the software state of a computing system in special registers called PCRs (for further details see Sect. 2.2.4). PCR values can be used as evidence of the system's state, so if there is any discrepancy between the measurement log and the PCRs, it means the integrity has been compromised.

- ***Hybrid RA*** combines hardware and software to create an attestation protocol that is more secure than software-only attestation but less expensive than hardware-only attestation. The security architecture may include a ROM and MPU (MicroProcessing Unit), where the ROM is used to store the secret keys and the attestation code, and the MPU ensures that only authorized processes can access the memory of the ROM, providing immutability and exclusive access properties for the RA. Hybrid attestation approaches use the properties of security architectures to protect against local and remote adversaries who only operate on the software. The key protection and immutability of the attestation code prevent these adversaries from breaking the attestation. Additionally, the attestation protocols rely on the security of HMAC algorithms and other cryptographic functions, so if these are compromised, the attestation will be vulnerable.

In this work, the focus is on hardware-based RA approaches that leverage the use of a TPM that acts as hardware RoT. As seen in Sect. 2.2.5, the trusted boot process allows for the measurement and storage of the integrity of the boot sequence in dedicated PCRs, specifically PCR0-7. This enables a remote entity to verify that a platform has booted correctly by analyzing an integrity report that can be generated on the target platform. However, the system's integrity could be altered even after a successful boot. This is because the software that runs post-boot, such as kernel modules, device drivers, or privileged applications that are dynamically loaded, can potentially execute at any time and compromise the system's protection requirements.

3.1 Integrity Measurement Architecture

The *Integrity Measurement Architecture* (IMA)[13] is a Linux kernel module that allows for the extension of the chain of trust from the BIOS to the application layer: it extends the principles of Trusted Boot to the Linux kernel, making IMA an important component to be included in the TCB of a Trusted Platform. IMA has been a part of the Linux Integrity Subsystem since version 2.6.30 in 2009, and it is currently a widely accepted TCG-compliant solution for measuring runtime loaded content. Before kernel modules, configuration files, and executables are loaded onto the Linux system, IMA measures the integrity value of each loaded component and extends these measurements in the TPM. This allows external entities to verify not only the boot of the system but also the applications and kernel modules loaded in the platform.

When IMA is enabled in the Linux kernel, it extends the measurements in PCR10; but it can be changed by modifying the kernel's configuration. As opposed to the trusted boot process, the order in which IMA aggregates the measurements into the PCR is not predictable. That is because after the kernel takes control, a variety of software components are loaded and managed (e.g. kernel modules, shared libraries, executables) in a non-predictable order. For this reason, a way to track the order in which the measurements were made is needed. IMA implements this by appending to a Measurement Log (ML) file every Measurement Event (ME)

that occurs in the system. The IMA PCR protects the integrity of this ML. In this way, IMA enables a remote entity to determine if the ML is consistent with the IMA PCR value received, being able to assess the trustworthiness of the dynamically loaded components in the system.

3.1.1 IMA Design

The kernel integrity subsystem aims to identify if files have been changed without authorization, both from remote and local sources, compare a file's measurements to a stored "good" value as an extended attribute, and maintain the integrity of local files[13]. To achieve such goals, IMA provides several integrity functions:

- ***Collect***: measure a file prior to accessing it.
- ***Store***: add the measurements to the kernel ML and extend the IMA PCR if the platform is TPM-equipped.
- ***Attest***: sign the IMA PCR to allow a remote entity to validate the ML.
- ***Appraise***: ensure the measurement of a file matches the stored "good" value in its extended attribute before allowing access.
- ***Protect***: safeguard the extended attributes (such as the appraisal hash) of a file from offline attacks.
- ***Audit***: review the file hash values.

These functions have been incorporated into three IMA major components:

- ***IMA Measurement*** is a Linux kernel subsystem that uses the TPM to provide measurements of the system's runtime state. IMA Measurement is responsible, on the attesting system, for determining what files to measure, performing measurements on files, and, securely maintaining them. Hence, it enables a challenging system to perform the RA of the entire system's runtime state of the attesting system. Verifiers can retrieve the measurement list and validate its freshness and integrity. By linking the aggregate integrity value to the TPM, it ensures that any unauthorized changes to the measurement list would be detected, making IMA measurement a reliable way to confirm the integrity of the system. This means that IMA measurement can be used to attest to the system's runtime integrity by providing a secure and tamper-proof record of all the files executed and loaded into memory
- ***IMA Appraisal*** is an extension that builds upon the IMA measurement component ensuring the local integrity of files by comparing their measurements against a pre-determined good value stored in the "security.ima" extended attribute. This validation process uses hashing and digital signature, to guarantee both the integrity and authenticity of the files. This way, IMA Appraisal can provide additional security and protection to the system during runtime.

- ***IMA Audit*** is responsible for maintaining a record of all file measurements and names in the system audit logs, which can later be used for advanced security analysis and forensic investigations.

The various aspects of the kernel's integrity subsystem, such as IMA Measurement, IMA Appraisal, and IMA Audit, complement one another to provide a comprehensive security solution. However, these functions can also be configured to operate independently, giving users the flexibility to apply only the security measures that are necessary for their specific use case.

3.1.2 IMA Measurement

The component responsible for constructing the ML is the IMA Measurement. The ML file is located in the **securityfs** and is available in two different formats: one in ASCII format called **ascii_runtime_measurements**, and the other in a binary format called **binary_runtime_measurements**. The ML describes the order in which the aggregate value stored in the IMA PCR is calculated in terms of Measurement Events. The IMA Measurement process begins upon receipt of a ME (e.g. loading a program, mapping a file in RAM, or opening a file) by an IMA Hook in the system. The process calculates the hash value of the file's content using a secure hash function. The file digest and related metadata are then stored in a list of MEs in the kernel, and the digest is added to a TPM PCR (often PCR 10) via the extend operation. The first entry of the ML is always the **boot_aggregate** (as shown in figure 3.1) which can be either all zeros if a TPM is not present on the platform, or its value is the digest computed over the trusted boot PCRs (PCR0 to PCR7). After boot_aggregate, IMA Measurement assesses all accessed files that conform to a measuring rule in the IMA policy to determine if re-measurement is required. A new measurement is performed only if:

- The file has not been measured yet.
- The file content has changed since the last measurement.
- The kernel cannot detect changes in that file.

Additionally, the Extend operation is executed before the measured component (executable or data file) takes control of the platform, preventing a potentially corrupted component from extending a measurement that does not reflect its actual state. While a corrupted component may cause additional extensions to the IMA PCR once it has gained control of the platform, the secure hash algorithms ensure that the aggregate in the IMA PCR cannot be altered to represent a trusted system. Therefore, corrupt systems may alter the measurement list, but this is discovered by recomputing the list's aggregate and comparing it with the securely stored aggregate in the TPM [14].

IMA lets the user customize the measurement behavior by adding some kernel command line parameters that can be specified in terms of the **policy**, that IMA adopts during the measurement and re-measurement phase; also in terms of the

template that constitutes the format of every ME stored in the ML; and finally in terms of the **hash algorithm** used to describe the integrity value measured by IMA.

IMA Policy

The IMA policy can be specified in the kernel command line parameters, by specifying *"ima_policy=built-in policy"*. The available built-in policies are the following [15]:

- **tcb**: this policy measures kernel modules, software in execution, files mapped into memory for execution via mmap, and files accessed for reading by the root user.
- **appraise_tcb**: this policy goes beyond mere measurement and conducts in-depth evaluations of the same components as the tcb policy. It denies access to any files that fail to match their previously established and verified hash values.
- **secure-boot**: this policy specifically evaluates the kernel, its modules, and the IMA policies.

Those are the policies that IMA offers and the user can choose which one fits better for its environment. However, the IMA measurement policy can be modified by the administrator by accessing the *securityfs* and modifying the **"/ima/policy"** file.

IMA Hash

The hashing algorithm that IMA uses during the measurements can be specified via the *"ima_hash=hash"* kernel command line parameter. The available hash algorithms can be looked up in the kernel's source file **"/crypto/hash_info.c"**.

IMA Template

As for the policy, the IMA template can be specified in the kernel command line parameters, by adding *"ima_template=template"*. IMA offers three possible templates:

- **ima** template allows the recording of events only with SHA-1 or MD5 digests, with the name of the event limited in size (up to 255 bytes).
- **ima-ng** template allows the recording of events with an arbitrary hash algorithm implemented by IMA, with the name of the event unlimited in size. The hash algorithm can then be specified by the "ima_hash" command line parameter.

- **ima-sig** template has the same properties as the "ima-ng" template, but it adds a signature based on either the file's digest or on the extended file attribute "security.ima".

Figure 3.1 shows an example of how an ASCII-based ML is formed when the kernel parameters are: **ima_hash=sha256** and **ima_template=ima-ng**. The fields, listed from left to right, represent:

- the *PCR's index*, in this example PCR 10, where the entry was extended.
- the *template-hash* which is the SHA-1 digest that has been extended into the IMA PCR. If the PCR bank is a non-SHA-1 bank, requires zero padding of the digest to meet hash algorithm requirements. For example, if the IMA PCR is in the SHA-256 bank, the 20-byte digest is padded to 32 bytes.
- the *template-name* used for the given entry.
- the *filedata-hash* which is the digest calculated over the file's content. In this case, the hash algorithm is SHA-256.
- the *filename-hint* that is the pathname of the file.

PCR	Template-hash	Template-name	Filedata-hash	Filename-hint
10	b2428e790[...]36ed834d6a	ima-ng	sha256:999cdeec[...]e5fc0417	boot_aggregate
10	aod123sa3[…]ffoi40948nld	ima-ng	sha256:918df370[...]a10a33ab	/init
10	s928asjbh7[…]q109k8qwb0	ima-ng	sha256:12b731d8[...]e8099sd3	/usr/lib64/ld-2.16.so
10	dbj1l204ujn[…]df378kboa8	ima-ng	sha256:b73daou8[...]e809sdsd	/etc/ld.so.cache
…	…	…	…	…

Figure 3.1. Example of IMA ML with template **ima-ng** and **sha256** digests

.

3.2 TPM Quote Operation

The IMA and Trusted Boot mechanisms enable remote verification of a platform's state through Remote Attestation (RA). A remote entity (Verifier) can challenge a given platform (or attesting system) to send a report containing the requested PCR values and eventually the IMA ML if PCR10 was specified as well. The PCRs together with the IMA ML only provide integrity, thus the Verifier does not have the insurance that the PCRs values are coming from a trusted and genuine TPM. For this reason, the TPM provides the **Quote** operation.

The Quote operation defines the authenticity of the report generated in response to a RA request. The operation consists in hashing the requested PCRs and then signing the digest with a non-duplicable restricted signing key: an *Attestation Key* (AK). This high-level description of the quote operation hides important security pieces of information that a quote's structure (the one hashed and signed) internally contains [5]:

- *Magic number* TPM_GENERATED: protects against unauthorized use of a restricted signing key to sign arbitrary data and falsely claim it as a TPM quote.
- *Qualified name of the signing key*: A key may seem secure but be vulnerable due to a weaker ancestral algorithm. The qualified name represents all the descendants of the key.
- *Extra data provided by the caller*: This information usually consists of an anti-replay nonce, serving as evidence of the quote's freshness.
- *TPM firmware version*: Included to enable the verifier to determine trust in a specific TPM code version.
- *TPM clock state*: represents the number of times a TPM has been restarted/resumed. When the signing key is not in the endorsement hierarchy, this value is obfuscated because it could aid in correlation.
- The attestation structure type (in this case, a quote).
- The PCRs selected to be included in the quote.
- The digest of the selected PCRs, called *calcDigest*.

To verify the authenticity of the quote, the attesting system needs to provide a way to retrieve the AK certificate, so that the verifier can be ensured that the AK is an actual TPM-resident key having as the root of the chain of trust the EK.

3.3 Remote Attestation Protocol

In this section, a generic RA Protocol is proposed describing all the interactions that happen between the various actors and all the components needed to perform a TCG-compliant RA. As already mentioned, to enable RA two actors are required: the Verifier or Challenger, and the Attester or Remote Attestor. The Attester platform must be TPM-equipped and has to provide an EK with the corresponding EK certificate (usually issued by the TPM manufacturer). These are needed to represent the root of the chain of trust for the attestation keys (Aks). The Attester should also be able to receive RA challenges coming from the remote Verifier employing a *Trusted Platform Agent* (TPA). The TPA should be a privileged application with the ability to interact with the TPM and request attestation-related operations, such as requesting to perform a quote with a given AK. On the other

side, the Remote Attestor needs to be able to retrieve the AK certificate from a Certification Authority, or from the platform itself. Also, to verify that the PCR values represent a trusted state for the attesting platform, the Verifier must compare each measurement contained in the ML with a list of *"golden values"*; the **whitelist**. The whitelist is typically made of a filename hint (i.e. the file pathname) with its associated integrity measure (i.e. the file data hash). The Remote Attestor can retrieve the whitelist either from the attesting system, where its administrator generated the list in a secure environment the first time the platform has booted; or from the platform manufacturer and the software vendor.

Figure 3.2 shows how the Trusted Boot and the IMA Measurement components enable RA.

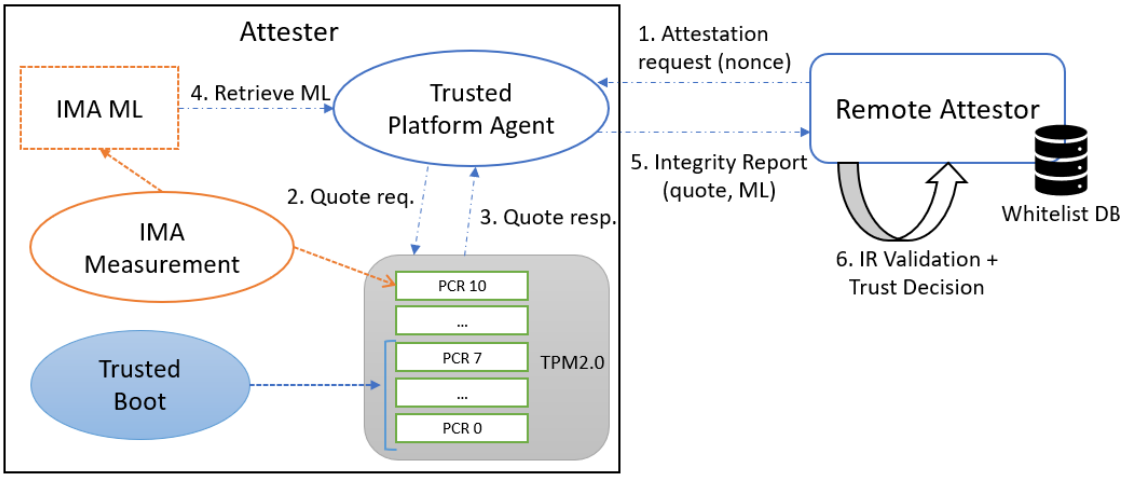

Figure 3.2. Remote Attestation scheme.

The initiator of the RA challenge/response protocol is the Remote Attestor which sets off a series of operations that have to be performed by the TPA to respond to the challenge. The chain of events and operations referring the figure 3.2 are the following:

1. The Verifier requests an attestation from the TPA that resides on the Attester system, providing a non-predictable **nonce** and a **list of PCRs** to be included in the quote (usually the Trusted Boot and IMA-related PCRs).

2. The TPA is then responsible for requesting the TPM to perform a Quote operation, including in the request the nonce and the list of PCRs that have to be included in the quote response. The nonce plays an important role because it allows the verifier to check the quote's freshness, thus protecting against replay attacks (see Sect. 3.2 for a detailed description of the Quote operation).

3. The TPM fulfills the quote request by loading the AK and using its private portion (AK_{priv}) to sign the hash of the selected PCRs, the nonce, and the other quote's structure metadata.

4. The IMA ML is retrieved by the TPA.
5. The TPA constructs and then sends an Integrity Report (IR) consisting of the IMA ML, the signed quote together with the quote's structure, and the public part of the AK (AK_{pub}), eventually with its certificate.
6. Once the Remote Attestor receives the IR, it validates the quote's freshness by comparing the nonce contained in the quote's structure with the previously sent nonce, and also the quote's authenticity by verifying the signature with AK_{pub}. Also, it validates the boot process of the platform, and the integrity of the IMA ML, and finally assesses the trustworthiness of the runtime measurements contained in the IMA ML by comparing them to the measurements contained in the whitelist.

The quote's structure that the Verifier receives contains a field called *calcDigest*, which represents the digest of the PCR values selected to perform the quote. The Remote Attestor uses this digest to verify the integrity of the quoted PCR values by recalculating this digest and comparing it with the one present in the quote's structure. For example, let us consider the case in which the Verifier requests only to quote the IMA PCR. When the IR is received the IMA PCR aggregate value has to be recomputed in the same way the IMA Measurement component does. Figure 3.3 shows how the IMA PCR is extended in the SHA-256 bank: for every measurement in the ML extend the 20-bytes *template-hash* with the hash algorithm that matches the PCR bank until the last measurement is reached.

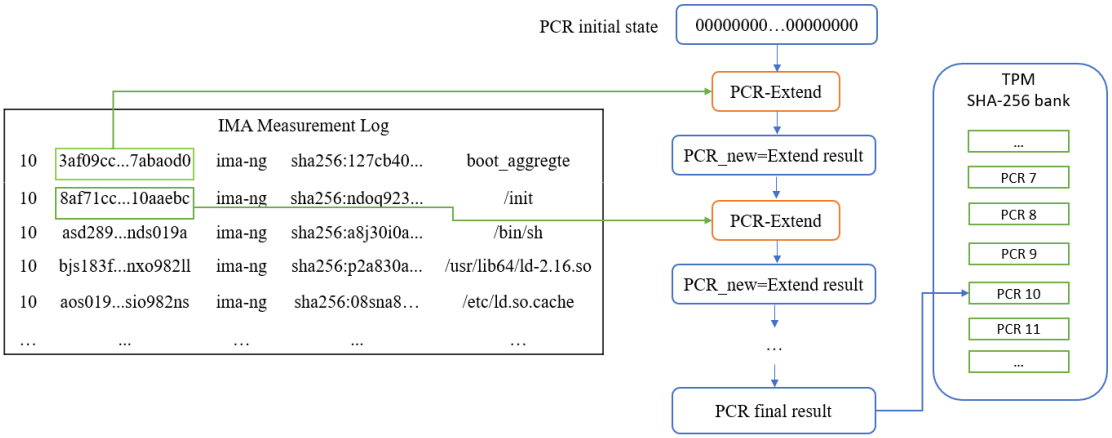

Figure 3.3. IMA PCR calculation process.

Thus, the Verifier reads the ML and computes the PCR aggregate value, and subsequentially can perform the digest over the calculated value and compare it with the one present in the quote's structure. If the comparison of the two digests results in a match, it indicates the ML has not been tampered with by an attacker and can be relied upon to assess the Attester platform's integrity state.

To decide whether the Attester is in a trusted state, the *whitelist* is exploited to evaluate the trustworthiness of the measurements contained in the IMA ML, as

shown in figure 3.4. Validation of measurements for executable files is equivalent to that of data files. For every entry in the whitelist, the Remote Attestor checks whether the ML contains an event whose filename hint is the same as the one present in the whitelist. The Verifier may then find himself in one of these three situations:

1. The ML contains the ME with a filename hint equal to the one present in the whitelist. There is a match between the file data hash of the ML event and the one stored in the whitelist, which means that the file or executable is in a known trusted state that does not alter the platform's trustworthiness.

2. The ML contains the ME with a filename hint equal to the one present in the whitelist. There is NOT a match between the file data hash of the ML event and the one stored in the whitelist, which implies that the file could be an updated version of the file/program or its data/code have been altered by an attacker.

3. The ML does NOT contain the ME with a filename hint equal to the one present in the whitelist. This could imply that the file/executable has not been measured because it has not been opened or executed; or even worse it could happen that the file has been deleted or altered by a malicious actor.

The Remote Attestor must have a policy that outlines the steps to be taken when unknown file names or untrusted measurements are detected in ML to properly evaluate the level of trust of the Attester's platform.

The Remote Attestation process should be repeated **periodically** to monitor the Attester system's integrity over time.

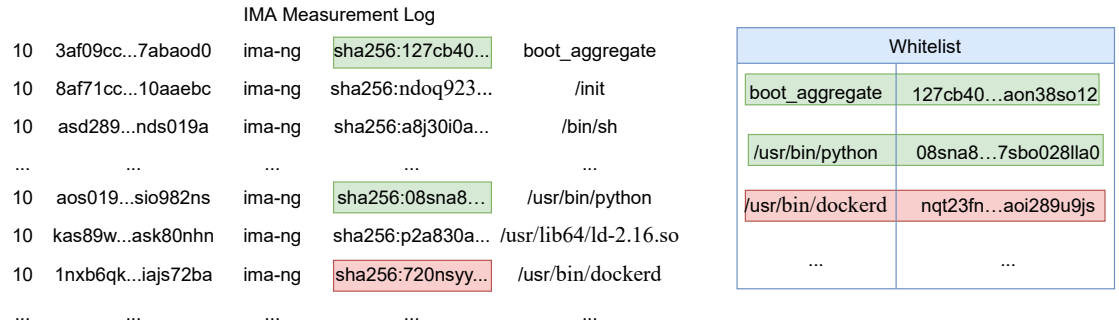

Figure 3.4. Integrity Measurements Validation against whitelist.

Chapter 4

Distributed Trusted Computing Base

Distributed Ledger Technologies (DLTs) are gaining increasing attention as a potential basis for the global financial system of the future. The security, reliability, and longevity of DLT systems are becoming increasingly important as they take on a larger role in the digital economy, serving as the infrastructure for future cryptocurrency and virtual asset exchange networks. With DLTs the exchange of any kind of information between different parties is no longer processed through a central authority, such as banks for traditional financial systems, but the exchange is completely managed between the parties/peers in a **decentralized** manner. Transactions are then stored in the distributed ledger and validated by employing a consensus protocol among the decentralized nodes. However, **trust** between the parties needs to be achieved otherwise, a consensus outcome reached by a group of decentralized nodes is of little value if those nodes have been infected by malware or viruses without their knowledge.

To this extent, the notion of Distributed Trusted Computing Base (DTCB) will be introduced by expanding the principles of Trusted Computing, such as TCB (see Chap. 2 for further details) and RA (see Chap. 3 for further details), to operate in decentralized P2P networks enabled by DLTs. The objective is to create a DTCB model that strengthens the trust level among all participating nodes in forming the DTCB, utilizing a DLT-based RA protocol. As a means to restrict the scope of the problem, the proposed solution operates within a **permissioned** DLT allowing only authorized nodes to participate as peers in the DTCB network.

Hence, a proposal of a DTCB model will be given, accompanied by a brief look at the features provided by DLTs, and an examination of the DLT selected for the proposed implementation.

4.1 Distributed Ledger Technologies

The first widespread DLT was introduced in 2009 by *S. Nakamoto*, called Blockchain [16]. A blockchain is only one type of DLT that stores transactions in a specific format (linked list of blocks), while there are other forms of ledgers with alternative

data formats. When a ledger, including a blockchain, is spread across a network, it can be considered a Distributed Ledger or simply a ledger. A **Distributed Ledger** is an *append-only* decentralized database that is managed by multiple participants. This database consists of multiple identical copies that are distributed among the participants and updated in a synchronized manner. Unlike traditional distributed databases where participants trust each other to maintain data consistency, in a distributed ledger the parties do not fully trust each other and a mechanism is needed to verify the ledgers collectively before they are shared. Hence, a consensus protocol enables all nodes in a distributed ledger to reach an agreement on a single, definitive version of the truth, without relying on a trusted third party.

DLT enables decentralization by relying on a peer-to-peer (P2P) network used to scale up the system, eliminate a single point of failure, and prevent a single entity or small group from dominating the network. Another core aspect of DLT is that they can ensure **immutability and irreversibility of the Ledger State**. That is because by achieving consensus among a large number of nodes in a distributed manner, the state of the ledger becomes practically immutable and irreversible after a certain period. Different ledger deployment strategies exist depending on the application domain. These strategies lead to two main types of ledgers [17]:

- **Permissionless or Public Ledger**: this enables nodes to both create and validate blocks, as well as execute transactions that store and update data between participating nodes, thus modifying the ledger state. This implies that anything regarding the ledger (its state, transactions, and data store) are transparent and accessible to any entity. Hence, this solution is not suitable in scenarios where data need to be privacy-preserved.
- **Permissioned or Private Ledger**: it only allows authorized and trusted entities to participate in its activities. This restriction ensures the privacy of ledger data, making it suitable for use cases where privacy is a concern.

Nowadays, DLTs are most commonly used for cryptocurrencies, which are digital assets that employ cryptography to secure transactions and use distributed ledgers to store them. This eliminates the need for a traditional trusted third party, such as a bank, to hold user-sensitive information and to execute the money exchange. Blockchain has been the starting technology that introduced this decentralized approach thanks to the Bitcoin cryptocurrency. In the last few years, DLTs have gained significant attention from several fields of application, that spread across the financial economy down to IoT data management. To this extent, various implementations of DLT have arisen, as figure 4.1 shows an abstract representation of the existing technologies based on the type of data structure.

However, the different technologies all share some common properties that are needed to achieve the desired decentralized infrastructure: public key cryptography, distributed P2P networks, and consensus mechanisms.

As the name suggests, the **blockchain** is populated by *blocks*. The blocks are linked to each other in a linear, chronological order, forming a chain-like structure, with each block containing the hash of the preceding block. In the blockchain, entities that connect to it are referred to as *nodes*. They are responsible for grouping

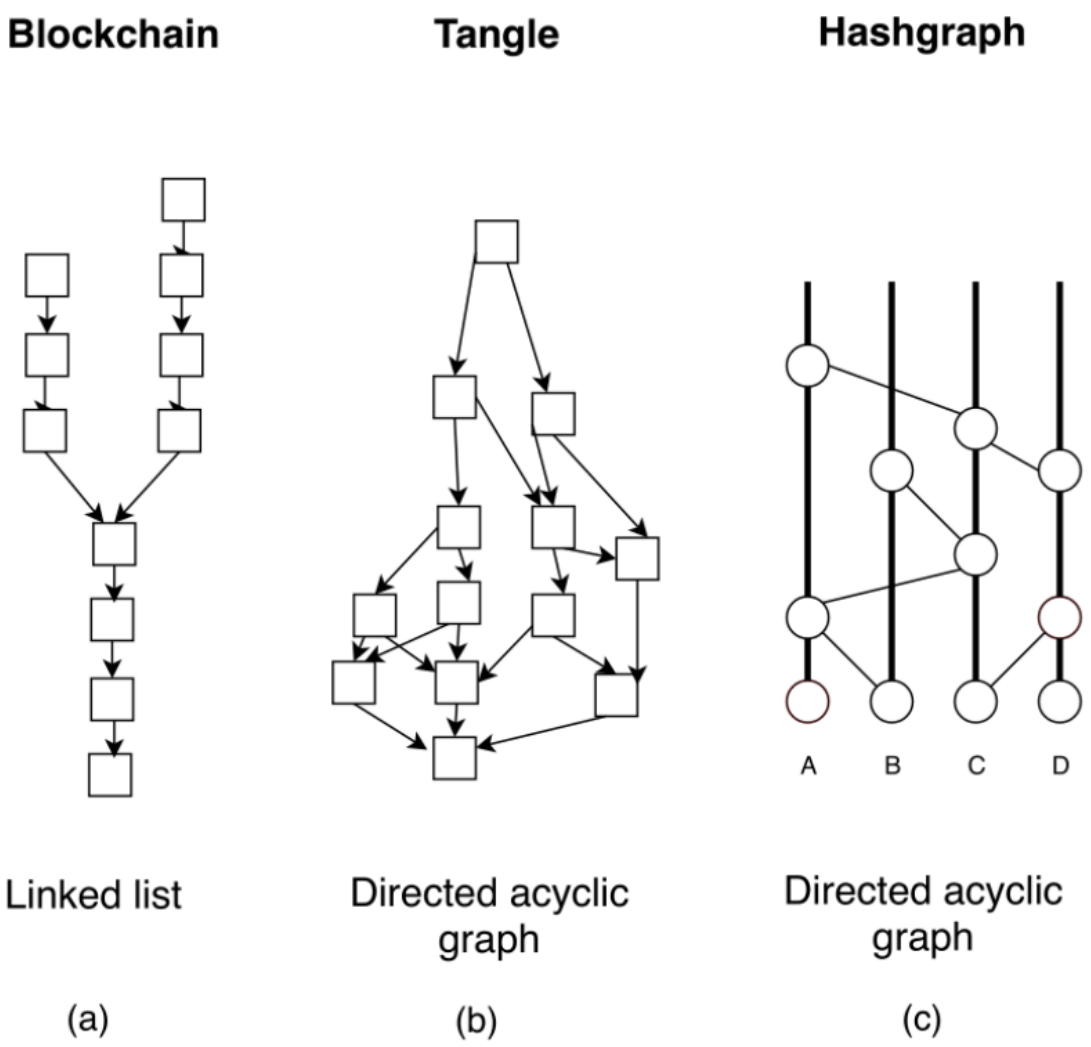

Figure 4.1. Overview of existing DLTs [18].

transactions into blocks and determining their validity, deciding which transactions should be added to the blockchain and which ones should not. To this extent, a consensus mechanism is needed to achieve a distributed consensus among the participant entities. In the Bitcoin blockchain, the consensus protocol used is employed through what is called **Proof-of-Work** (PoW). When a transaction needs to be verified, PoW consists in having some nodes (called **miners**) responsible for solving a cryptographic puzzle to prove that enough computational power has been used. Typically, a node might be required to find a "nonce" that, when hashed along with the transactions and the hash of the previous block, produces a hash with a certain number of leading zeros. The computational effort required for this is exponential with respect to the number of required zero bits, but the verification process is straightforward, consisting of just a single hash calculation. For their given computation effort, miners are rewarded with an amount of the related cryptocurrency. The major downside facing Bitcoin lies in the need to keep the number of added blocks to the blockchain within a tight limit over set time intervals. This limit is set

such that a block can be validated every ten minutes. Hence, it can be considered a bottleneck because applications, like healthcare or IoT, require a higher rate of Transaction per Second (TPS). Additionally, in blockchain technology *transparency* is a drawback because everything on the blockchain is open and visible to all.

Tangle-based DLTs are a decentralized data storage architecture and consensus protocol that utilizes a Directed Acyclic Graph (DAG) data structure. Each node in the graph represents a transaction and the direct edges connecting the nodes represent the validation chain of the transactions. The first implementation of this type of DLT has been developed by IOTA [19]. IOTA's goal is to extend the scope of decentralized ledger technology (DLT) solutions to include both conventional and constrained computational devices, particularly in the context of the Internet of Things (IoT). The IOTA ledger enables fast and fee-free transactions without the need for miner validation, thus all the participants contribute to the DLT by validating transactions. IOTA's consensus protocol operates without a leader and uses a probabilistic approach to validate transactions in parallel, without the need for complete ordering. Hence, the consensus mechanism adopted in this implementation requires a user to perform a PoW that validates two already existing transactions. The PoW is of reduced complexity to allow the inclusion of IoT devices. The Tangle due to its design offers a highly efficient ledger solution.

As for the Tangle, **hashgraph** employs a DAG as its data structure for storing transactions and, differently from the other solutions, utilizes a voting algorithm in conjunction with a gossip protocol to rapidly achieve consensus among nodes. The consensus mechanism enables a much faster replication of data with respect to blockchain implementation. Hashgraphs can be described in terms of columns, where each column represents a user, and in terms of vertices which are events. Thus, a user can submit an event containing a new transaction on the ledger. A user can then *gossip* about a transaction by randomly choosing other users to spread its knowledge about the ledger. This process facilitates a quicker exchange of information about newly submitted transactions. Although Hashgraphs provide the fastest throughput with respect to the other technologies, it is still a pre-mature technology that has only been tested in private environments [20].

Being the Tangle technology a promising solution to cope with the limitation of blockchain technology, we will focus on the IOTA Tangle technology and will give a brief overview of the functioning and the features offered by the IOTA protocol.

4.2 The IOTA Tangle

IOTA [19] is a DLT that enables individuals to take control of their data, execute programs securely without interference, and transact and own assets without the need for intermediaries with the proper implementation of DLT. IOTA operates on the Tangle, a system where recent transactions validate previous ones, while most other DLTs operate on a blockchain. To maintain its state and record, a blockchain must compile transactions into blocks and link them sequentially, which results in a bottleneck, similar to loading the entire world's goods onto a single train, one wagon at a time. IOTA explicitly states that its technologies eliminate the issues

carried out by a blockchain solution [21]. In this work, the ***IOTA Chrysalis*** version (IOTA 1.5) will be considered as the protocol reference.

Decentralized cryptocurrencies, such as Bitcoin and Ethereum, require users to pay a fee in order to broadcast transactions on the network. In contrast, IOTA eliminates the need for miners and does not impose any fees on users. Instead, the amount taken from the sender's wallet is equal to the amount added to the recipient's wallet. This results in a fee-free transaction network. This is an opening for a vast type of IoT applications, where a large number of microtransactions can be submitted with any imposed fee by the IOTA network. Hence, IOTA relies on a purely community-driven platform which is feeless and designed to be more performant as the number of nodes increases.

As mentioned in Sect. 4.1, IOTA relies on an architecture called the Tangle. The Tangle is a decentralized network of **nodes** that replicate a data structure that keeps track of token ownership information. In the IOTA network, nodes serve as the record-keepers and verifiers of all information. Every node has a real-time understanding of the current status and holdings of all addresses in the network, referred to as the ledger state. Additionally, every IOTA node is referred to as an **Hornet** node and is also the entry point for all clients willing to interact with the Tangle.

The Tangle is structured as a DAG of blocks, where each new block is connected to several previous ones. Figure 4.2 shows the interconnections of blocks that make up the Tangle. Each block represents a message and the edges pointing out from it represent the parent messages.

From figure 4.2, we can observe that a block can be a:

- ***genesis*** block, which represents the starting point for the Tangle and serves to initialize it by generating the total supply of tokens. No further tokens will ever be produced. This block is the first one in the Tangle, has no parent transactions, and is marked as solid, and confirmed.
- ***message*** defines the structure and the type of its content through the inclusion of a *payload*. IOTA defines different payloads that allow the representation of various types of messages, such as transactions and many others.
- ***tip*** is a valid solid message which has not been approved yet by any other transaction, and all its directly and indirectly referenced blocks are also valid.

Before diving into the details of an IOTA Message and how it can be published on the Tangle, let us first define how consensus is reached in the IOTA network.

4.2.1 Consensus in IOTA

Consensus in the IOTA network is reached by means of a ***Coordindator***. The Coordinator acts as a client that generates and sends signed messages, called ***milestones***, which nodes trust and use to confirm messages within the Tangle.

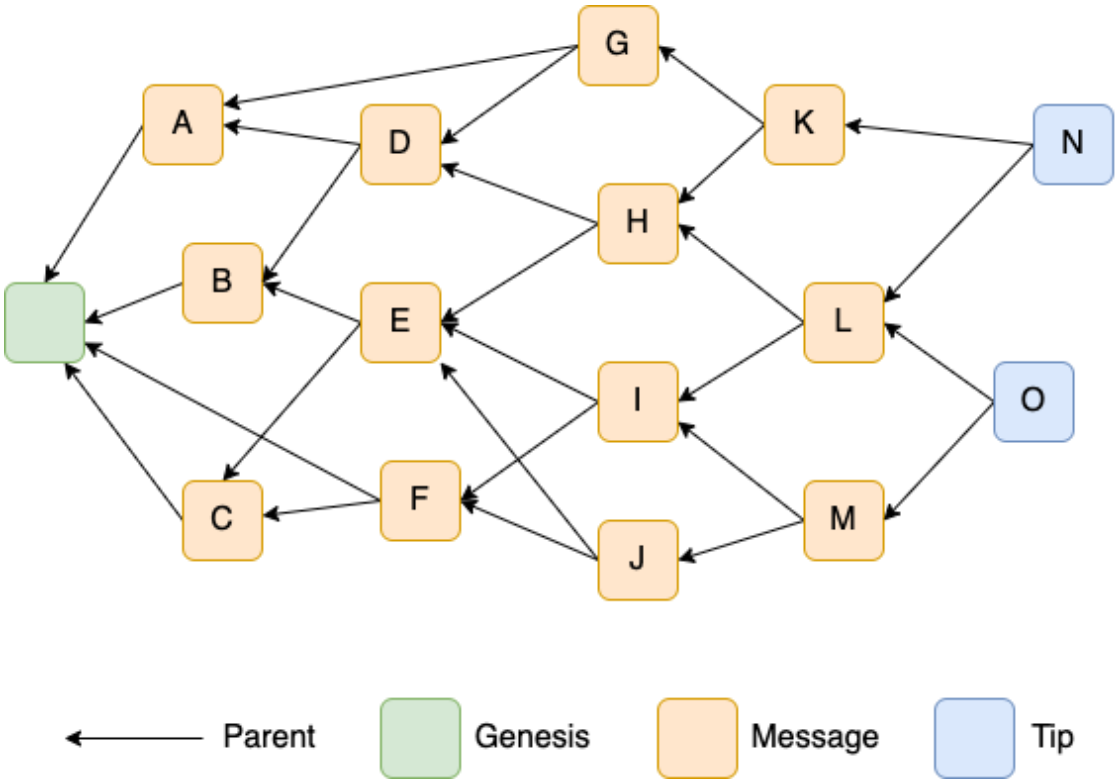

Figure 4.2. Tangle structure overview.

The authenticity of the issued milestone is achieved through Ed25519 signatures. For a message to be considered confirmed, it must be linked either directly or indirectly to a validated milestone from the nodes, as figure 4.3 illustrates.

To ensure nodes can identify valid milestones, all IOTA nodes on a network are configured with the signatures of a trusted Coordinator node. With this information, nodes can validate the signatures in issued milestones and confirm if they were indeed signed by the trusted Coordinator. To give new messages a chance of being confirmed, the Coordinator regularly sends indexed milestones that are signed with its trusted signatures. This process guarantees that nodes can compare the indexes of their milestones to verify their synchronization with the rest of the network. The cadence with which milestones are issued is fixed and is, typically, 10 seconds, but also in case a private tangle is desired, the periodicity can be configured through a specific configuration file.

On the public IOTA ledger, the Coordinators are maintained by IOTA itself. This solution introduces some sort of centralization, which obviously, represents a limitation of the current IOTA technology (IOTA version 1.5). Hence, IOTA declares that this is a temporary solution that will be discarded as soon as IOTA 2.0 (Coordicide) will be published. As of now, the Coordicide protocol is under continuous development and the testing network (Shimmer) has been launched to test the new features that will be introduced.

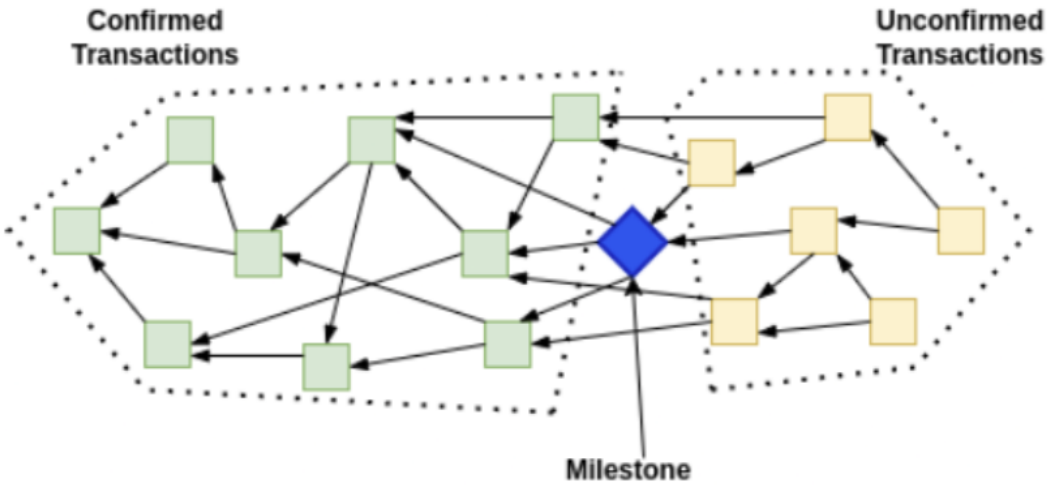

Figure 4.3. Coordinator-issued milestone confirming other messages.

4.2.2 IOTA Messages

In IOTA, different types of messages can be used for different purposes. Some messages transfer IOTA tokens or digital assets, while others only transfer data. Some messages even combine both value and data. This versatile message structure allows for secure, decentralized transport of both value and data without any fees, with network nodes verifying the validity of the messages, and also ensuring secure distribution via the Tangle. A Message is made up of basic information that specifies its type and structure, and may also include various payload types which may vary based on the type of message that needs to be published.

The IOTA protocol defines the syntactical message structure that every message generators (wallets or applications) have to rely on, otherwise, nodes will discard the message. Table 4.1 defines an IOTA message:

Name	***Type***	***Description***
NetworkID	uint64	It is the first 8 bytes of the BLAKE2b-256 hash obtained by the concatenation of the network type and the protocol version string
Parents length	uint8	The number of messages it directly approves (value between 1-8)
Parents	ByteArray{[}32 * parents length{]}	The referenced Message IDs
Payload length	uint32	The payload's length.
Payload	See available Payload types	See details of the payload types
Nonce	uint64	The nonce needed to satisfy the PoW requirement.

Table 4.1. IOTA Message structure. [21]

Once the message has been generated, the **Message ID** is obtained by hashing the entire serialized message by employing a BLAKE2b-256 hashing algorithm. Syntactically, the message is considered to be valid if the following rules apply:

1. The maximum message size is limited to 32 KiB (32 * 1024 bytes).
2. When parsing the syntax structure of a message, there are no unknown bits. Any unreadable information, which could potentially contain harmful code, is rejected for security reasons.
3. The payload type must be known to the node, and the payload itself must be syntactically correct.
4. The Message nonce (PoW) is considered valid if it meets the PoW requirements established by the network or node.
5. The number of parents is between 1 and 8 and must be sorted in lexicographical order, and each one of them must possess a unique Message ID.

As table 4.1 suggests, a message may contain a payload. Table 4.2 shows the IOTA-defined core payloads that can be embedded into a message:

Payload Name	***Type***	***Value***
Transaction payload	0	
Milestone payload	1	
Indexation payload	2	

Table 4.2. IOTA defined payloads. [21]

Transaction payloads are used for *value transactions*, Milestone payloads are used by the Coordinator to issue new milestones, and finally, Indexation payloads are used for *data messages*.

To initiate a token transfer within the IOTA network, a client composes an IOTA message that carries a signed transaction payload. This payload provides all the required details to specify the transfer of a specified number of tokens from address A, owned by the message issuer, to address B, effectively allowing the state update of the IOTA ledger.

For the scope of this work, data transfer is a crucial ingredient, thus the Indexation payload represents a core notion. Hence, IOTA defines an Indexation payload as shown in table 4.3.

The Indexation payload enables the addition of an **index** to the message and the inclusion of additional data. Nodes provide an API that allows for querying messages based on their index. Differently, from the other two payload types, indexation payloads are not sent with a signature and the data contained in them are visible to any peer that knows the message index. Hence, if the privacy of the data is a concern, an additional cryptographic protocol may be needed to protect the payload content. This problematic issue is covered in Sect. 4.3.

Name	***Type***	**Description**
Payload type	uint32	An Indexation payload is denoted with a value of 2
Index length	uint16	The length in bytes of the below index field.
Index	ByteArray[Index length]	The Message's index
Data	ByteArray	Binary data

Table 4.3. Indexation payload structure. [21]

4.2.3 Sending a message in IOTA

A client who has the ability to generate an IOTA message can send it to an IOTA node, which is responsible for processing it. An IOTA node must first check if the syntactic rules against the proposed message: if the syntactic requirements are not met the message will be discarded by the node. Otherwise, the message is considered valid and thus must be attached to two *tips*, and must validate them by employing PoW. The *Weighted Uniform Random Tip Selection* (W-URTS) algorithm is used to select the tips from a pool of tips, that is constructed on every IOTA node. IOTA classifies the tips by assigning a **score** to each one of them. The scoring system is based on the relation between the tip's approval cone (all the messages directly and indirectly referenced) and the previous milestone. Hence, a tip can assume three different score states [22]:

- **0**: the tip is ***lazy*** and should not be selected. IOTA defines them as the tips attached to a sub-tangle where the most recent confirmed messages were verified by an old milestone.
- **1**: the tip is ***semi-lazy*** and can be selected. IOTA defines them as the tips with a single parent connected to a sub-tangle where the newest confirmed messages were validated by a recent milestone.
- **2**: the tip is ***non-lazy*** and can be selected. IOTA defines them as the tips attached to a sub-tangle where the most recent confirmed messages were verified by a recent milestone.

"Non-lazy" refers to a tip that is not attached to a cone of transactions that are too far in the past, as such cones are likely to have already been confirmed and therefore do not increase the rate of newly confirmed transactions when a milestone is released. Hence, in order to increase the confirmation rate, the tip selection algorithm has to return non-lazy tips.

If the node is synchronized with the ledger state, it is asked to randomly select two or more (up to eight) non-lazy tips to approve from its pool of tips. Once the selected tips are approved by the node, they are removed from the tips pool and the message becomes a **new tip** that will be later approved by future incoming messages. Additionally, a PoW computation is needed in order to fulfill the requirements of a valid message. Differently from the Bitcoin PoW, IOTA's PoW (refer to Sect. 4.2.4 for further details) solution is not meant to reach consensus into the

DLT, but only to limit the rate of the network. Once the PoW requirement has been satisfied, the message is broadcasted into the IOTA Tangle to all its direct neighbor nodes through a **gossip protocol**. Additionally, every neighbor receiving the message will again repeat the gossiping process. In this way, every node in the network quickly perceives the message and possesses identical information and understanding of the network's "state" at a specific moment.

4.2.4 IOTA Proof-of-Work

In IOTA, PoW serves to limit the rate of the network and, thus, prevent spamming. Completing PoW by finding a suitable nonce enables you to attach your message to the tangle, but it does not give you the power to determine the truth.

To enhance the efficiency of the PoW, IOTA embraces a two-stage approach that allows a faster validation process. To begin with, the message is processed through the BLAKE2b-256 hash function to generate a fixed-length digest. Then, this digest and the nonce are combined and processed through Curl-P-81. By only computing the digest once and varying the nonce, Curl remains the hash function crucial to the PoW process. In comparison to a complete Curl implementation (which reaches only 2 MB/s on a single core), the validation process is much more efficient, as BLAKE2b-256 has a performance rate of approximately 1 GB/s.

Curl-P-81 is a trinary hash function that accepts **trits** as input. A trit defines a *Ternary system* where a trinary digit, or trit, can assume only three different values (or states): -1, 0, 1. The notion of trit is essential for PoW calculation.

The IOTA network assesses the PoW requirement by verifying that the nonce satisfies a certain amount, called the **PoW score**. The PoW score is determined as the average number of iterations needed to locate the number of trailing zero trits in the hash, divided by the size of the message. Hence, to calculate the PoW score that validates the PoW, the following steps have to be performed [23]:

- Calculate the BLAKE2b-256 hash of the serialized message, excluding the Nonce field, and convert the hash to its 192-trit representation.
- Convert the 8-byte Nonce into its 48-trits representation and append it to the hash obtained in the previous step.
- Add three 0-trits to obtain a 243-trit string.
- Calculate the Curl-P-81 hash.
- From the hash result of the previous step, count the number of trailing zero trits.
- Calculate the **PoW score** as $3^{num.zeros}/size(message)$.

In case of a deployment of a private tangle, IOTA allows an administrator to set the desired PoW score that a message has to satisfy. This may help in speeding up (lower the desired PoW score), or further limit the rate of the network (increase the desired PoW score).

4.2.5 Spammer

In IOTA transactions are confirmed by adding subsequent transactions to the tangle. The more transactions that are added, the quicker previous transactions are confirmed. To this extent, Hornet includes a compact plugin that floods the network with messages, known as **Spammer**.

This plugin can be useful in case the network needs to boost the message submission rate. Within the public IOTA main net, a Spammer may not be needed, but in case a private Tangle is deployed the rate of published transactions may be very low. Hence, spamming the private network can help in increasing the message validation rate. This plugin can also be utilized during a network stress test, in which the community assesses the network's ability to handle high volumes of transactions.

Typically the plugin is disabled by default, thus, an administrator may enable the Spammer plugin through the Hornet node's configuration file. Additionally, the number of messages per second (TPS) that the Spammer tries to send can be also specified.

4.2.6 Snapshot

Every Hornet node attached to the Tangle maintains a local database where all the ledger's information is stored. As the Tangle grows over time, the node's ledger stores every message that is published, leading to disk capacity saturation. To cope with this problem, IOTA introduces the concept of ***local snapshots***.

A local snapshot consists in having a Hornet node record the state of the ledger into a local file. Hence, a snapshot file can be used to represent the ledger starting from the genesis (or an old milestone) up to a recent specific milestone. By taking local snapshots, nodes are able to rapidly synchronize with the ledger by starting from a recent milestone instead of an older one. Additionally, these snapshots allow for the deletion of older messages that are located below the last milestone index included in the snapshot.

Snapshot creation policies can be customized through a configuration file, where this feature can be:

- Enabled or disabled.
- If enabled, the user can specify the maximum number of milestones to keep (e.g. keep a maximum of 300.000 milestones).
- If enabled, the user can specify the maximum desired size for the local node's database.

There exist two types of snapshots:

- A **Full snapshots** captures the entire ledger status up to a designated milestone

- A **Delta snapshots** references a specific full snapshot and includes only the differences since the last full snapshot

However, snapshots may cause some data to be permanently lost as it will no longer exist on the Tangle. Applications that may require the persistence of data on the Tangle for a long time would be limited. To this extent, IOTA provides a special client node that can be deployed to record all the Tangle history and save it into a database. This node's implementation is referred to as **Chronicle** [21].

4.3 Ensuring Data Safety on the Tangle

In order to write and read data over the Tangle, IOTA proposes an implementation of two L2 protocols. They are responsible for structuring the data coming from the lower layer in case a write operation is needed, or de-structuring when the data needs to be retrieved from the tangle. Hence, the objective is to facilitate the transmission and retrieval of data streams. Furthermore, while the Tangle ensures the immutability and integrity of data, an L2 protocol must still provide security measures to secure the transmission of data among peers on the Tangle. To this extent, cryptographic primitives verify the provenance and ownership of data, encrypt and decrypt data, and also have a mechanism to retrieve data streams while ensuring their authenticity.

The integration of an L2 cryptographic protocol and the Tangle offers a secure transport alternative to TLS, allowing for secure, multi-point to multi-point data transfer. This combination creates a trust layer for any decentralized system to exchange data securely in nearly real-time.

IOTA proposes two implementations of such a protocol: *Masked Authenticated Messaging* (MAM) and *STREAMS*. Although they both provide an efficient solution, in this work we adopted a third-party library developed by the LINKS Foundation, named ***Wrapped Authenticated Message*** (WAM) [24]. The reason for this choice stands behind the fact that both of the solutions proposed by IOTA were not adaptable to work on IoT devices due to the programming languages used for their implementations: MAM is written in Javascript, and STREAMS is written in Rust. Moreover, MAM is only suitable to work on legacy versions of the IOTA protocol and thus has been substituted by the STREAMS protocol.

Before diving into the details of the WAM protocol, let us give a brief overview of MAM and STREAMS protocols.

4.3.1 The MAM protocol

The MAM protocol [25] can only be employed in the legacy version of the IOTA Tangle and allows the data publishment by means of transactions over the network. The purpose of MAM is to offer a system for structuring and protecting data streams, ensuring their subsequent verification by any device. For this reason, the concepts of *data channels* and *channel ownership* have been introduced, enabling

only the channel owner to have the power to publish onto the channel. By doing so, it is ensured that attempts by any malicious actors to compromise the channel and inject false information can be detected by either the channel owner or by the (only-read authorized devices).

MAM utilizes a Merkle tree signature scheme [26] to sign the encrypted message's cipher digest. The root of this Merkle tree functions as the channel ID. The previous trees are not referenced, but only the subsequent ones are thanks to the internal inclusion of the next tree's root within each message. When a device publishes data to its channel, it obtains a channel ID, which serves as an identifier that enables other devices to subscribe to the channel and retrieve the data stream.

Channels can be created in three different modes of operations: public, private, or restricted. The address of a transaction containing MAM data in public channels is the root of the Merkle Tree, allowing any device to decrypt the data using the channel ID as the decryption key. In private channels, the address of the transaction holding MAM data is the hash of the root of a Merkle Tree. As a result, only the device possessing the original root can decrypt the data. Restricted channels enhance privacy by incorporating a pre-shared symmetric key. The transaction address containing MAM data is generated by combining the hash of the pre-shared key and the root of the Merkle Tree. As a result, only devices that have access to both the original root and the required symmetric key can decrypt the data.

4.3.2 STREAMS

The MAM protocol has been substituted by STREAMS. STREAMS is an organizational tool for structuring and navigating secure data through the Tangle, and it organizes data by ordering it in a uniform and interoperable structure [27].

Differently from MAM, IOTA STREAMS has been extended to provide not only a "Channel" implementation but an entire framework for cryptographic applications. Hence, the feature of channels has been redesigned and works as a protocol that operates in the STREAMS framework; the Channels protocol. This allows building different solutions on top of the STREAMS framework when the Channel application is not suitable for the specific case. As in MAM implementation, the roles of Publisher and Subscriber remain, but with the difference that also Subscribers can publish *unsigned* messages onto the channel. STREAMS also enhances the message's structure management allowing to publish on the same channel different message formats, whereas in MAM multiple channels were needed for each different message structure.

In MAM, messages were standalone, but in IOTA Streams, messages can reference other messages by linking to them. This allows a message to provide additional information about itself through another linked message. Additionally, in MAM to replace an old message with a new one you had to create a new channel; while in IOTA Streams you can update messages in the same channel. The older message stays in the Tangle to ensure integrity, but applications can access only the latest, valid message. Lastly, STREAMS improves access control for channels by allowing for the customization of cryptographic methods for each message based on its type, facilitating the implementation of specific access rules.

While IOTA Streams offers a robust solution for securing data on the Tangle, its use of the Rust programming language limits its usefulness in resource-constrained environments.

4.3.3 The WAM protocol

As previously stated, the WAM protocol was chosen for its lightweight implementation, making it a suitable solution for the scope of this project. Therefore, WAM is a cryptographic protocol designed to structure and securely read and write data over the IOTA Tangle, specifically designed for use with the Chrysalis version of the IOTA protocol.

A WAM message is encapsulated in the IOTA Chrysalis message, called **indexation** payload (for further details refer to Sect. 4.2.2). The indexation payload is composed of arbitrary data, usually application data, and an index. The index is used as an address, thus is a pointer to the message stored in the tangle.

Figure 4.4 depicts the structure of a WAM message and its encapsulation within an IOTA Chrysalis message.

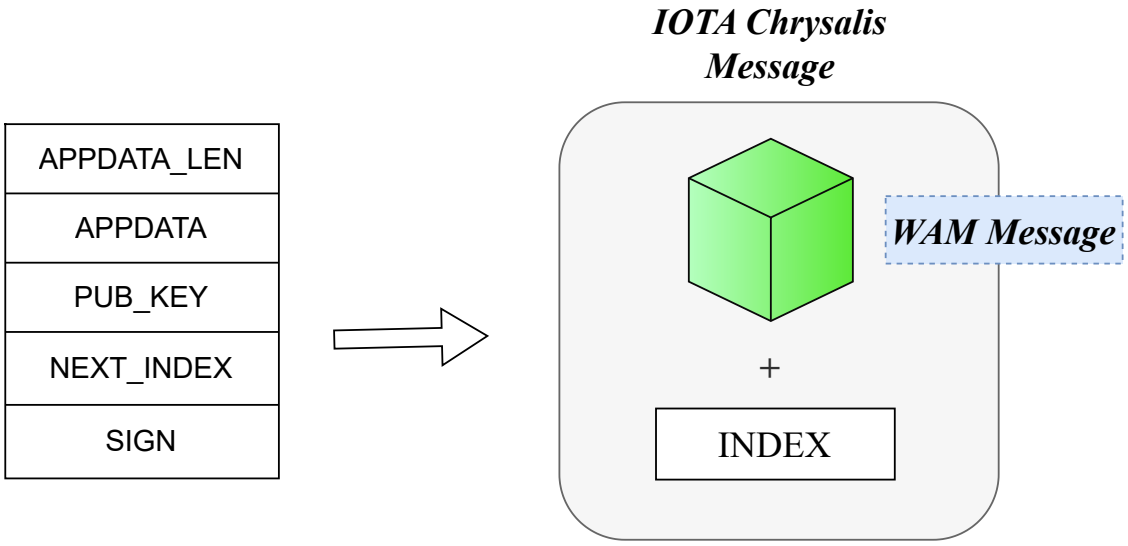

Figure 4.4. WAM encapsulation into an IOTA Chrysalis message.

Higher-level protocols or applications that wish to utilize WAM, will place their data in the APPDATA field and specify its length in the APPDATA_LEN field. In cases where the data exceeds the maximum allowed length for a single message a sequence of linked data must be created. To this extent, WAM structures a data stream as a series of linked elements on the Tangle. Each piece of data is connected to the next one through an index, allowing subscribers to reconstruct the stream by starting at any point and following the chain of linked data. In essence, each data message holds the current index and the index of the next message (the *NEXT_INDEX* field) in the series, thus allowing the chaining of these messages.

By constructing the chain as a single-linked series of messages, subscribers can only read the data stream in a forward direction. This design prevents the retrieval of previous information, as it restricts access to previous messages within the same data stream. The chaining mechanism enables the concept of ***WAM channels***: a single-linked series of messages where only allowed subscribers can read the messages from the chain. Thus, similar to the blocks within blockchains, WAM channels logically represent messages in a linear manner. However, being IOTA a DAG-based DLT, each message within a WAM channel is "spread" across the DLT and chained thanks to the Next_Index included in each WAM message. Figure 4.5 illustrates the WAM chaining mechanism.

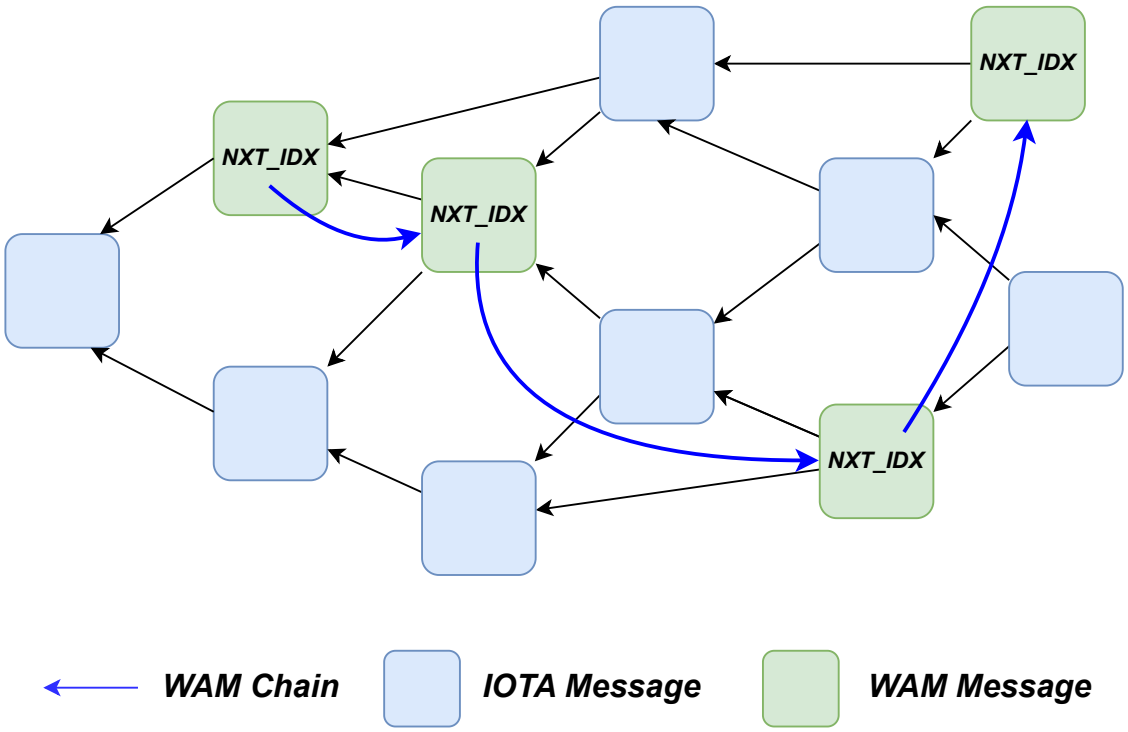

Figure 4.5. Chaining of WAM messages in the IOTA Tangle.

Additionally, WAM is responsible for determining the values of the INDEX and NEXT_INDEX fields. It uses a random source to generate a seed, which is then utilized to create a keypair based on the **edwards25519 curve**. The public part of the key pair is then hashed to produce the index. The value of the NEXT_INDEX field is generated by leveraging the same process but starting from a different key pair. Figure 4.6 illustrates the generation flow of an index.

The WAM protocol has been designed to allow a subscriber to verify that the data in a stream are coming from the same source. This is achieved through the SIGN field. This field is populated by signing a digest h with the PRIV_KEY previously generated. The digest is the BLAKE2b hash of the APPDATA_LEN, APPDATA, PUB_KEY, and NEXT_INDEX fields. Therefore, a subscriber can verify the message by recomputing the same digest and checking the signature with

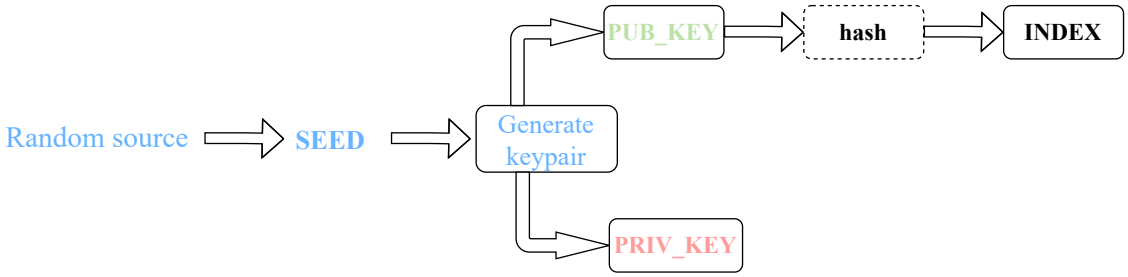

Figure 4.6. Index field generation flow.

the public key contained within the message itself. Moreover, it is possible to also verify that the hash of the PUB_KEY matches the index of the retrieved message. At every message retrieval, those two verifications are performed. It's important to note that an adversary attempting to redirect the next message to a malicious stream would need to discover the public key used to generate the NEXT_INDEX and include it in their message. Therefore, the signature and the public key fields used for verification ensure that no malicious actors can use the NEXT_INDEX to append their chain of messages, as they do not have the key pair used to generate the next index. Figure 4.7 shows how the two verifications are performed.

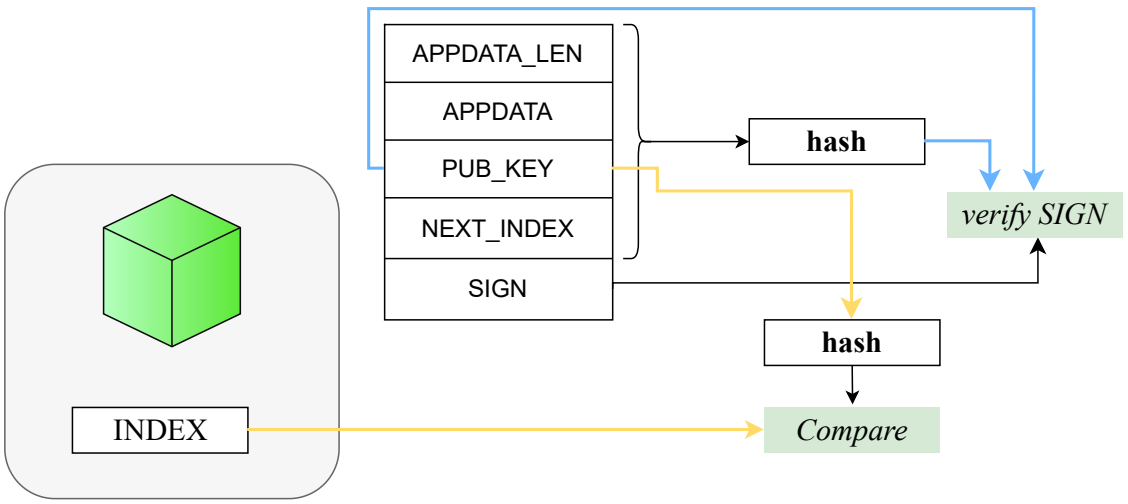

Figure 4.7. INDEX and SIGN fields verification procedure.

Furthermore, every WAM message is encrypted to maintain the confidentiality of the data, which will be made public on the Tangle. The encryption process uses a symmetric cryptographic key and a nonce serving as an initialization vector. The entire WAM message, including all its fields, is encrypted using the XSalsa20 cipher. The encryption key is a Pre-Shared Key (PSK) shared between the communicating parties: the data source and the subscribers. It's important to note that the message chaining mechanism, together with the above cryptographic features, offers forward

secrecy (FS) for the data stream. Even if an attacker gains access to the encryption key, they won't be able to retrieve previous messages in the same data stream.

4.4 DTCB Model

Chapter 2 discussed the properties that a system's TCB must possess to align with the principles of Trusted Computing. These properties can be expanded to define the fundamental characteristics of a DTCB. A system can be considered DTCB-enabled if it meets the following requirements [3]:

1. ***Group Membership***: the DTCB is comprised of TCB nodes that meet specific membership criteria. Membership is enforced to prevent non-TCB nodes from joining and to expel compromised or non-compliant nodes from the group. The use of distributed consensus algorithms may be exploited as a method for enforcing group membership.

2. ***Truthful Attestation***: all the nodes forming the DTCB must be able to accurately report the integrity status of their hardware, software, and configuration.

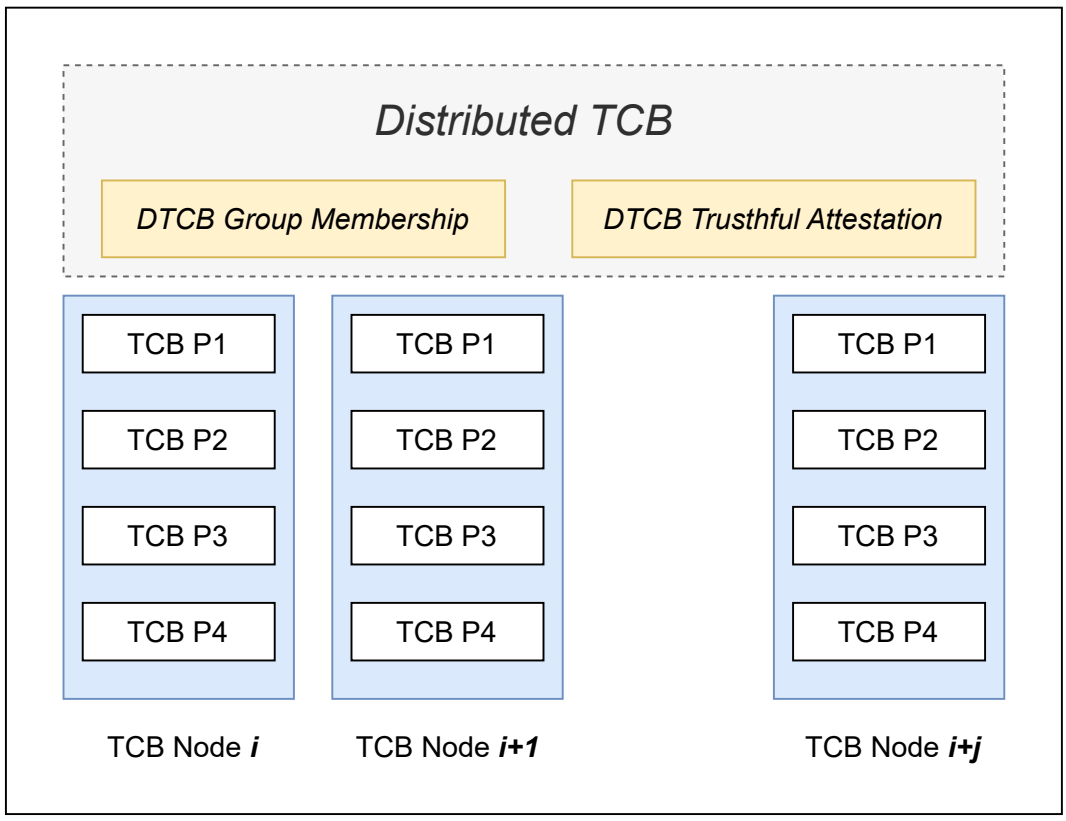

Figure 4.8. Abstract illustration of a DTCB [3].

To be considered a DTCB-enabled node, it must first meet the properties required for it to function as a TCB, as figure 4.8 illustrates. The property of **Truthful Attestation** enables a remote entity to challenge all the nodes to report about their internal state and verify if any node can be considered DTCB-compliant, thus enabling RA. In the context of this work, the keyword is **decentralization**. Hence, there should be no central entities that are enabled to act as Remote Attestors within RA protocol executions. As a consequence, every DTCB node must act both as a Verifier and as an Attester, enabling any DTCB node in the network to evaluate the trustworthiness of all active nodes on its own employing a mutual RA protocol. Every peer will then be structured with what we refer to as **Local Attester** (LA) and **Local Remote Attestor** (LRA) and, thus, have the capabilities to not only produce integrity reports but also have the capabilities of verifying such generated reports from the other peers. By leveraging the common technologies, a mutual RA protocol would require a node to open multiple connections towards all the other nodes. Hence, in order to collect all the attestation reports generated by the other peers, a secure connection needs to be established for each node present in the network willing to act as a DTCB node. Figure 4.9 (a) illustrates the scenario just described, in which a full-mesh topology would be required if classical technologies, such as TLS connections, were to be used.

The limitation of this scenario consists of the complexity driven by the number of connections needed to link all the peers among them: the number of connections needed grows proportionally to the square of the number of nodes. If N is the number of peers in the network and C is the totality of the needed bi-directional connections, C can be obtained by applying the following formula:

$$C := N * (N - 1)/2$$

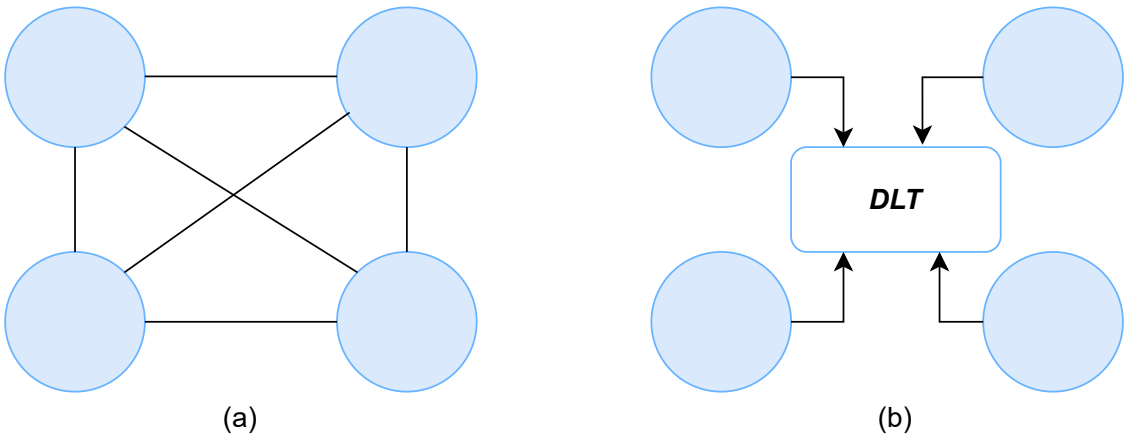

Figure 4.9. Full mesh topology (a) vs. DLT-based topology (b).

To address this issue, figure 4.9 (b) illustrates that nodes have the option of establishing a **single connection** to a DLT. This arrangement enables a node to gather all necessary attestation reports on its peers for verifying compliance with

the DTCB, while only having to open one connection. Due to its properties, the DLT can act as an immutable data storage, where the information can be easily submitted and retrieved by any peer. The usage of a DLT seems to offer nice properties that meet the requirements needed for a DTCB to be implemented. The IOTA Tangle (refer to Sect. 4.2 for further details) will be used as a means for storing peer data and will enable data exchange between them. As already mentioned, data are sent to the ledger by structuring them in IOTA Messages which internally have an *indexation payload* attached to its structure. Once an IOTA Message is submitted onto the Tangle and has been gossiped to all the peers, the data contained inside each message are public and thus are readable and retrievable by anyone that knows its Index. To overcome this limitation, the *WAM protocol* (refer to Sect. 4.3 for further details) will be employed to also ensure on-Tangle data confidentiality. Hence, a Tangle-based mutual RA protocol is proposed and it will be applied in the context of a ***permissioned Tangle*** where only a restricted group of nodes will participate in forming the DTCB. A private Tangle is deployed in this project to limit its scope and improve testing. If the public IOTA main net were used, authorized nodes could be isolated by using the PSK, allowing only those with the key to encrypt and decrypt messages on the Tangle.

Additionally, the **Group Membership** property must also apply, allowing the DTCB to evaluate the TCB compliance of the participating nodes. To this extent, a *group-oriented consensus protocol* may be employed to decide which non-TCB nodes have to be expelled from the group and also to ensure that non-compliant and compromised nodes become part of the DTCB. Hence, a node's capability to accurately report its internal status enables it to demonstrate its active participation in a group-consensus computation. Consensus is crucial in constructing a DTCB as it helps to prevent compromised nodes from lying about their internal state and falsifying the results of the other DTCB node's verifications.

In other words, a DTCB can be seen as a group of nodes that belong to the **same trust domain**, where they all cooperate in a decentralized mutual attestation process in order to maintain a distributed state of trust, and it is enforced by a **group-oriented consensus protocol** to regulate nodes compliance inside the DTCB itself.

Chapter 5

DTCB Implementation

In this chapter, a proposal of a *Proof of Concept* (PoC) implementation is presented to meet the requirements of the DTCB model described in section 4.4. Hence, a customized (RA) protocol is proposed to facilitate the mutual exchange and verification of attestation data among the nodes forming the DTCB, allowing the verification of such reports to be performed by the same group of nodes responsible for exchanging the attestation data. Additionally, a consensus protocol is proposed that aims at maintaining a distributed state of trust among the DTCB participants.

The communication among the DTCB nodes is facilitated by utilizing the IOTA Tangle, with a private tangle deployed specifically for testing purposes. The private deployment of the Tangle also enables customization of its configuration, providing the flexibility to optimize the network according to specific requirements without being limited by the constraints of the public main network (e.g. PoW complexity).

First, an overview of the deployed private tangle infrastructure is given, followed by an explanation of the implemented PoC.

5.1 Private Tangle Infrastructure

To have a fully functional private tangle based on the Chrysalis protocol, a regular Hornet and a Coordinator node are needed. Hornet nodes can be directly compiled from the source code which is public on the IOTA official Github repository. However, to simplify and automate the deployment of a Chrysalis private Tangle, IOTA provides Docker-based automated tools for deploying regular Hornet nodes, Coordinator nodes, and other types of nodes such as Spammers. Therefore, each node can run within a Docker container, and the essential nodes mentioned above are all built starting from the same Docker image available on the Docker Hub registries. To instruct the behavior of a specific *Hornet-based container*, configuration files have to be defined before launching the container. Afterward, they will be mounted within the containerized application by leveraging Docker-based tools. These configuration files, which are in JSON format, specify the type of node to be launched together with all its relevant parameters.

Figure 5.1 illustrates the general architecture for the deployment of a private tangle using Docker together with the three main nodes needed for its correct

functioning. A Spammer node is essential for achieving a higher throughput in a tangle in which there is a low affluence of new messages and consequentially a low number of tips to be approved. Hence, each node is executed inside a Docker container, and every one of them is connected to the same Docker network.

As soon as a Hornet node is started for the first time, it will be associated with a unique identifier generated by the node itself called ***PeerID***. The identity of the node is represented by an asymmetric key pair and its peerID is obtained by hashing the public part of the key pair. This allows a verifiable linkage between the given peer and its public key, also enabling secure communications among the various peers as they can use the hash to verify a peer's identity. The identity-related data of a given node are locally stored in a file and should never be disclosed to anyone, while the peerID is public. Once the PeerID has been generated, Hornet will use it also between subsequent restarts.

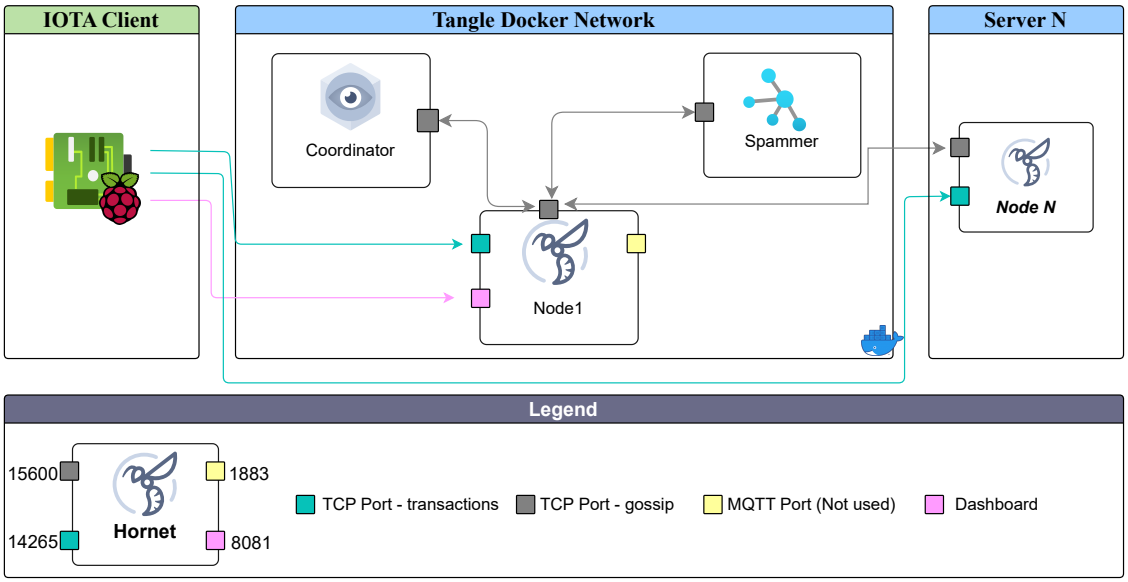

Figure 5.1. Private tangle infrastructure overview.

To be functional, a private tangle must at least deploy three main nodes:

1. The ***Coordinator*** which is responsible for periodically emitting milestones. The Coordinator bootstrap process expects to generate an Ed25519 key pair which is used to sign the emitted milestones. The key pair is stored in a local file that has to be protected and not disclosed. The public key is also stored in a second local file which is passed to every instantiated node to let them verify the Coordinator's milestones.

2. The ***Spammer*** sends messages to the Tangle at regular intervals, enabling the minimum message load necessary for transaction approval in accordance with the IOTA protocol.

3. The ***Regular Hornet Node*** is responsible for processing the requests coming from the clients (e.g. validate messages, perform PoW, etc.). Additionally, it may also need to connect with other nodes that can be later attached to the Tangle.

Figure 5.1 also shows what type of communications are needed inside the Tangle infrastructure. All the nodes involved leverage the *TCP port 15600* to exchange gossiping information, allowing them to quickly update the ledger state and also to inform if new peers have joined the tangle. Additionally, regular Hornet nodes also provide access to REST APIs through *TCP port 14265*. Being the container executed inside a virtual environment, it may be needed to map the ports onto the available machine ports and enable the traffic forwarding from the machine port to the correct container port. This also protects against exposing nodes like the Coordinator, which should not interact with external clients. Hence, only regular Hornet nodes should expose the REST API's port onto the hosting machine. This allows any client, that can communicate with the hosting machine, to make requests for tasks such as sending messages on the Tangle, reading messages, and others. Additionally, access to exposed REST APIs can be restricted by enabling JSON Web Tokens (JWT) [28] authorization. For a more friendly way to interact with a node, a *dashboard* can be obtained by exposing the *TCP port 8081*, allowing to navigate Tangle information through a visual interface.

The Tangle can be bootstrapped in two ways: either by using an existing *snapshot* file or by generating it on the fly starting a Tangle completely from scratch. Once everything is set up, any client can freely interact with the IOTA Tangle.

Additional regular Hornet nodes can be easily added to an already existing tangle by instructing the new node with the correct peering information needed to reach the other peers (nodes). To this extent, Hornet utilizes the ***MultiAddress format*** (*multiaddr*) for combining multiple layers of address information into a single path structure. After obtaining your node's *multiaddr*, it can be shared with other node owners to establish mutual peer connections. For example, a node that is reachable using IPv4 at address 192.168.0.1 on port 15600 and knowing its PeerID, the *multiaddr* encoding would look like it is shown in 5.2.

```
/ip4/192.168.0.1/tcp/15600/p2p/(PeerID string)
```

Figure 5.2. Example of encoding peer information with *multiaddr*.

It is recommended to have 3 to 6 peer neighbors for increased redundancy. As already mentioned, every node must be equipped with the Coordinator's public key to ensure the correct verification of every emitted milestone. Hence, every new instance of Hornet nodes must be supplied with the correct Coordinator's cryptographic material. Additionally, for each new node that joins the Tangle, a snapshot file must be supplied to initiate the synchronization process. This enables the node to quickly catch up to the current state of the Tangle, allowing it to participate in it effectively.

5.2 Proof of Concept Implementation

This section provides a high-level overview of all the operations that have to be performed in order to implement the proposed DTCB model, which comprises a *Tangle-based RA* together with a *group-consensus protocol*. The fundamental principles of RA have been taken into account to design a custom RA protocol (refer to Sect. 3.3 for further details) that allows a subset of nodes to exchange attestation reports among themselves by leveraging a *permissioned IOTA Tangle* together with the *WAM protocol* for secure data exchange. The proposed protocol behaves in **decentralized** fashion, allowing all the nodes that participate in forming the DTCB to mutually attest to each other without the need for any central authority. With the support of a group-oriented consensus mechanism, it is possible to maintain a distributed state of trust among the DTCB nodes. Hence, the goal is to implement all the functional requirements described in Sect. 4.4.

To avoid ambiguity, we will refer to the client nodes as simply "nodes" and refer to the IOTA nodes responsible for handling the requests of these nodes as "Hornet nodes".

In this work, **RaspberryPi** devices have been used to deploy the application. To support the correct functioning of the implementation, every device must be equipped with a functioning TPM. The TPM-based operations performed in the PoC have been developed by leveraging the APIs offered by TSS. From the available software stack, the ESAPI have been chosen because they offer all functionalities exposed by the TPM with a higher layer of abstraction with respect to the SAPI (refer to Sect. 2.3 for further details). The entire PoC has been implemented using the C language to achieve a lightweight application and to have the possibility to deploy it on any IoT device. A modular approach has been adopted, which separates all the core components into independent units that can be deployed without any interdependencies. This design enables greater flexibility, scalability, and maintenance of the application. As mentioned in Sect. 4.4, every node consists of two major components, identified as the ***Local TPA*** (i.e. Local Attester) and the ***Local Remote Attestor*** (i.e. Verifier). The local TPA module will be responsible for correctly generating the TPM-related cryptographic material and the attestation reports that have to be published onto the Tangle. The role of the local Remote Attestor module, instead, is to verify reports submitted by all peers other than itself and to actively participate in the group consensus protocol. Every node will be correctly set up to execute the whole application, thus acting both as the Attesting system and as a Remote Attestor. By avoiding interactions with a central authority, this approach facilitates a fully **decentralized** implementation, which is a key requirement for a functional DTCB.

From the described RA protocol in Sect. 3.3, the Remote Attestor acts also as the initiator of the protocol, by challenging the Attesting system to report its current state. The challenge is composed of a nonce, that ensures the freshness of the quote, and of a list of PCRs that the attester should include in the quote operation. But within a P2P network, every node's local Remote Attestor should challenge all the other nodes participating in forming the DTCB. This results in a high-complexity solution because the number of required challenges grows proportionally to the square of the number of nodes. To minimize the complexity of

this process the concept of ***Heartbeat node*** is introduced. As the name suggests, a Heartbeat node is a system that produces some information at a specific periodic rate, just like a human's heart does. Hence, one of the participating nodes will be selected by the administrator to act also as the Heartbeat node. It will be responsible for providing, in a periodic fashion, a nonce on the Tangle making it accessible to all DTCB nodes for inclusion in the quote to guarantee freshness. When the Heartbeat node publishes a new nonce, all local TPAs are responsible for generating new attestation reports, and all local Remote Attestors are prepared to evaluate these reports. The Heartbeat node thus controls the frequency of all the *Attestation cycles* throughout the lifecycle of the DTCB.

Hence, for every Attestation cycle, each local Remote Attestor collects all the attestation reports which are composed of a quote signature, signed with the AK_{pub}, together with the "quoted" structure and the Attesting system's IMA ML. For every attestation evidence, the local Remote Attestor verifies the signature of the quote. If successful, it validates the received quote's structure and finally evaluates the IMA ML against the whitelist which was previously retrieved from the Tangle. If the evaluation process successfully terminates, the node related to the analyzed evidence is considered to be ***Trusted***, otherwise, it will be marked as ***Untrusted***. By executing the verification process over all the DTCB attestation reports, the local Remote Attestor can construct its own table that encapsulates the trust evaluations of the other DTCB nodes. We refer to this table as the ***Local Trust-Status table***. Once the table is built, it will be published on the Tangle making it available to all the other DTCB nodes. Therefore, during each Attestation cycle, every DTCB peer will publish its own Local Trust Status table. These tables can then be utilized as inputs for a ***group-consensus protocol*** to construct the ***Global Trust-Status Table***. Every DTCB node will be instructed to perform the same consensus mechanism. Through this method, each DTCB node will be able to generate the Global Trust Status table by utilizing the same inputs, resulting in consistent and uniform outcomes across all nodes. This protocol guarantees the detection of compromised DTCB nodes, enabling the DTCB network to remove non-compliant nodes and maintain a distributed state of trust among all DTCB peers.

Assumptions

The strong assumptions that have to be taken into account are the following:

1. Each node is assumed to boot in a trusted manner, hence, the Trusted Boot process is not enabled, and the corresponding PCRs are not used during quote operations.

2. The WAM protocol utilizes a PSK to encrypt the data embedded into every indexation payload, the WAM message. The distribution of this PSK is assumed to be performed off-chain in a secure manner, preferably by using secure protocols like the Diffie-Hellman key exchange protocol [29].

3. The AK is assumed to come from a genuine TPM where its corresponding EK resides in the same hardware RoT. As a means of simplifying the deployment

of this proof of concept and avoiding the need to establish a CA during the testing phase, AK certificate provisioning has not been incorporated.

It is worth highlighting that every message shared by the nodes in the Tangle is assigned a unique identifier based on its corresponding node ID. By using the SHA256 hashing algorithm the node ID is computed as follows:

$$NodeID = H(AK_{pub})$$

5.2.1 Off-chain Operations

Before participating in the mutual RA protocol and interacting with the Tangle, it is necessary for every node to undergo some initial setup procedures. These preparations ensure the smooth operation of the application and enable the node to effectively communicate with the Tangle and other DTCB peers.

Hornet nodes expose a REST API to allow any client to request the retrieval of a Message given its Index. Also, WAM constructs a chain of Messages starting from an Index and chaining them by calculating also the Next Index, as explained in Sect. 4.3. Therefore, in order to access data published by another node, each node must be aware of the starting indexes at which the other nodes initiated their chain. This allows the node to subscribe to the publisher's chain. To this extent, the administrator must generate what we refer to **Indexing Files**: they contain all the indexes that a node must know in order to write and read data to/from the Tangle. Every node must be equipped with two Indexing files: one for the local TPA module and one for the local Remote Attestor module.

The local TPA module will need a file containing:

1. A node, as the file owner, will use an index to begin writing its attestation reports. To ensure the proper structure of the WAM message, the key pair associated with the index must be known to the node and saved within the file.

2. An index, together with its key pair, to write the verification policies (i.e. **whitelist**) to allow the Remote Attestors to truthfully evaluate the software status.

3. An index, together with its key pair, to write the AK's public part to allow the Remote Attestors to verify the signature of the resulting quote operation.

4. An index to read the nonces published by the Heartbeat node. The index is saved together with only the corresponding public key that was used to generate it.

Instead, the local Remote Attestor module must be equipped with a file containing:

1. A list of indexes to read at the correct index of the attestation reports published by other peers. These indexes are only used for reading, hence, only the public part of the key pair must be known to correctly verify a WAM message. Then, the file will contain a list of "index-publicKey" pairs.
2. An index to read the nonces published by the Heartbeat node.
3. A list of read indexes needed to access the verification policies (i.e. **whitelist**) published by the other peers.
4. A list of read indexes needed to obtain the public part of the AKs owned by the DTCB peers.
5. An index, together with its key pair, to write the Local Trust Status table.
6. An index to read the Local Trust Status tables submitted by the other DTCB peers.

Once the administrator generates the Indexing files, he distributes them off-chain in a secure way. The implemented application provides an automated script that helps the administrator in creating the correct files. For more readable content the "Indexing files" are generated following the JSON format [30].

Additionally, every node must generate the required cryptographic material to perform RA. To this extent, the node must create an EK with its corresponding AK and persist them inside the TPM's Non-Volatile memory. Also, a local file is generated which contains the two NV-Indexes values to be able to retrieve them when necessary. It is assumed that the lower NV-index references the EK, while the highest one references the AK. For this task, the application provides an automated script that generates and stores the two keys according to the TCG's guidance.

The last step consists in generating the so-called *whitelist*. The application provides a Python script to allow an administrator to easily generate the whitelist file. The content will be organized as a list of "filename-digest" pairs, which the Remote Attestors will use for comparison with the matching entries in the IMA ML. It is of crucial importance that the client generates the whitelists in a secure and isolated environment. This also enables a **distributed** whitelisting approach.

5.2.2 On-chain Operations

Once all preparatory steps have been completed, every node is ready to participate in the formation of the DTCB. Every node will execute both the core needed modules: the local TPA and the local Remote Attestor modules.

Local TPA Operations

In order to read and write the necessary information, each local TPA must be able to access its corresponding indexing file, which contains the relevant indexes. The local TPA first checks that the EK and AK have been previously generated, and

also it tries to publish on the Tangle the public part of the AK (AK_{pub}) together with its whitelist. To be part of the DTCB, a node must first have a pre-existing whitelist. If this requirement is not met, the node will be unable to participate. However, if the node does have a whitelist, it will be able to publish both its AK_{pub} and its whitelist. The node is now ready to subscribe to the Heartbeat channel and wait for a new nonce to be published. Once the nonce has been published, it triggers the local TPA to proceed in performing the Quote operation. The local TPA has been statically instructed to include only the PCR10 inside the quote request that will be sent to the TPM. The PCR10 will contain the aggregate value of all the Extend operations performed over the IMA ML, as explained in Sect. 3.1.2. The quote structure together with its signature is returned by the TPM after a successful Quote operation. The local TPA then retrieves the IMA ML from the security filesystem and together with the TPM Quote's output it constructs the IR (Integrty Report). The IR is then published on the Tangle at the index specified in the node's Indexing file. The chaining mechanism is transparent to the application, it is the WAM library that takes care of correctly chaining subsequent IRs submissions. After publishing the IR on the Tangle, the local TPA waits for the next nonce to be published before restarting the process from the beginning. It is important to note that the size of the entire IMA ML may grow indefinitely over the Attestation cycles. This could slow down the writing process onto the Tangle. To this extent, the full IMA ML is stored inside the IR only in the first Attestation cycle, while in the following cycles, only the **difference** between the current IMA ML and the previously sent ML is incorporated inside the new IR. This enables a faster writing process and, also, enables the local Remote Attestors for a faster verification process.

This whole process performed by the local TPAs is designed to be repeated indefinitely, but there is a possibility that a node may be excluded from the DTCB group as a result of the group-consensus protocol. If this occurs, the node's local TPA can still publish its IRs, but they will no longer be evaluated by the other DTCB peers.

To summarize, each local TPA among the DTCB nodes performs a Quote operation and constructs an IR upon receipt of a new nonce published by the Heartbeat node as long as it's DTCB-compliant, as figure 5.3 illustrates.

Local Remote Attestor Operations

The local Remote Attestors expect to read from the Tangle the necessary information needed to validate the IRs that will be published by the local TPAs. This information includes the AK_{pub} and the whitelist of all the nodes participating in the DTCB. The local TPAs are responsible for publishing them. Each local Remote Attestor utilizes its Indexing file to know at which index these data are stored on the Tangle, and once they are retrieved it constructs a table where each row contains a tuple of AK_{pub} and whitelist. This allows the usage of the correct key for the verification of the quote's signature and the usage of the correct whitelist for the evaluation of the node's software state trustworthiness. The construction of this table must be successful otherwise the correct verification of all the IRs will not be possible. The local Remote Attestor is designed not to retrieve from the Tangle

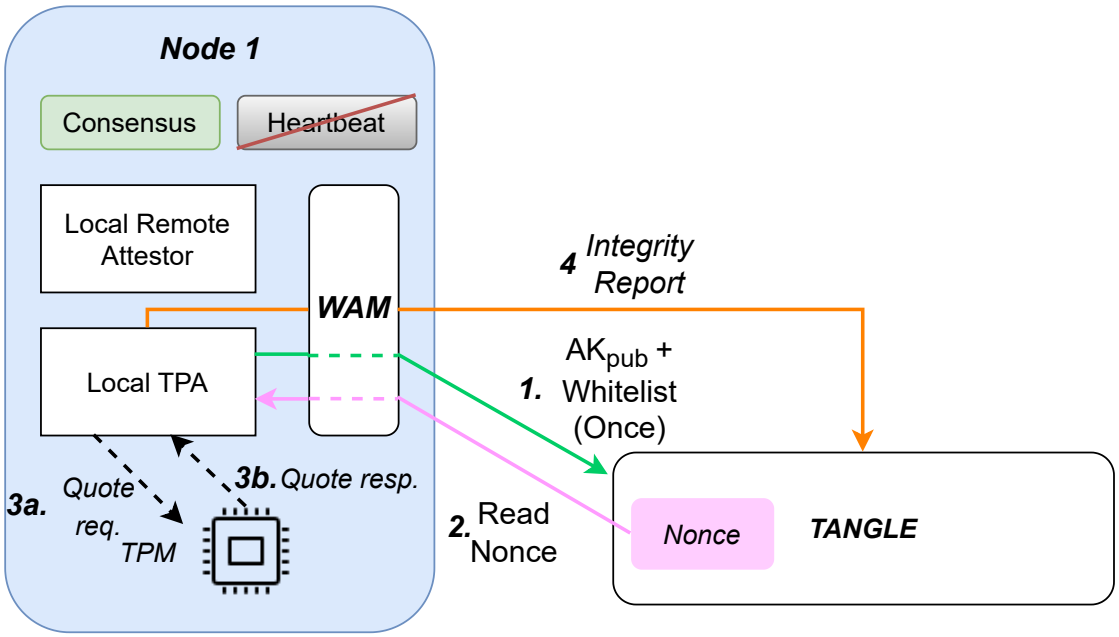

Figure 5.3. Local TPA operations performed at each Attestation cycle.

the data associated with the local TPA that is running on the same machine, as highlighted in figure 5.4.

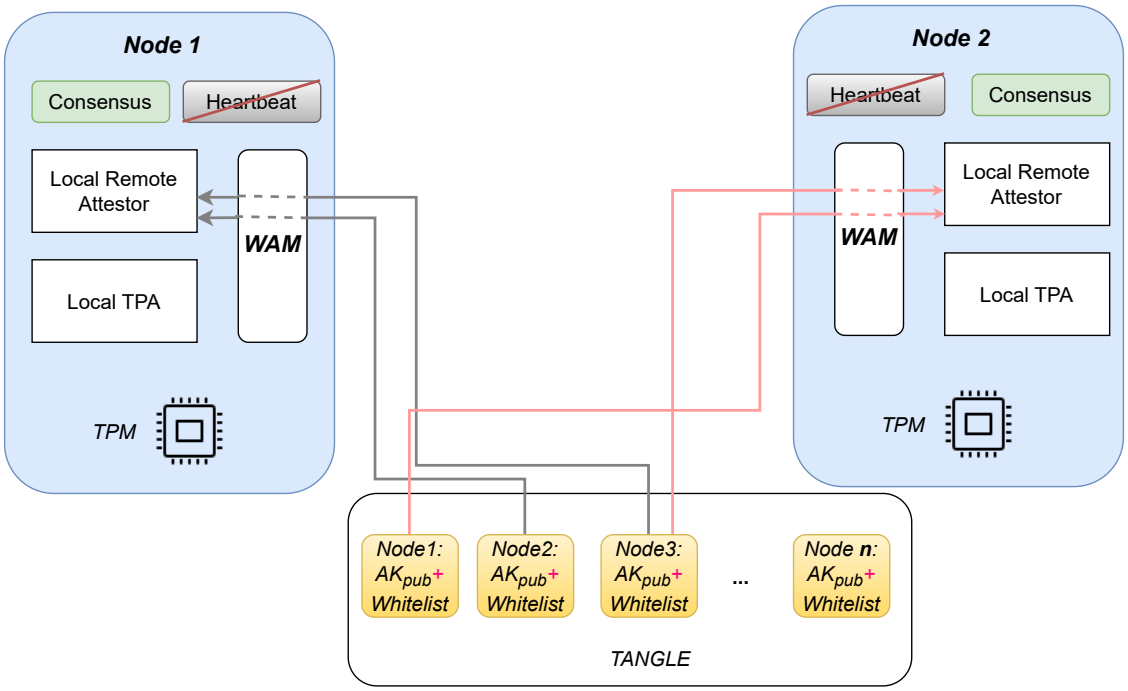

Figure 5.4. Local RAs retrieving the other DTCB nodes whitelists and $AK_{pub}s$.

Once this step has been successfully terminated, the local Remote Attestor is ready to retrieve the IRs from the Tangle. The Indexing files denote the starting indexes at which the local TPAs will write their attestation reports, allowing every local Remote Attestor to subscribe to the correct WAM channels. The local Remote Attestor will need to verify the IRs published by the other nodes on the Tangle except the IR generated by the local TPA that is running on the same device as the local Remote Attestor. Hence, if N is the number of DTCB nodes every local Remote Attestor verifies $N - 1$ reports. That is because an adversary who gains control of a device will typically attempt to present itself as a Trusted entity. As a result, the assessment of a device's trustworthiness is delegated to the other DTCB devices.

The verification process of a single IR is executed as explained in Sect. 3.3:

1. The quote's signature is validated.
2. The PCR10 aggregate value is recalculated, and a digest of the resulting value is also computed (the *calcDigest*).
3. The calcDigest value is compared with the digest included in the received quote's structure. This ensures that the IMA ML has not been modified by a malicious actor.
4. If all previous steps successfully terminate, it finally assesses the trustworthiness of the runtime measurements contained in the IMA ML by comparing them to the integrity measurements contained in the whitelist.

Based on the results of the verification processes, the local Remote Attestor is able to state whether the system of each Attester is in a Trusted or Non-Trusted state. Each local Remote Attestor will then store its trust decisions in the ***Local Trust-Status table*** and will publish these tables on the Tangle when all the $N-1$ reports have been evaluated. Figure 5.5 illustrates this behavior from the point of view of a single DTCB node.

Every node will be then able to read the trust decisions made by the other DTCB nodes that can be used as inputs to the ***group-consensus protocol***. The consensus protocol will be executed by every DTCB node and will output what we refer to as ***Global Trust-Status Table***. This process allows the detection of compromised nodes. In the case in which a Non-Trusted device is detected, every local Remote Attestor will not consider any attestation report coming from the infected device. In addition, each Remote Attestor will unsubscribe from the infected device's WAM channel, preventing it from receiving any further messages related to the untrusted device. The Remote Attestor of the untrusted device determines that it is no longer part of the DTCB group by examining the trust tables published by the other peers. If it finds that the other peers have not evaluated its IR and have therefore not included it in the consensus protocol, the Remote Attestor will understand that it is no longer considered part of the group and will stop its execution.

Also, the Heartbeat node will read the submitted Local Trust-Status tables and will participate in the consensus mechanism. In this way, also the Heartbeat

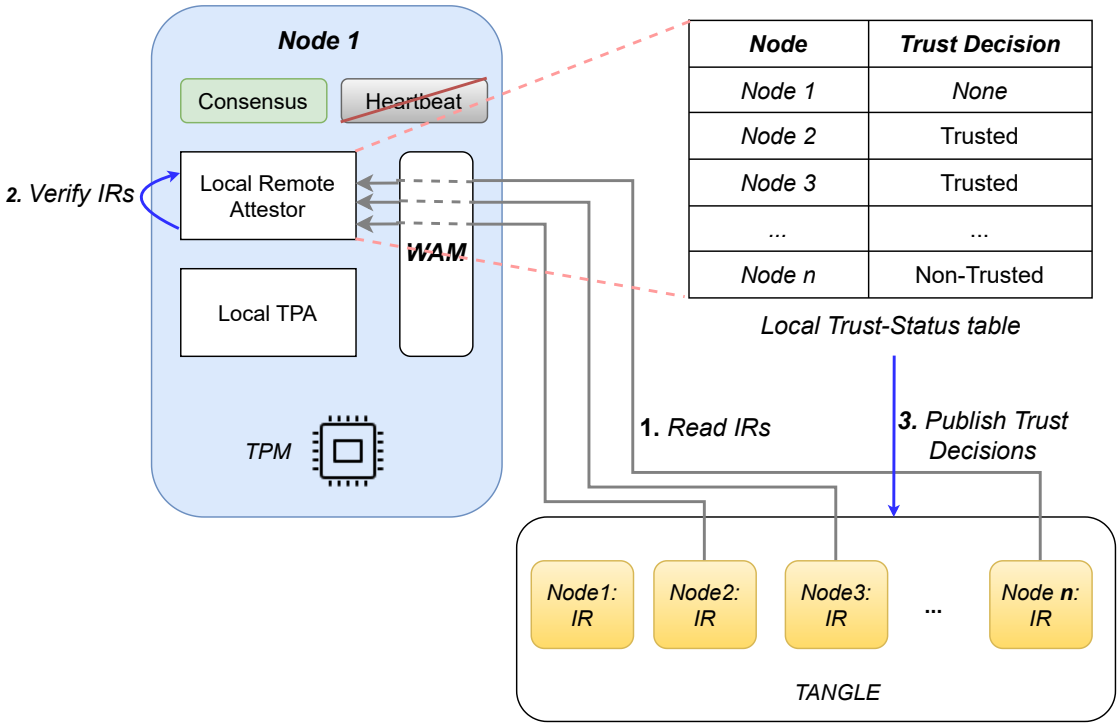

Figure 5.5. The Local RA verifies the IRs of other nodes and then publishes the local Trust-Status table.

instance will have a complete view of the trusted DTCB nodes. After the consensus mechanism terminates the Heartbeat node is ready to publish a new nonce, starting a new instance of an Attestation cycle.

5.2.3 Group-Consensus Protocol

The proposed protocol allows the detection of compromised nodes only in some specific cases, which we will illustrate in this section. Every single node must adhere to the Attestation procedure, otherwise, it will be expelled from the DTCB group. If this is not the case, a single node reads the local Trust-Status tables published by all the other peers. The consensus protocol takes as input a collection of trust tables, which includes the local trust table of the "host" node as well as the tables of all other nodes that have been obtained from the Tangle. The node's local Remote Attestor is responsible for executing the consensus mechanism and providing it with the necessary inputs.

The consensus logic executes the following steps:

1. Scans all the local Trust-Status tables to initialize the global Trust-Status table by assigning a row to the distinct NodeIDs.

2. It calculates the consensus rule, which varies based on the number of DTCB participants.

3. It determines the Trusted and Untrusted nodes by applying the rule obtained from step 2 and populates the global Trust-Status table.

4. It returns the global Trust-Status table to the local Remote Attestor.

This mechanism works on the assumption that non-corrupted nodes are able to correctly determine if a node is Non-Trusted already when verifying the IRs. Additionally, every single node will always consider himself a trusted node.

After the consensus execution, the node's local Remote Attestor reviews the table generated by the consensus mechanism to determine whether any Non-Trusted nodes have been detected. If any non-trusted nodes are identified, the local Remote Attestor will take appropriate action, such as ***unsubscribing*** from their corresponding WAM channels. This helps to prevent the node from receiving further potentially harmful messages from those nodes, thus ensuring the continued security and integrity of the DTCB group.

The consensus rule varies based on the number of DTCB nodes (N) and it can be obtained by applying the following formula:

$$rule(N) \geq \begin{cases} N/2, & \text{if } N \text{ is even} \\ (N \textbf{ div } 2) + 1, & \text{if } N \text{ is odd} \end{cases} \tag{5.1}$$

The ***div*** operator represents an integer division. The rule outcome is used in taking the overall trust decision related to every single node, which will construct the global Trust-Status table.

To better understand this process, let's suppose that the DTCB is formed by ***four nodes*** and that they all have published on the Tangle their local Trust-Status tables. Additionally, let's suppose that "Node3" has been compromised and lies about the trust state of the other evaluated nodes. From the point of view of "Node1", after having retrieved all the trust tables of the other peers, launches the execution of the consensus mechanism, which outputs the final trust decision table as shown in figure 5.6.

From figure 5.6, it can be highlighted how each honest node is able to correctly detect untrusted nodes even before the consensus outcome is produced. However, the consensus logic takes its trust decisions driven by a ***majority*** approach. In this example, Node3 is considered to be Non-Trusted by all the other nodes, and thus, it will be detected as compromised by all the honest nodes and also by the consensus logic. Even if Node3 tries to lie about the trust status of the other DTCB nodes, the consensus mechanism allows the correct identification of honest nodes also in such cases. The node does not evaluate its own trust, which is why in each local trust table there is no trust decision about itself, but it is assumed that each node considers itself as Trusted. By utilizing the formula 5.1 we obtain the consensus rule to apply. In the case shown in figure 5.6, the $rule(4) \geq 2$ is obtained and for every row, the number of Non-Trusted decisions is summed. If this sum satisfies the consensus rule, it means that the node related to the scanned row is Non-Trusted, otherwise, it will be considered Trusted.

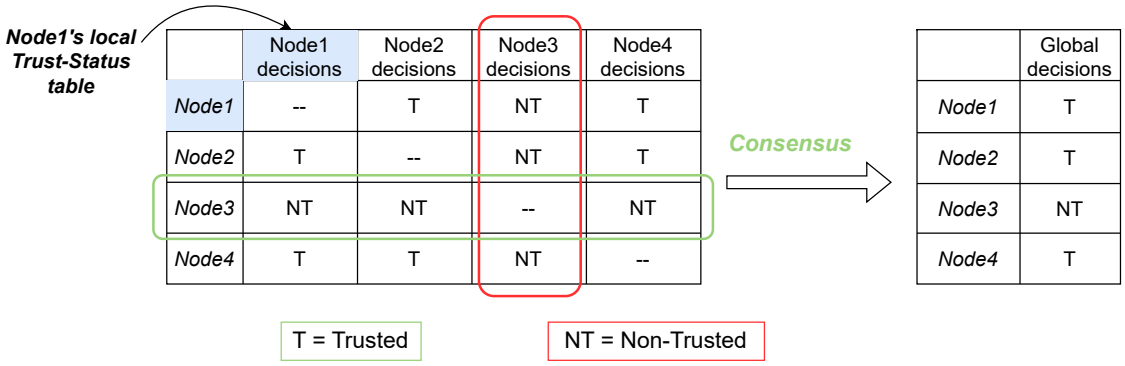

Figure 5.6. Example of a consensus outcome from Node1's perspective.

Taking as reference the above example, it is important to note that if more than one node has been compromised the majority trust decision cannot be taken properly. Hence, supposing that two out of the four nodes have correctly been expelled from the DTCB group and that one of the two remaining nodes is compromised, only one trust decision per node will be available. This situation would lead the consensus outcome to be of little value. Hence, given that N is the number of DTCB participants and that A is the number of compromised DTCB nodes, to autonomously maintain a distributed state of trust, the following conditions must be met:

$$N \geq 3, \qquad A(N) < N \ \mathbf{div}\ 2 \tag{5.2}$$

As a result, in the case that $N = 4$ or $N = 5$, the maximum tolerable number of compromised nodes is equal to one; instead, when $N = 6$ only two compromised nodes are tolerable.

However, the proposed consensus protocol is only at a preliminary stage and covers only a limited set of possible use cases. In fact, this is only a setup for further investigation and improvements to this dowel of the DTCB.

Chapter 6

Tests and Results

In this chapter, we will provide a detailed description of the tests that were performed, as well as an overview of the infrastructure that was deployed to measure the results. This information will give readers a clear understanding of the methodology used to conduct the tests and the tools used to obtain accurate and reliable measurements.

The goal is to measure the time of multiple ***Attestation cycles***. It is important to note that the Attestation cycle refers to the duration between the publication of a new nonce by the heartbeat and the verification of all the IRs published by other DTCB peers by a local Remote Attestor, followed by the execution of the consensus protocol.

By evaluating the time performance of Attestation cycles, we can determine the amount of time an attacker has to compromise a device before it is verified by other DTCB nodes. This information is crucial in understanding the system's security and integrity, as a long time between cycles provides more opportunities for an attacker to go undetected.

6.1 Testbed

The PoC was tested on an IOTA private Tangle based on the Chrysalis protocol, with an infrastructure illustrated in figure A.1. The Tangle configuration with which the tests have been executed is the following:

- The Coordinator has been instructed to generate a milestone every ***30 seconds***.
- To enhance the message validation rate of a private Tangle, which experiences low message traffic, a Spammer has been configured to publish messages at a Message Per Second (MPS) rate equal to ***10***. Consequently, the Spammer generates 10 new messages every second.
- All the three Hornet nodes have been set up to perform the PoW with a score (see Sect. 4.2.4 for detailed information about the PoW) equal to ***1000***.

- By evenly distributing the traffic load across all three Hornet nodes, the NGINX load balancer facilitates PoW computation on each node. If only one Hornet node were to receive all the requests, it would be the sole node capable of providing computational power.

To test the performance of the application in terms of Attestation cycles, three RaspberryPi devices (version 4) were deployed. These devices must be synchronized in order to output valuable timing results. In order to ensure that the clocks of the devices involved in the testing were synchronized with an error lower than one millisecond, a Network Time Protocol (NTP) server was set up. This allowed for accurate timekeeping across all devices and ensured consistent and reliable measurements during the testing process. The NTP server ensures a precise time because it uses Global Navigation Satellite System (GNSS) as a time source to provide highly accurate time information to the devices on the network. The low clock synchronization error is also achieved because both the NTP server and the RaspberryPi devices were located within the same Local Area Network (LAN). Figure 6.1 illustrates the utilized testbed.

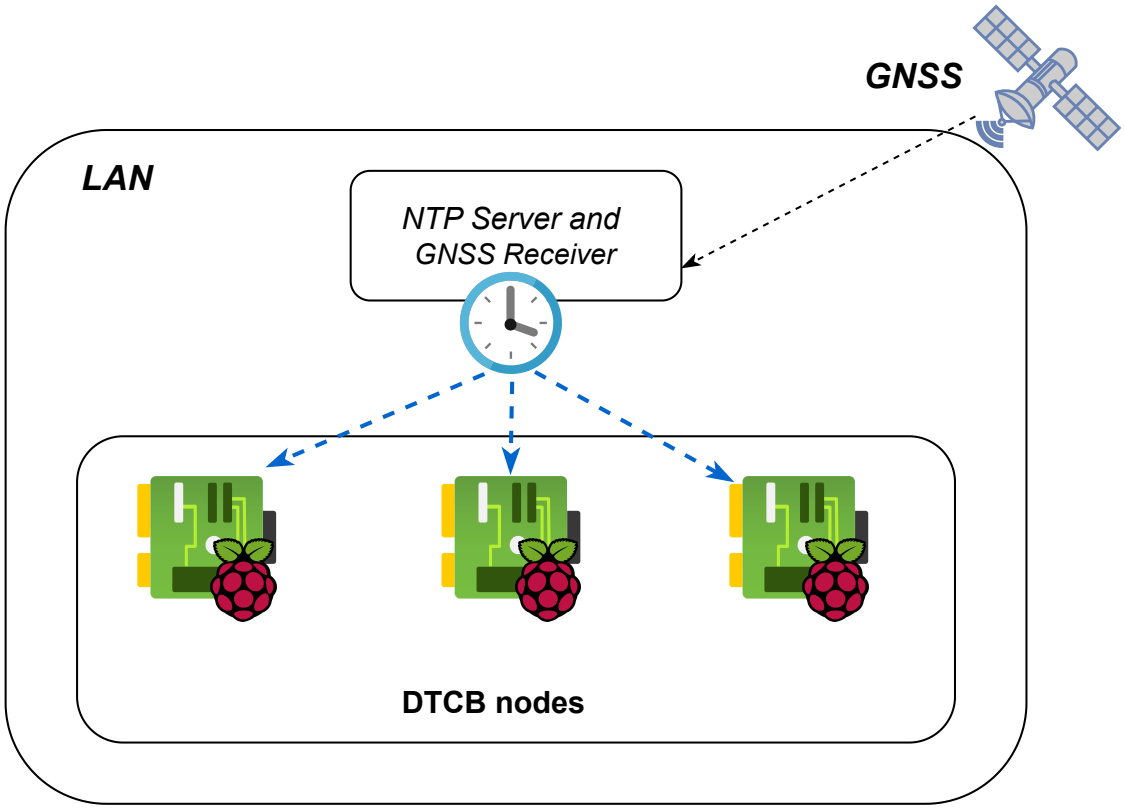

Figure 6.1. Installed testbed overview.

6.2 Performance tests

During performance tests, the time required for each DTCB node to finish an Attestation cycle is considered. This time interval represents the duration for which a node is exposed to attacks before being validated by the other members of the DTCB. An Attestation cycle starts as soon as the heartbeat publishes a new nonce. As a result, the length of a cycle is calculated by subtracting the timestamp when the local Remote Attestor completed the verification and consensus process from the timestamp when the corresponding nonce was published. In this case, three DTCB nodes have been deployed with one of them acting also as the heartbeat. Hence, the obtained timings have been calculated by applying the following formula:

$$cycle\_device_i = device_i Timestamp_{end} - heartbeat Timestamp_{nonce}$$

The NTP server error margin is approximately 0.150 milliseconds, which is negligible compared to the time required to complete a single cycle.

Figure 6.2 displays the trend of the time needed to execute a complete Attestation cycle based on 100 samples. In this scenario, the configuration employed is identical to that which is outlined in the above section.

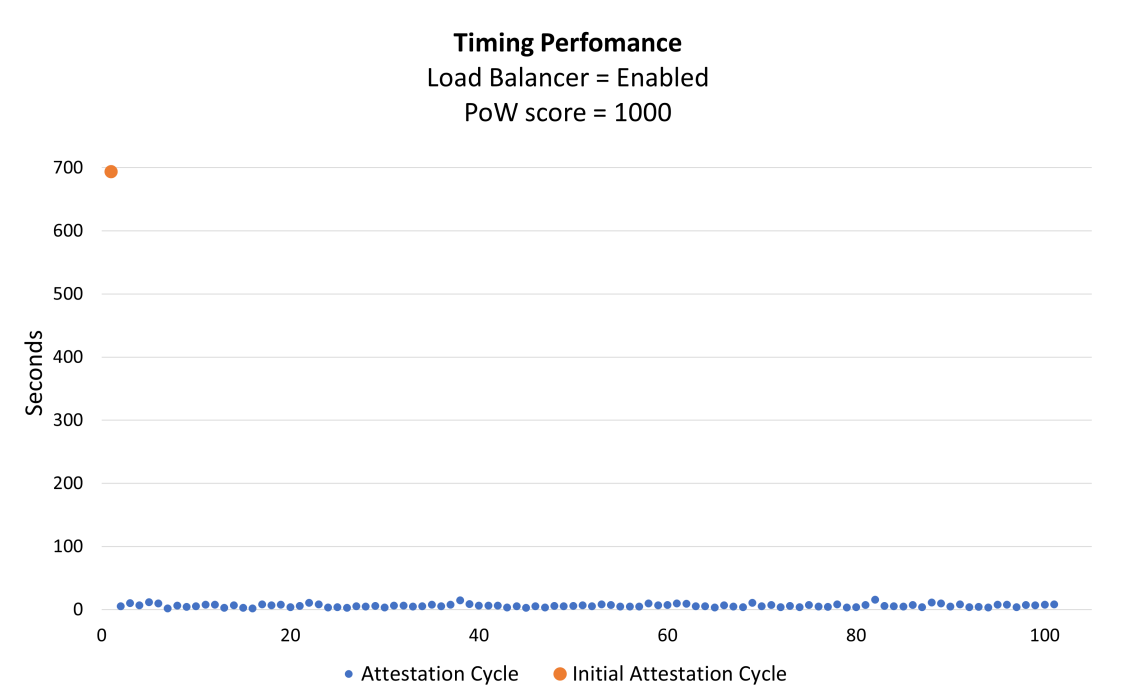

Figure 6.2. Attestation cycles trend.

The above figure provides limited insights, except for the initial Attestation cycle. Notably, there is a significant difference between the first timing value, which is around 700 seconds, and all subsequent ones. This occurs because, in the first cycle, the local Remote Attestor published an Integrity Report (IR) containing

a complete IMA ML model. In deployed RaspberryPi devices, the size of the ML model is typically around 150-160 kB shortly after booting, leading to a larger IR in the first cycle. Subsequent executions only transmit the changes made to the previously sent IMA ML, allowing for a faster writing process.

To facilitate a more comprehensive interpretation of the results, the first Attestation cycle has been excluded. Figure 6.3 depicts the time distribution of the samples collected.

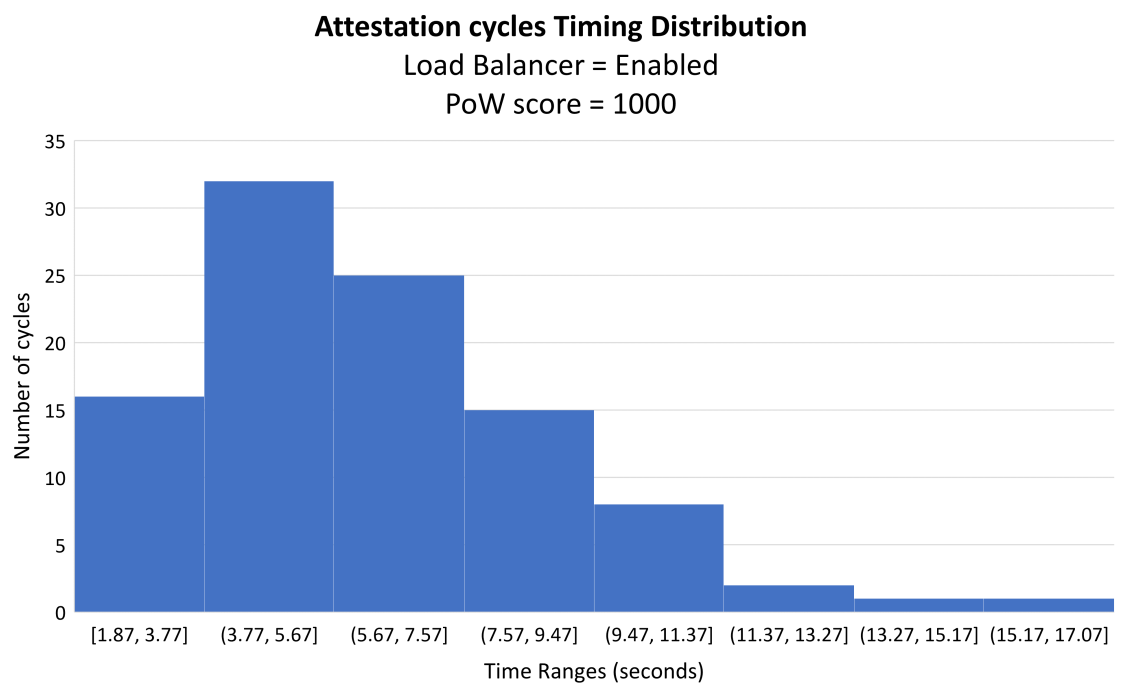

Figure 6.3. Attestation cycles timing distribution.

From the time distribution histogram, it can be noted that almost every Attestation cycle gets concluded within at most 7,570 seconds. Around 15% of the cycles terminate in the first time range and the most probable execution time of a cycle is around 30%. The slightly elevated time ranges are attributed to the possibility that during the execution, the IMA ML may expand, causing a few Integrity Reports (IRs) to be larger than usual. Only about 10% of the samples lay in the last three time ranges. This may depend on the Tangle performance or may depend on the network traffic load that can slow down the delivery of the requests. Finally, on average an Attestation cycle is terminated in around 6,240 seconds.

To conclude, except for the first Attestation cycle, the average time that a node is exposed to attacks is relatively low. The high Tangle's PoW score limits the speed of the writing and reading process. The PoW is necessary to shape the traffic load of the Tangle which allows avoiding attackers, with a higher computational power with respect to the other peers, to take the control of the whole network.

Chapter 7

Conclusion and Future Work

In conclusion, this work presents the concept of DTCB that enables a decentralized approach where a group of nodes autonomously maintain a distributed state of trust among them without the need for any interaction with centralized entities. To achieve such a goal, a custom RA protocol has been proposed to work in a decentralized context enforced by the usage of a DLT, specifically the IOTA Tangle. To maintain a trust status distributed among the DTCB nodes a group-consensus protocol has been also proposed. In the future, this opens up new scenarios employing more sophisticated consensus protocols to support various scenarios that have not been considered in this project. Also, a dynamic joining mechanism could be developed to allow external nodes to become part of the DTCB at any time. Hence, these considerations present promising research topics for further investigation in the future. To conclude, this work serves as an introduction to the concept of DTCB and its various facets. However, it is important to note that this is only the beginning, as there are many other potential capabilities and use cases of DTCB that have not been explored in this study. Further research can explore the various untouched topics and advance our understanding of DTCB's full potential.

Appendix A

User Manual

A.1 Private Tangle Infrastructure Setup

This section illustrates the steps needed to set up a function Chrysalis-based private tangle which has been for testing the PoC implementation throughout the entire project progress. The private tangle has been deployed inside the LINKS cybersecurity laboratory. Figure A.1 shows a high-level view of the infrastructure. Three Hornet nodes together with a Coordinator and a Spammer run in Docker containers to sustain the lifecycle of the Tangle. One Hornet node is deployed in a different machine to allow better isolation of core nodes like the Coordinator. Hence, clients will make requests by contacting "Server-1" which is equipped with a public IP address, while the rest of the nodes are deployed within "Server-2" which lies in the private network of the laboratory. Additionally, an NGINX load balancer is used to evenly distribute the traffic to all the Hornet nodes.

Installation steps

On the ***Server-2***, clone the one-click-tangle project from Github and access the correct folder dedicated to private tangle deployments.

```
$ git clone https://github.com/iotaledger/one-click-tangle.git
$ cd one-click-tangle/hornet-private-net/
```

As the next step, the `docker-compose.yml` file needs to be modified as follows:

- Disable the Autopeering node by deleting its docker-compose section as well as in the script `private-tangle.sh`.
- For every service in the file, substitute the image *gohornet/hornet:1.2.1* with *iotaledger/hornet:1.2.2*.
- Modify the *"ports"* of the Hornet node docker-compose section as in figure A.2. This allows exposing the ports on the host machine.

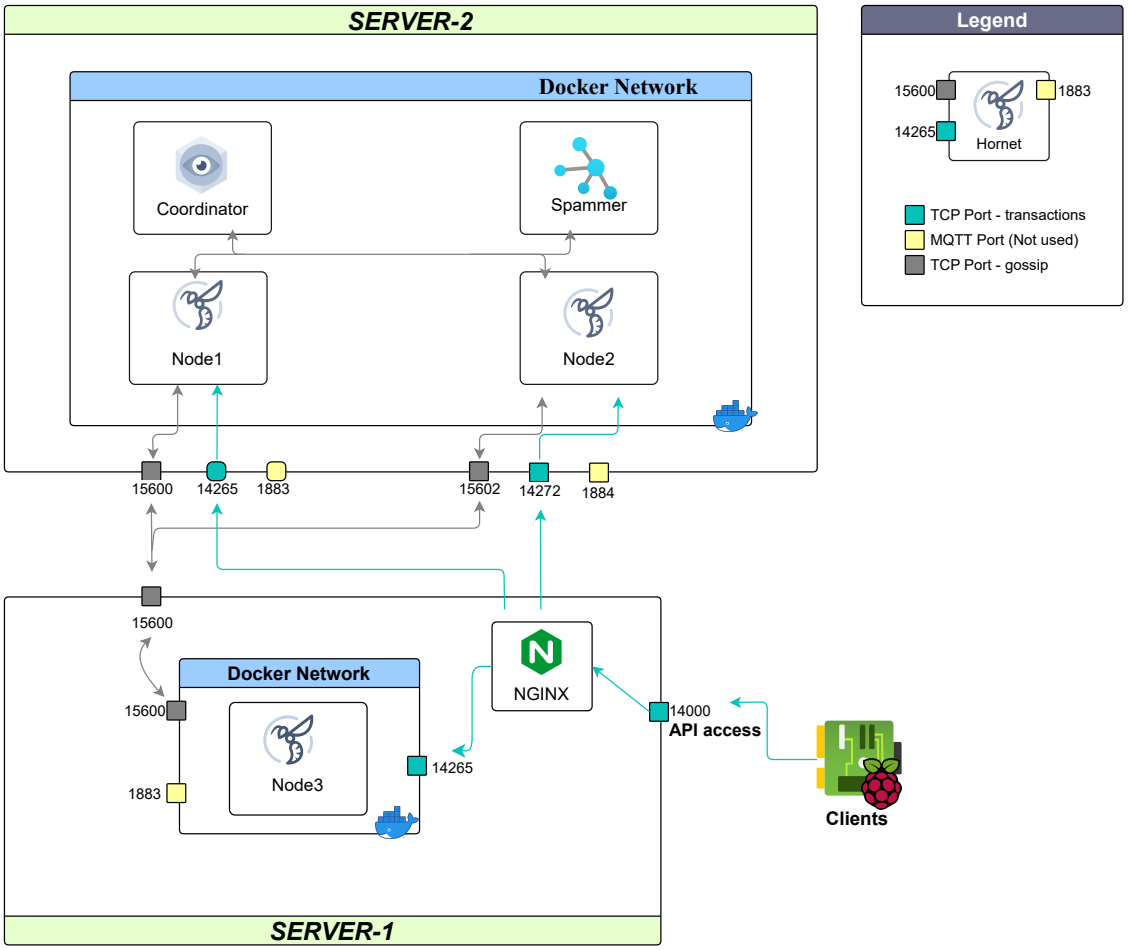

Figure A.1. Overview of the deployed private infrastructure.

```
    - "Server2-IP:14265:14265"
    - "127.0.0.1:8081:8081"
    - "Server2-IP:1883:1883"
    - "Server2-IP:15600:15600"
```

Figure A.2. Mapping ports on the host machine.

Now everything is ready to launch the provided script which deploys a Coordinator, a Hornet node, and a Spammer.

```
$ chmod +x private-tangle.sh
$ ./private-tangle.sh install
```

Figure A.3 shows what the output should look like, and illustrates that a **snapshot** has been generated as well as the other identities, specifically the Coordinator's one together with its key pair. The Coordinator's public key is saved in the "`coo-milestones-public-key.txt`" file.

```
private-tangle

e3aa8383f0d33dd53f30f7873b3a946e512109bc339b39917e69ec83211987c4
Pulling coo ... done
Pulling node ... done
Pulling spammer ... done
Generating an initial snapshot...

...

Snapshot creation successful!
Initial Ed25519 Address generated. You can find the keys at
   key-pair.txt and the address at address.txt

...

Bootstrapping the Coordinator...
Creating hornet-private-net_coo_run ... done
Waiting for 10 seconds ...
2023-02-16T11:17:22Z INFO Coordinator milestone issued (1):
d1d73d493a16c8d55704ea143026ff558847ace29267cb98f10f88dd5eb2ab92
Coordinator bootstrapped!

...

Creating coo ... done
Creating spammer ... done
Creating node1 ... done
```

Figure A.3. Example of private-tangle output.

The PeerIDs are locally saved in different files which should be kept safe. For example, the "node1" PeerID is in the "`node1.identity.txt`".

```
Your p2p private key (hex):
   1432bfbaca5badbefd065f46b57053d532a9f505470a6b7563bd...
Your p2p public key (hex):
   469458a1e84e691b1487d130ebaed97115fe81c74a57facadb8a...
Your p2p public key (base58):
   5kWjkF8J4ZPuqbegh6sPk9fuS9ZTst2ZwqaMzSoPVkus
Your p2p PeerID:
   12D3KooWEZsuVNi1FTfQFVA7KhcjWwkjvUc6ua73uRwPNRxHPhdo
```

The second Hornet node has to be deployed. The snapshot path, the multiaddr information of the Hornet "node1", and the Coordinator's public key have to be supplied. The peers comminucate through the gossip protocol which for node1 runs on the 15600 TCP port.

```
$ cd extra-nodes/
```

Open the docker-compose.yaml file and substitute the image *gohornet/hornet:1.2.1* with *iotaledger/hornet:1.2.2*. Now launch the new node.

```
$ chmod +x private-hornet.sh
$ ./private-hornet.sh install "node2:14272:15602:8082"
   (../coo-milestones-public-key.txt)
   "/ip4/(Node1-IP)/tcp/15600/p2p/(Node1-PeerID)"
   "../../../snapshots/private-tangle/full_snapshot.bin"
```

Being a different machine the snapshot file, the Coordinator's public key, and a multiaddr of one of the two running Hornet nodes need to be provided also to "Server-1". Open the docker-compose.yaml file in the ./extra-nodes folder, and substitute the image *gohornet/hornet:1.2.1* with *iotaledger/hornet:1.2.2*. The third Hornet node can be now deployed as well.

```
$ git clone https://github.com/iotaledger/one-click-tangle.git
$ cd one-click-tangle/hornet-private-net/
$ mkdir snapshots && cd snapshots
$ mkdir private-tangle && cd ../
$ sudo chown -R 65532:65532 ./snapshots # now copy the
   snapshot in ./snapshots/private-tangle/
$ cd extra-nodes/
$ chmod +x private-hornet.sh
$ ./private-hornet.sh install "node3:14265:15600:8083"
   (coo_public_key_PATH)
   "/ip4/(Server2-IP)/tcp/15600/p2p/(Node1-PeerID)"
   "../../../snapshots/private-tangle/full_snapshot.bin"
```

Server-1 is equipped with a public IP address to allow clients outside the private network to make requests to the Hornet nodes. To this extent, a ***firewall*** (Uncomplicated Firewall), together with an ***NGINX load balancer***, have been set up on this machine.

```
$ sudo apt update
$ sudo apt-get install ufw
$ sudo ufw allow in on enp2s0 from any to any port 14000
    # Allow incoming traffic to port 14000 on public IP
       address interface
$ sudo apt install nginx
```

Create the server block configuration to instruct the even traffic distribution of API requests among the Server-1 and Server-2 nodes. Create the new file in `/etc/nginx/sites-available/dtcb.io` and add the lines displayed in figure A.4:

```
upstream dtcb {
  server Server1-HornetNode-ContainerIP:14265;
  server Server2-PrivateIP:14265;
  server Server2-PrivateIP:14272;
}
server {
  listen 14000;
  server_name Server-1-PublicIP;
  location / {
    proxy_pass http://dtcb;
  }
}
```

Figure A.4. NGINX Load Balancer configuration.

Next, the file needs to be "enabled" by creating a link to the directory which NGINX reads from the startup. Now the NGINX daemon must be restarted to catch up with the changes.

```
$ sudo ln -s /etc/nginx/sites-available/dtcb.io
   /etc/nginx/sites-enabled/
$ sudo nginx -t # Test for syntax errors
$ sudo systemctl restart nginx
```

At this point, a private Tangle should be up and running.

A.2 RaspberryPi Setup

RaspberryPi-4 devices, equipped with an Infineon TPM-slb9670, have been used to act as clients for the Tangle and to develop the PoC. From a Windows device, download the Raspberry Pi Imager tool from https://www.raspberrypi.com/software/ and flash an SD card by installing the 32-bit version of Raspi OS lite. The Raspberry can be launched. The kernel has to be rebuilt to support the IMA module and the TPM-related modules.

```
$ sudo apt update
$ sudo apt install git bc bison flex libssl-dev make
$ git clone --depth=1 https://github.com/raspberrypi/linux
$ cd linux
$ KERNEL=kernel7l
$ make bcm2711_defconfig
$ nano .config
```

Modify the "`.config`" by adding/modifying the lines displayed in figure A.5:

```
CONFIG_INTEGRITY=y
CONFIG_IMA=y
CONFIG_IMA_MEASURE_PCR_IDX=10
CONFIG_IMA_NG_TEMPLATE=y
CONFIG_IMA_DEFAULT_TEMPLATE="ima-ng"
CONFIG_IMA_DEFAULT_HASH_SHA256=y
CONFIG_IMA_DEFAULT_HASH="sha256"
CONFIG_IMA_AUDIT=y
CONFIG_IMA_LSM_RULES=y
CONFIG_TCG_TPM=y
CONFIG_HW_RANDOM_TPM=y
CONFIG_TCG_TIS_CORE=y
CONFIG_TCG_TIS_SPI=y
CONFIG_SPI_BCM2835=y
CONFIG_SPI_BCM2835AUX=y
CONFIG_SPI_BITBANG=y
CONFIG_SPI_GPIO=y
```

Figure A.5. Modules to enable for re-building the kernel.

Save and close the file. Now the kernel can be built.

```
$ make -j8 zImage modules dtbs
$ sudo make modules_install
$ sudo cp arch/arm/boot/dts/*.dtb /boot/
$ sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
$ sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
$ sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
```

Enable the SPI interface and the TPM's device tree overlay.

```
$ sudo nano /boot/config.txt
```

At the bottom of this file add the following lines:

```
dtparam=spi=on
dtoverlay=tpm-slb9670
```

Finally, add to the kernel boot parameters the desired IMA properties.

```
$ sudo nano /boot/cmdline.txt
```

Concatenate the following line to the already existing parameters of the file.

```
lsm=integrity ima_policy=tcb ima_template=ima-ng
   ima_hash=sha256
```

Reboot the device and the kernel should be correctly set up with both IMA and the TPM drivers enabled. Repeat this process for every RaspberryPi that needs to be installed.

! The **rpi-5.15.y kernel** could not properly boot the IMA module and the TPM drivers in the correct order. The IMA module was loaded before the TPM's spi-driver, and thus, IMA was not able to detect the TPM. Me and my colleague Alberto, managed to **patch** the rpi-5.15.y kernel. The accepted Pull-Request is available at https://github.com/raspberrypi/linux/pull/5003.

A.3 TPM-related Software setup

In order to access the TPM functionalities, the TCG offers the TSS library that exposes the needed APIs to develop TPM-based applications. Furthermore, to manage the concurrent connections and TPM resources, the TPM2 Access Broker and Resource Manager needs to be installed.

TSS Library Installation

On the RaspberryPi device, the TSS library can be compiled as follows:

```
$ sudo apt -y update
$ sudo apt -y install autoconf-archive libcmocka0 \
    libcmocka-dev procps build-essential pkg-config \
    libtool automake libssl-dev uthash-dev autoconf \
    libjson-c-dev libini-config-dev libcurl4-openssl-dev \
    libltdl-dev uuid-dev libglib2.0-dev
$ wget https://github.com/tpm2-software/tpm2-tss/releases/
    download/3.2.0/tpm2-tss-3.2.0.tar.gz
$ tar -xvf tpm2-tss-3.2.0.tar.gz && cd tpm2-tss-3.2.0
$ ./configure --prefix=/usr
$ make -j8
$ sudo make install
$ sudo ldconfig
```

TPM2 Access Broker and Resource Manager

For installing and correctly install the *tpm2-abrm* daemon, these steps shown in figure A.6 have to be performed on the device:

```
$ wget https://github.com/tpm2-software/tpm2-abrmd/
   releases/download/2.4.1/tpm2-abrmd-2.4.1.tar.gz
$ tar -xvf tpm2-abrmd-2.4.1.tar.gz && cd tpm2-abrmd-2.4.1/
$ ./configure --with-dbuspolicydir=/etc/dbus-1/system.d
   --with-systemdsystemunitdir=/usr/lib/systemd/system
      --libdir=/usr/lib --prefix=/usr
$ make -j8
$ sudo make install
$ sudo udevadm control --reload-rules && sudo udevadm trigger
$ sudo systemctl daemon-reload
$ systemctl status tpm2-abrmd.service # here it should be dead
$ sudo systemctl start tpm2-abrmd.service # after this it
   should be active
```

Figure A.6. TPM2-abrm installation guide.

A.4 IOTA and WAM libraries installation

For the correct execution of the PoC, the ***iota.c*** and the ***WAM*** libraries need to be properly installed.

For the correct installation of the IOTA C library, the following steps have to be performed:

```
$ cd
$ sudo apt-get install cmake
$ git clone https://github.com/iotaledger/iota.c.git && cd
   iota.c && git checkout dev
$ mkdir build && cd build
$ cmake -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
   -DIOTA_WALLET_ENABLE:BOOL=TRUE -DCryptoUse=libsodium
   -DCMAKE_INSTALL_PREFIX=$PWD ..
$ make all
$ make install
```

The WAM library can be obtained as follows:

```
$ cd
$ git clone https://github.com/Cybersecurity-LINKS/WAM.git
```

It is important that the WAM folder and the iota.c folder lay in the same root directory (i.e. `/home/user/WAM`, and `/home/user/iota.c`).

A.5 Proof of Concept

This section describes the structure of the project as well as how to build and run it. A high-level view of the project exposes the following folders and files:

```
+ PoC/
|-- Consensus/
| # Contains the consensus logic.
|-- HeartBeat-Utils/
| # Contains the source code for launching a Heartbeat node.
   Also, it contains the scripts necessary for the generation
   of the Indexing Files.
|-- IMA/
| # Contains the source code for reading the IMA ML from the
   security fs.
|-- RA/
| # Contains the source code for deploying a local Remote
   Attestor.
|-- TPA/
| # The source code necessary for launching a local TPA.
|-- Whitelist_generator/
| # A python script file for easily generating whitelists.
|-- build_HeartBeatutils.sh
| # Bash script for automatically compiling the source code
   related to the HeartBeat-Utils folder.
|-- build_RA.sh
| # Bash script for automatically compiling the source code
   related to the RA folder.
|-- build_TPA.sh
| # Bash script for automatically compiling the source code
   related to the TPA folder.
```

Configuring the project

For every needed Raspberry, to successfully run the project, prepare the local environment by performing the following steps:

```
$ cd /etc/
$ sudo mkdir tc
$ sudo mkdir tc/TPA_AKs
$ sudo chown $USER tc
```

If all the previous components have been correctly installed, the root user directory (`/home/$USER/`) should be composed of:

```
+ PoC/
+ WAM/
+ iota.c/
...
```

Now compile the various main project components of the PoC:

```
$ cd && cd PoC
$ ./build_HeartBeatutils.sh /home/$USER/
$ ./build_RA.sh /home/$USER/
$ ./build_TPA.sh /home/$USER/
```

Prepare the *Indexing Files* that have to be distributed to all the other nodes. The distribution process is left to the administrator.

```
$ cd && cd PoC/HeartBeat-Utils/generateIndexesWAM/
$ sudo chmod +x WAM_generateIndexes
$ NUM_FILES=4 # the number should be equal to the number of
   nodes that have to run the PoC
$ ./WAM_generateIndexes $NUM_FILES
```

The executable generates in this case the following files:

```
heartbeat_write.json
RA_index_node1.json, TPA_index_node1.json
RA_index_node2.json, TPA_index_node2.json
RA_index_node3.json, TPA_index_node3.json
RA_index_node4.json, TPA_index_node4.json
```

Each node must possess one RA Indexing file and one TPA Indexing file. Only one node will be selected to run also as HeartBeat and will also need the heartbeat Indexing file. These files **should** be saved in the `/etc/tc` folders of the various nodes, except for the heartbeat file which **must** be placed in the following directory: `./PoC/HeartBeat-Utils/heartbeat_WAM/`. Before going online each node must generate the whitelist with which the other nodes will then verify the integrity status. To generate a new whitelist, a python script may be used as in A.7.

```
$ cd && cd PoC/
$ cd Whitelist_generator/
$ DIRS=/home/$USER/.ssh'' # The files you want to include
   in the whitelist
$ python whitelist_generator.py sha256 $DIRS # python
   v.3.8.0 used
```

Figure A.7. Whitelist generation script.

The script outputs a whitelist that should contain something similar to figure A.8.

```
d77d5531019c462af5f8036ab6bbdf98f26bfe6959d9551d19b4c846d9c33f19
   32 /home/$USER/.ssh/id_rsa.pub
b1e83554ebd981ec56e26e7668a5ac663b0e650a89cfe60cb2ec07dafa868285
   37 /home/$USER/.ssh/known_hosts.old
06cb4ca3ea41542a289513d0450b3b34f808e4fc425effa80f90c08e657a65fd
   33 /home/$USER/.ssh/known_hosts
8ae475f630877ba10cb1a80fe0d97898716b102bf11bcdd40d1b1e6e29d82f79
   28 /home/$USER/.ssh/id_rsa
```

Figure A.8. Example of a whitelist.

Running the project

The PoC can be launched on every node. Hence, both the TPA and the RA modules have to be executed, and optionally also the HeartBeat module. Before launching the TPA, the TPM key material, such as the EK and the AK, must be first generated:

```
$ cd && cd PoC/
$ cd TPA/
$ sudo ./PoC_TPA init
```

The TPA can be launched by performing the following commands:

```
$ cd && cd PoC/
$ cd TPA/
$ sudo ./PoC_TPA (path_to_file)/TPA_index_node1.json run
```

For executing the RA module, hence a local Remote Attestor, perform the following commands:

```
$ cd && cd PoC/
$ cd RA/
$ NODES_NUM = 4
$ NUM_VERIFIERS = $NODES_NUM - 1 # The RA must know the
   number of DTCB nodes that will participate minus it self.
$ sudo ./RA (path_to_file)/TPA_index_node1.json run
   $NUM_VERIFIERS
```

Repeat this process on all the deployed RaspberryPis (i.e. the DTCB participants).

At this point, both the TPAs and the RAs are waiting for the heartbeat to publish a nonce in order to further proceed. The selected node that has to run the HeartBeat module must perform the following commands:

```
$ cd && cd PoC/
$ cd /HeartBeat-Utils/heartbeat_WAM/
$ NUM_NODES = 4 # The Heartbeat must know the number of DTCB
   nodes that will participate.
$ sudo ./WAM_heartbeat $NUM_NODES
```

All the modules will run indefinitely to check the DTCB compliance of the deployed devices.

Appendix B

Developer Manual

B.1 Generating The Indexing Files

The source code contained in "`./PoC/HeartBeat-Utils/generateIndexesWAM/`" can be compiled and leveraged for an automated generation of the Indexing Files. Once compiled, the executable `WAM_generateIndexes` generates the following files based on the number of needed files that can be passed as input:

```
heartbeat_write.json
RA_index_node1.json, TPA_index_node1.json
RA_index_node2.json, TPA_index_node2.json
...
RA_index_nodeN.json, TPA_index_nodeN.json
```

These files are stored in the `./PoC/HeartBeat-Utils/generateIndexesWAM/` folder, then the administrator has the task of correctly distributing them to the DTCB nodes.

The function `generate_iota_index` is used for generating an Index together with its corresponding Ed25519 key pair. This function is exposed by the WAM library and is defined as follows:

`uint8_t generate_iota_index(...)`
The generation process conforms to what is explained in 4.3 and illustrated in figure 4.6.

Input:

- `IOTA_Index *idx`, pointer to a IOTA_Index structure which contains the parameters defined in figure B.1. First, a seed is created and used for the private key generation. The key pair is then generated and, by hashing the public part, the index is obtained.

Output: It returns a WAM_ERR_NULL code in case of error or a WAM_OK code for successful function execution.

```
uint8_t index[INDEX_SIZE];
uint8_t berry[SEED_SIZE];
iota_keypair_t keys;
```

Figure B.1. IOTA_Index data type.

The cJSON library is used for formatting raw data in JSON-like encoding. This allows more readable file content. The relevant cJSON-related functions are the following:

`cJSON* cJSON_CreateObject(void)`

Input: None.

Output: In case of failure NULL is returned, otherwise, a reference to the allocated object is returned.

`cJSON* cJSON_CreateString(...)`
It generates a cJSON item from a string.

Input:

- `const char* string`, the string is encapsulated into a cJSON item.

Output: In case of failure NULL is returned, otherwise, the reference to the cJSON item is obtained.

`cJSON_bool cJSON_AddItemToObject(...)`

Input:

- `cJSON* object`, pointer to a cJSON object. It acts as the "destination" object as the name of the function suggests.
- `const char* string`, it represents the key that will be put in front of the source value.
- `cJSON* item`, it is the cJSON item added to the destination object. A string can be transformed into a cJSON item.

Output: In case of failure false is returned, otherwise, true is returned.

`char* cJSON_Print(...)`
Renders a cJSON object to text for transfer/storage.

Input:

- `cJSON* object`, pointer to a cJSON object to be formatted.

Output: The pointer to the formatted text. In case of failure, NULL is returned.

Figure B.2 roughly illustrates a pseudo-code of the performed operation.

```
json = cJSON_CreateObject();

indexes = (IOTA_Index *) malloc(sizeof(IOTA_Index) *
   number_of_indexes);

for(i = 0; i < number_of_indexes; i++) {
    generate_iota_index(&(indexes[i]));
    ...
    item = cJSON_CreateString(indexes[i].index);
    cJSON_AddItemToObject(json, "index", item);
    ...
}

out = cJSON_Print(json);
fprintf(Indexing_File, "%s", out);
```

Figure B.2. Pseudo algorithm for generating an Index File.

B.2 The local Trusted Platform Agent

The local TPA is designed to be executed in two modes depending on the received parameters: the **"init"** parameter allows preparing the environment by generating the EK and AK needed for the Quote operation, and after, it can be run with the **"run"** parameter which launches the execution of continuous quote operations upon receipt of new nonces published on the tangle. The file containing the "root" source code is "`./PoC/TPA/PoC_TPA.c`".

The Init phase

The main function that initializes the TPM is **`initialize_tpm()`** and it is responsible for generating the EK and the AK. The keys are generated and their corresponding NV indexes (pointers to the TPM's NV memory) are saved into a local file called `/etc/tc/keys.conf`. The first index of the file always represents the EK handle, while the second represents the AK handle. To access and modify the file, root privileges are required. If the file does not exist or is empty, the program launches the keys generation process, otherwise, it reads the NV indexes in the file and checks that the corresponding keys are actually in the TPM NV memory. If both of the NV indexes or one of the two indexes do not reference the correct object in the TPM the program launches an error, otherwise, the indexes will be saved in the received inputs. Hence, the `keys.conf` file must be deleted, and at the next program's execution, the keys will be correctly generated. Figure B.3 shows a pseudo-code of the described behavior.

```
bool initialize_tpm(uint16_t *ek_handle, uint16_t *ak_handle){
    tss_r = Tss2_TctiLdr_Initialize(NULL, &tcti_context);
    tss_r = Esys_Initialize(&esys_context, tcti_context, NULL);
    keys_conf = fopen("/etc/tc/keys.conf", "r");
    if(keys_conf == NULL || file_is_empty(keys_conf)){
        goto generate_keys;
    } else {
        ek_handle = fread(..., keys_conf);
        ak_handle = fread(..., keys_conf);
        if(ek_handle == NULL || ak_handle == NULL)
            goto error;
        rc = tpm2_getCap_handles_persistent(esys_context,
            ek_handle);
        rc2 = tpm2_getCap_handles_persistent(esys_context,
            ak_handle);
        if(rc < 0 || rc2 < 0)
            goto error;
    }
}
```

Figure B.3. Pseudo code for checking EK and AK persistence.

`TSS2_RC Tss2_TctiLdr_Initialize(...)` [31].
It initializes a TCTI context used to communicate with the TPM.

`TSS2_RC Esys_Initialize(...)` [31].
Creates an ESYS_CONTEXT instance that will store and manage all the data for communicating with the TPM.

`int tpm2_getCap_handles_persistent(...)`
Reads the number of persistent handles in the TPM and checks that the given input handles exist and are persisted.

Input:

- `ESYS_CONTEXT *esys_contex`, a previously allocated ESYS_CONTEXT.
- `uint16_t *handle1`, the NV index which has to be checked for persistence.

Output:

- `-1` in case the TPM's persistent handles cannot be read.
- `-2` if "handle1" is not an existing persistent handle.
- `0` in case of success.

If it the case, the keys generation depends on two fundamental functions: **`tpm2_createek`** and **`tpm2_createak`**. The key generation phase is illustrated in the pseudo-code shown in B.4.

```
generate_keys:
    keys_conf = fopen("/etc/tc/keys.conf", "w");

    tss_r = tpm2_createek(esys_context, ek_handle);
    ...
    tss_r = tpm2_createak(esys_context, ek_handle, ak_handle);
    ...
    fwrite(ek_handle, ..., keys_conf);
    fwrite(ak_handle, ..., keys_conf);
    fclose(keys_conf);
    ...
```

Figure B.4. Pseudo code for creating the EK and the AK.

`TSS2_RC tpm2_createek(...)`
Creates an EK in the endorsement hierarchy, and persists it in the TPM NV memory. The EK will be created by following the TCG EK profile guidance. Figure B.5 illustrates the key type that will be generated. The public key is locally stored in PEM format in the `/etc/tc/ek.pub.pem` file.

Input:

- `ESYS_CONTEXT *esys_contex`, a previously allocated ESYS_CONTEXT.
- `uint16_t *ek_handle`, the NV index with which the EK can be later referenced.

Output:

- `TSS2_RC_SUCCESS` in case the TPM's EK has been successfully created and persisted.
- `TSS2_ESYS_RC_BAD_VALUE` in case of errors. Internal functions have their error codes to allow a multi-level error log.

`TSS2_RC tpm2_createak(...)`
Creates an AK within the endorsement hierarchy. The generated AK is an RSA 2048-bit key that uses **rsassa** as the signing algorithm and the **SHA256** as a hashing algorithm. It is a signing-restricted key as explained in Sect. 2.2.3. The public key is locally stored in PEM format in the `/etc/tc/ak.pub.pem` file.

Input:

- `ESYS_CONTEXT *esys_contex`, a previously allocated ESYS_CONTEXT.
- `uint16_t *ek_handle`, the NV index with which the EK can be later referenced.
- `uint16_t *ak_handle`, the NV index assigned to the AK. It can be used for later referencing.

Output:

- `TSS2_RC_SUCCESS` in case the TPM's AK has been successfully created and persisted.
- `TSS2_ESYS_RC_BAD_VALUE` in case of errors. Internal functions have their error codes to allow a multi-level error log.

Parameter	Type	Content
type	TPMI_ALG_PUBLIC	TPM_ALG_RSA
nameAlg	TPMI_ALG_HASH	TPM_ALG_SHA256
objectAttributes	TPMA_OBJECT	fixedTPM = 1 stClear = 0 fixedParent = 1 sensitiveDataOrigin = 1 userWithAuth = 0 adminWithPolicy = 1 noDA = 0 encryptedDuplication = 0 restricted = 1 decrypt = 1 sign = 0
authPolicy	TPM2B_DIGEST	
size	UINT16	32
buffer	BYTE	0x83, 0x71, 0x97, 0x67, 0x44, 0x84, 0xB3, 0xF8, 0x1A, 0x90, 0xCC, 0x8D, 0x46, 0xA5, 0xD7, 0x24, 0xFD, 0x52, 0xD7, 0x6E, 0x06, 0x52, 0x0B, 0x64, 0xF2, 0xA1, 0xDA, 0x1B, 0x33, 0x14, 0x69, 0xAA TPM2_PolicySecret(TPM_RH_ENDORSEMENT)
parameters	TPMS_RSA_PARMS	
symmetric->algorithm	TPMI_ALG_SYM_OBJECT	TPM_ALG_AES
symmetric->keyBits	TPMI_AES_KEY_BITS	128
symmetric->mode	TPMI_SYM_MODE	TPM_ALG_CFB
symmetric->details		NULL
scheme->scheme	TPMI_ALG_ASYM_SCHEME	TPM_ALG_NULL
scheme->details		NULL
keyBits	TPMI_RSA_KEY_BITS	2048
exponent	UINT32	0
unique	TPM2B_PUBLIC_KEY_RSA	
size	UINT16	256
buffer	BYTE	All 0

Figure B.5. Endorsement Key profile. (source [6])

The Run phase

First, the `keys.conf` file is read to retrieve the NV indexes that reference the two keys for correctly performing the quote operation. If the keys cannot be found in the TPM the program launches an error and stops, otherwise, it will read the TPA's Indexing file and will initialize all the WAM channels needed for writing and reading to/from the tangle. Then, it starts polling the HeartBeat channel, and as soon as a new nonce is published, the TPA performs the Quote over PCR10 and imports the IMA ML into an internal data structure. Finally, construct the IR and writes it onto the tangle at a specific index. Once this process is over, the TPA starts waiting again for a new nonce. The execution runs indefinitely until the user stops it or an error occurs. A pseudo-code is shown in figure B.6.

```
...
if(!initialize_tpm(...)) {
    printf("Could not retrieve keys handles");
    return ;
}
...
rc = get_Indexes_from_file(...);
for(i = 0; i < num_channels; i++){
    rc = WAM_init_channel(...);
    rc = set_channel_index_read(...);
}
if(!sendAkPub_Whitelist_WAM(...)) {
    printf("Could not write on tangle");
    return ;
}
while(1){
    ret = WAM_read(...) // read NONCE
    if(ret == WAM_OK){
        tss_r = tpm2_quote(...);
        if (tss_r != TSS2_RC_SUCCESS) {
            printf("Error while computing quote!");
            return ;
        }
    }
    sendDataToRA_WAM(...);
}
```

Figure B.6. TPA Run mode pseudo-code.

`int get_Indexes_from_file(...)`
It parses the JSON TPA's Indexing file and stores the read indexes in the corresponding IOTA_Index data structure. In the case of a "write" index, the whole Ed25519 key pair is also stored in the data structure, otherwise, in the case of a "read" index, only the public key will be present in the Indexing file.

Input:

- `FILE *indexing_file`, the file pointer to the Indexing file. The file path is passed as an argument in the command line.
- `IOTA_Index *heartBeat_index`, it contains the Index for reading the nonce on the Tangle.
- `IOTA_Index *writeIR_index`, it references the index used for writing the IR on the Tangle.
- `IOTA_Index *writeAk_Whitelist_index`, it contains the index used for writing the AK and the whitelist on the Tangle.

Output:

- `-1` in case of wrong or unexpected file content.

- `0` in case the file parsing and the file content are correct.

`uint8_t WAM_init_channel(...)`
It initializes the WAM_Channel structure with all the necessary data. The **WAM_Channel** is defined as illustrated in figure B.7.

Input:

- `WAM_channel* channel`, a pointer to a WAM_Channel structure.
- `uint16_t id`, represents the ID of the channel.
- `IOTA_Endpoint* endpoint`, the IOTA_Endpoint structure contains the necessary metadata to be able to connect to a Hornet node.
- `WAM_Key* PSK`, a structure that encapsulated the PSK together with its length. A reference is needed to instruct the WAM_Channel to use that PSK.

Output:

- `WAM_ERR_CH_INIT`, in case of NULL pointers or of wrong provided data, this code error is returned.
- `WAM_OK`, it is returned in successful cases.

```
typedef struct WAM_channel_t {
    uint16_t id;

    IOTA_Endpoint* node;

    IOTA_Index start_index;
    IOTA_Index current_index;
    IOTA_Index next_index;

    uint8_t read_idx[INDEX_SIZE];

    WAM_Key *PSK;

    uint16_t sent_msg;
    uint16_t recv_msg;
    uint16_t sent_bytes;
    uint32_t recv_bytes;

    uint8_t buff_hex_data[IOTA_MAX_MSG_SIZE];
    uint8_t buff_hex_index[INDEX_HEX_SIZE];
} WAM_channel;
```

Figure B.7. WAM_Channel data structure definition.

`uint8_t set_channel_index_read(...)`
This function instructs a WAM_Channel to set the writing starting with the index passed as a parameter.

Input:

- `WAM_channel* channel`, a pointer to a WAM_Channel structure.

- `IOTA_Index* write_index`, it contains the index where the chain starts, together with the Ed25519 keypair.

Output:

- `WAM_ERR`, in case of NULL pointers or of wrong provided data, this code error is returned.
- `WAM_OK`, it is returned in successful cases.

`bool sendAkPub_Whitelist_WAM(...)`
This function writes within one single message on the Tangle the public part of the AK and the previously generated whitelist. The message is "tagged" by including the hash of the AK_{pub} (***NodeID***). The message is built by leveraging the data structures illustrated in figure B.8. The AK_{pub} PEM file is retrieved from `/etc/tc/ak.pub.pem`, while the whitelist can be read from the `./PoC/Whitelist_generator/` folder, and created as explained in A.7

Input:

- `WAM_channel* channel`, a pointer to a WAM_Channel initialized with the correct index.
- `IOTA_Index* write_index`, it contains the index where the chain starts, together with the Ed25519 keypair.
- `AK_BLO *ak_blob`, contains the Ak's public part together with its SHA256 digest.
- `WHITELIST_BLOB *whitelist_blob`, contains the golden values that the other nodes will use in the verification process.

Output:

- `false`, is returned in case of NULL pointers or the writing process failed.
- `true`, it is returned in successful cases.

```
struct whitelist_entry {
    u_int8_t digest[SHA256_DIGEST_LENGTH*2+1];
    u_int16_t path_len;
    char *path;
};
typedef struct {
    u_int16_t number_of_entries;
    struct whitelist_entry *white_entries;
} WHITELIST_BLOB;

typedef struct {
    u_int16_t size;
    u_int8_t *ak_pub;
    u_int8_t ak_hex_digest[SHA256_DIGEST_LENGTH*2+1];
} AK_BLOB;
```

Figure B.8. AK and Whitelist data structures.

`uint8_t WAM_read(...)`
This function reads once from a channel and stores the read bytes in a buffer.

Input:

- `WAM_channel* channel`, a pointer to a WAM_Channel initialized with the correct index where it reads.
- `uint8_t* outData`, the buffer where the read data will be stored.
- `uint32_t *outDataSize`, it is the size of the outData buffer.

Output:

- `WAM_BROKEN_MESSAGE`, is returned in case of corrupted messages.
- `WAM_NOT_FOUND`, it is returned when at the reading index there is no published message yet.
- `WAM_OK`, returned in case a successful read has been performed.

`TSS2_RC tpm2_quote(...)`
This function is responsible for performing the Quote operation only on the PCR10. After the quote has been produced by the TPM, the IMA ML is also read from the security file system. All the generated data, together with the IMA ML, are stored inside a user-defined data structure called **TO_SEND**. The internal composition of this structure is illustrated by figure B.9.

Input:

- `ESYS_CONTEXT *esys_contex`, a previously allocated ESYS_CONTEXT.
- `TO_SEND *TpaData`, a pointer to a previously allocated structure for containing IRs data.
- `ssize_t imaLogBytesSize`, it represents the bytes of the IMA ML that have been up to the time of calling the quote. It is used for avoiding to send the whole log at every cycle.
- `uint16_t *ak_handle`, it is the AK handle needed to load the key and sign the Quote.

Output:

- `TSS2_RC_SUCCESS` in case the quote and all the related operations have been successfully executed.
- `TSS2_ESYS_RC_BAD_VALUE` in case of errors. Internal functions have their error codes to allow a multi-level error log.

```
typedef struct {
    u_int16_t size;
    u_int8_t buffer[MAX_RSA_KEY_BYTES];
} SIG_BLOB; # Quote signature

typedef struct {
    u_int16_t size;
    u_int8_t buffer[sizeof(TPMS_ATTEST)];
} MESSAGE_BLOB; #Quote's structure

struct event_blob {
    struct {
        u_int32_t pcr;
        u_int8_t digest[SHA_DIGEST_LENGTH];
        u_int32_t name_len;
    } header;
    char name[TCG_EVENT_NAME_LEN_MAX + 1];
    u_int32_t template_data_len;
    u_int8_t template_data[512];    /* template related data
*/
};
typedef struct {
    u_int16_t size;
    u_int8_t wholeLog; // 1 = whole log will be sent, 0 =
only part of it
    struct event_blob *logEntry;
} IMA_LOG_BLOB;

typedef struct {
    u_int8_t ak_hex_digest[SHA256_DIGEST_LENGTH*2+1];
    SIG_BLOB sig_blob;
    MESSAGE_BLOB message_blob;
    IMA_LOG_BLOB ima_log_blob;
} TO_SEND;
```

Figure B.9. Data structures composing an Integrity Report.

`int sendDataToRA_WAM(...)`
This function is responsible for sending the IR on the Tangle, writing at the index stored in the WAM_Channel structure passed as input.

Input:

- `TO_SEND TpaData`, The IR data to write on the tangle.
- `ssize_t *imaLogBytesSize`, the function calculates the number of bytes of the IMA ML that has been written.

- `WAM_channel *ch_send`, it is a reference to the WAM channel used for writing on the tangle.

Output:

- `1` in case the IR has been successfully written on the tangle.
- `-1` in case of errors (i.e. NULL pointers or missing necessary data).

`uint8_t WAM_write(...)`
For writing on the Tangle the WAM library exposes this function.

Input:

- `WAM_channel* channel`, a pointer to a WAM_Channel initialized with the correct index where it reads.
- `uint8_t* outData`, the buffer where the read data will be stored.
- `uint32_t *outDataSize`, it is the size of the outData buffer.

Output:

- `WAM_ERR`, it is returned in case of internal errors.
- `WAM_OK`, returned in case a successful write operation has been performed.

B.3 The local Remote Attestor

The local Remote Attestor first reads its Indexing file and prepares itself for reading and writing on the Tangle by setting up all the WAM channels. Afterward, it reads the AKs and the whitelists and stores them in an array of objects. If a node does not publish the AK and the whitelist the execution stops. At this point, the local RA is ready to read the nonce from the tangle. As soon as a node publishes its IR, the verification process starts and the result is stored in the local Trust-Status table. When all the expected IRs have been verified, the consensus protocol is executed. As a consequence, some nodes may get expelled from the DTCB, and the local RA stops reading from their WAM channels. Figure B.10 displays a pseudo-code of the local Remote Attestor operations.

`int read_and_save_AKs_whitelists(...)`
This function reads from the tangle the nodes-related data (i.e. whitelists and the public part of the Aks), except himself, and constructs a table that will be used in the verification process. All the read AK_{pub} are saved in PEM format in the folder `/etc/tc/TPA_AKs`. If a node does not accomplish writing them the execution stops and an error is thrown. The data are stored by following the structure illustrated in figure B.11.

Input:

- `WAM_channel* channel`, a pointer to a WAM_Channel initialized with the correct index where it reads.
- `AK_WHITELIST_TABLE* ak_white_table`, an already initialized object where the AKs and the whitelists will be stored
- `int node_number`, it is the index pointing to a row of the table.
- `FILE *ak_file`, it represents the file where the AK_{pub} will be stored. The root folder is ‘‘`/etc/tc/TPA_AKs/`’’.

```
...
rc = get_Indexes_from_file(...);
for(i = 0; i < num_channels; i++){
    rc = WAM_init_channel(...);
    rc = set_channel_index_read(...);
}

for(i = 0; i < num_channels; i++){
    int res = read_and_save_AKs_whitelists(...);
}

while(1) {
    ret = WAM_read(...) // read NONCE
    if(ret == WAM_OK){
        readIRs = 1;
    }
    while(readIRs > 0) {
        if(IR_channels[i] != ignored && !already_verified){
            ret = WAM_read(...); // read IR of i-th node
            if(ret == WAM_OK) {
                rc = parseIR_to_TPAdata(...);

                rc = verify_PCR10_whitelist(...);
                if(!tpm2_checkquote(...))
                    ver_response[i].is_quote_successful == 0;
                else ver_response[i].is_quote_successful == 1;

                if(ver_response[i].is_quote_successful == 1 &&
                   ver_response[i].number_white_entries == 0)
                    local_trust_table[i] = T;
                else local_trust_table[i] = NT;

                readIRs += 1;
            }
        }
        if(readIRs == expected_IRs_to_verify){
            rc = sendLocalTrustStatus(...);
            rc = readOthersTrustTables_Consensus(...);
            readIRs = 0; // wait new nonce
        }
    }
}
```

Figure B.10. Local Remote Attestor's pseudo-code.

Output:

- `1` in case of successful execution.
- `-1` in case some data are missing from the tangle.
- `-2` in case of errors while processing the read data.

```
struct whitelist_entry {
  u_int8_t digest[SHA256_DIGEST_LENGTH*2+1];
    u_int16_t path_len;
    char *path;
};
typedef struct {
    u_int8_t ak_md[SHA256_DIGEST_LENGTH];
    u_int8_t *path_name;
    u_int16_t number_of_entries;
    struct whitelist_entry *white_entries;
} AK_WHITELIST_TABLE;
```

Figure B.11. AK plus Whitelist data structures used by local Remote Attestors.

`int parseIR_to_TPAdata(...)`
This function reads the IR from the tangle and parses them in the analogous data structure defined in figure B.9.

Input:

- `TO_SEND *TpaData`, it receives an array of the data structure that will be filled by parsing the IR's byte buffer.
- `uint8_t *read_attest_message`, it is the IR's byte buffer to parse.
- `int node_number`, it represents the TpaData's array index where the data will be written.

Output:

- `1` in case of successful execution.
- `-1` in case some data are missing from the tangle.
- `-2` in case of errors while processing the read data.

`bool verify_PCR10_whitelist(...)`
This function re-calculates the PCR10 aggregate value by reading the IMA ML contained in the IR. While reading the IMA ML also the whitelist is verified against the ML. The local Remote Attestor maintains the memory of the last calculated PCR10 of the given node. This allows for correctly constructing the PCR10 aggregate value even if the ML has not been entirely sent. All the verification results are stored in the data structure defined in figure B.12.

Input:

- `unsigned char *pcr10_sha256`, the last calculated PCR10. If the whole IMA ML is sent, it is reset to all zeros.
- `IMA_LOG_BLOB ima_log_blob`, it contains the IMA ML.
- `AK_WHITELIST_TABLE whitelist`, it represents the whitelist of the verifying node.

- `VERIFICATION_RESPONSE *ver_response`, the result of the whitelist verification is stored in this data structure. the whitelist verification is successful if the `ver_response[verifying_node].number_white_entries` is zero.

Output:

- `true` in case of successful execution.
- `false` in case of errors while processing the data.

```
typedef struct {
    uint16_t name_len;
    char *untrusted_path_name;
} UNTRUSTED_PATH;
typedef struct {
    uint8_t ak_digest[SHA256_DIGEST_LENGTH+1];
    uint16_t number_white_entries;
    uint8_t is_quote_successful;
    UNTRUSTED_PATH *untrusted_entries;
} VERIFICATION_RESPONSE;
```

Figure B.12. Data structures for Verification results.

`bool tpm2_checkquote(...)`
This function checks the authenticity of the received quote and also the integrity of the received IMA ML. The public part of the AK is read from "`/etc/tc/TPA_AKs/`". If the quote is successfully verified it also validates the whitelist verification done previously.

Input:

- `TO_SEND TpaData`, it contains the quote's structure together with its signature.
- `uint8_t* nonce`, the nonce read from the tangle will be used to prove the quote's freshness.
- `AK_WHITELIST_TABLE ak_white_table`, it is used to read the AK_{pub}. The file name is contained in the data structure.
- `unsigned char *pcr10_sha256`, it is used to compare its digest with the one contained in the quote's structure (*calcDigest field*).

Output:

- `true` in case of successful execution.
- `false` in case the quote is not genuine.

`bool sendLocalTrustStatus(...)`
This function writes on the tangle the trust decisions that the local Remote Attestor has made about the other nodes' IR. The local trust table is defined as illustrated in figure B.13.

Input:

- `WAM_channel *ch_send`, the channel where to write the local trust table.

- `STATUS_TABLE local_trust_status`, the table to be written on the tangle.

Output:

- `true` in case of successful execution.
- `false` in case the write operation fails.

```
typedef struct {
    uint8_t ak_digest[SHA256_DIGEST_LENGTH+1];
    int8_t status; // 0 = NT, 1 = T
    }STATUS_ENTRY;

    typedef struct {
    uint16_t number_of_entries;
    uint8_t from_ak_digest[SHA256_DIGEST_LENGTH+1]; //
       NodeID's source that created the table
    STATUS_ENTRY *status_entries;
    }STATUS_TABLE;
```

Figure B.13. Trust table definition.

`int readOthersTrustTables_Consensus(...)`
This function reads the local trust tables that the other nodes wrote on the tangle. When all the data are retrieved, a call to the consensus execution is made.

Input:

- `WAM_channel *ch_read`, the channel where to read the local trust tables published by the other DTCB peers.
- `STATUS_TABLE my_local_trust_status`, the local trust table constructed locally. The consensus algorithm needs all the tables.
- `int *invalid_channels_status`, the index of this array represents every node's channel. If a cell is marked it means that the channel related to the i-th cell is ignored. This happens when the owner of that channel (a node) was previously detected as Non-Trusted.

Output:

- `1` in case of successful execution.
- `0` in case the other trust tables could not be read.
- `-1` in case the consensus failed its execution.

B.4 The consensus protocol

This simple consensus mechanism allows for ensuring the coherence of the local trust decision made by every DTCB peer. It is based on a majority vote approach. The consensus mechanism receives all the local trust tables and constructs a global trust-status table. The sum of Non-Trusted decisions made about a single node is

taken, and if it satisfies the consensus rule it is marked as untrusted. Figure B.14 illustrates a pseudo-code of this mechanism.

```
int consensous_proc(...) {

    int *nt_array = calloc(nodes_number, sizeof(int));
    int consensus_rule = get_consensus_rule(nodes_number);

    // prepare the global table
    for(i = 0; i < nodes_number; i++) {
        for(j = 0; j <
            others_local_trust_status[i].number_of_entries; j++) {
            if(!inGlobalTable(others_local_trust_status[i].nodeID))
                setNodeID_inGlobal(global_trust_status,
                    others_local_trust_status);
        }
    }

    // Detect untrusted tables
    for(i = 0; i < nodes_number; i++) {
        for(j = 0; j <
            others_local_trust_status[i].number_of_entries; j++) {
            if(others_local_trust_status[i].status_entries[j].status
                == 0) // status = NT
                nt_array[k] += 1; // sum of NT decisions
        }
    }

    for(i = 0; i < nodes_number; i++) {
        if(nt_array[i] >= consensus_rule)
            global_trust_status->status_entries[i].status = 0; //
                tag the node id as NT
        else global_trust_status->status_entries[i].status = 1;
            // tag the node id as NT
    }

}
```

Figure B.14. Pseudo algorithm of the consensus protocol.

`int consensous_proc(...)`
This function constructs the global trust table as roughly shown in figure B.14.

Input:

- `STATUS_TABLE *others_local_trust_status`, an array that represents all the local trust tables.

- `STATUS_TABLE *global_trust_status`, the global trust-status table that will be constructed.
- `int nodes_number`, it represents the number of rows that the global table should have.

Output:

- `1` in case of successful execution.
- `-1` in case the consensus failed its execution.

`int get_consensous_rule(...)`
This function returns the rule on which the "majority" decision is taken.

Input:

- `int nodes_number`, the consensus rule varies based on the number of DTCB nodes.

Output:

- It outputs the result of the formula 5.1.

Bibliography

[1] The Trusted Computing Group, https://trustedcomputinggroup.org/

[2] Department of Defense, "Trusted computer system evaluation criteria", The 'Orange Book' Series, London (UK), December 26, 1985, pp. 1–129, DOI 10.1007/978-1-349-12020-8_1

[3] T. Hardjono and N. Smith, "Decentralized trusted computing base for blockchain infrastructure security", Frontiers in Blockchain, vol. 2, 2019, DOI 10.3389/fbloc.2019.00024

[4] Trusted Platform Module Library, Part 1: Architecture, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf

[5] Will Arthur, David C. Challener, Kenneth A. Goldman, "A Practical Guide to TPM 2.0", Apress Berkeley, CA, 2015

[6] TCG EK Credential Profile, https://trustedcomputinggroup.org/wp-content/uploads/EK-Credential-Profile-For-TPM-Family-2.0-Level-0-V2.5-R1.0_28March2022.pdf

[7] TPM 2.0 Keys for Device Identity and Attestation, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2p0-Keys-for-Device-Identity-and-Attestation_v1_r12_pub10082021.pdf

[8] C. Huang, C. Hou, H. Dai, Y. Ding, S. Fu, and M. Ji, "Research on linux trusted boot method based on reverse integrity verification", Scientific Programming, vol. 2016, 2016, pp. 1–12, DOI 10.1155/2016/4516596

[9] Trusted Platform Module (TPM) 2.0: A BRIEF INTRODUCTION, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf

[10] TCG TSS 2.0 Overview and Common Structures Specification, https://trustedcomputinggroup.org/wp-content/uploads/TSS_Overview_Common_v1_r10_pub09232021.pdf

[11] TCG TSS 2.0 Enhanced System API (ESAPI) Specification, https://trustedcomputinggroup.org/wp-content/uploads/TSS_ESAPI_v1p0_r08_pub.pdf

[12] A. S. Banks, M. Kisiel, and P. Korsholm, "Remote attestation: A literature review", CoRR, vol. abs/2105.02466, 2021, DOI 10.48550/ARXIV.2105.02466

[13] Integrity Measurement Architecture (IMA), https://sourceforge.net/p/linux-ima/wiki/Home/

[14] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture", 13th USENIX Security Symposium (USENIX Security 04), San Diego (CA, USA), August 9-13, 2004, pp. 1–16

[15] F. Bohling, T. Mueller, M. Eckel, and J. Lindemann, "Subverting linux integrity measurement architecture", ARES '20, New York (NY, USA), August 25, 2020, pp. 1–10, DOI 10.1145/3407023.3407058

[16] Bitcoin: A Peer-to-Peer Electronic Cash System, http://www.bitcoin.org/bitcoin.pdf

[17] M. J. M. Chowdhury, M. S. Ferdous, K. Biswas, N. Chowdhury, A. S. M. Kayes, M. Alazab, and P. Watters, "A comparative analysis of distributed ledger technology platforms", IEEE Access, vol. 7, 2019, pp. 167930–167943, DOI 10.1109/ACCESS.2019.2953729

[18] A. Lopez Vivar, A. Castedo, A. Sandoval Orozco, and G. Villalba, "Smart contracts: A review of security threats alongside an analysis of existing solutions", Entropy, vol. 22, 02 2020, p. 203, DOI 10.3390/e22020203

[19] IOTA, https://www.iota.org/

[20] N. El Ioini and C. Pahl, "A review of distributed ledger technologies", On the Move to Meaningful Internet Systems. OTM 2018 Conferences, Cham (CH), October 18, 2018, pp. 277–288, DOI 10.1007/978-3-030-02671-4_16

[21] IOTA Wiki, https://wiki.iota.org/learn/about-iota

[22] Weighted Uniform Random Tip Selection, https://wiki.iota.org/tips/tips/TIP-0003/

[23] IOTA Proof-of-Work, https://github.com/iotaledger/tips/blob/main/tips/TIP-0012/tip-0012.md

[24] Wrapped Authenticated Message (WAM), https://github.com/Cybersecurity-LINKS/WAM

[25] Masked Authenticated Messaging (MAM), https://blog.iota.org/introducing-masked-authenticated-messaging-e55c1822d50e/

[26] L. coronado garcÃa, "On the security and the eciency of the merkle signature scheme", IACR Cryptology ePrint Archive, vol. 2005, 01 2005, p. 192

[27] STREAMS, https://wiki.iota.org/streams/welcome/

[28] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC-7519, May 2015, DOI 10.17487/RFC7519

[29] A. S. Rawat and M. Deshmukh, "Efficient extended diffie-hellman key exchange protocol", 2019 International Conference on Computing, Power and Communication Technologies (GUCON), New Delhi (India), September 27-28, 2019, pp. 447–451

[30] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC-8259, December 2017, DOI 10.17487/RFC8259

[31] TPM2-TSS Documentation, https://tpm2-tss.readthedocs.io/en/latest/index.html