POLITECNICO DI TORINO

Master's degree course in Computer Engineering

Master's Degree Thesis

Standard-Based Remote Attestation

Supervisors
Prof. Antonio Lioy
Dr. Flavio Ciravegna
Dr. Lorenzo Ferro

Candidate
Davide Romani

Academic Year 2024-2025

Summary

In recent years, the rapid proliferation of Internet of Things (IoT) devices and the expansion of cloud computing have brought significant advantages and innovation across various industries. However, this growth raised serious concerns about security and trust. Devices and services, often operating in untrusted environments, are vulnerable to malicious attacks that can compromise sensitive data and disrupt operations. Remote attestation plays a crucial role in ensuring the security of such environments. This is a process that allows one system to verify the integrity and trustworthiness of another remote system. Central to achieving secure environments is the concept of Trusted Computing, which is based on standards developed by the Trusted Computing Group (TCG). Trusted Computing technologies provide the foundation for securing cloud computing and virtualized systems. A key technology within trusted computing is the Trusted Platform Module (TPM), a dedicated hardware component developed by TCG to enhance device security through secure storage and cryptographic operations. TPMs are fundamental in the remote attestation process, as they allow devices to generate cryptographic evidence of their integrity in a tamper-resistant manner. Recent advances in remote attestation protocols, such as the Veraison Project, made strides toward standardizing and improving the process. However, a gap still exists in fully adhering to the latest protocols. This thesis proposes a novel implementation of the remote attestation procedure in compliance with the RFC-9334 standard, addressing the critical need for secure, scalable, and modular solutions in the field of IoT and cloud computing. This proposal introduces a flexible architecture that can be easily adapted to various use cases while ensuring compliance with up-to-date standards. By filling this gap, the solution presented in this thesis contributes to advancing secure remote attestation, helping to safeguard the growing number of interconnected devices and services in today's digital landscape.

Contents

Chapter 1

Introduction

The increasing adoption of cloud computing and the widespread deployment of *Internet of Things* (IoT) devices have introduced new challenges in ensuring the security and integrity of modern computing environments. With cloud platforms shifting data storage and computational processes away from local infrastructures, concerns about trustworthiness and potential vulnerabilities have grown. Similarly, IoT devices, often deployed in untrusted environments and designed with minimal security considerations, remain susceptible to cyber threats. Ensuring that these systems maintain a verifiable and trusted state has thus become a critical priority in cybersecurity.

To address these challenges, *Trusted Computing* principles, developed by the *Trusted Computing Group* (TCG), provide a robust security foundation. Trusted Computing is based on the concept of a Root of Trust (RoT), which enables systems to verify their integrity through secure cryptographic mechanisms. A central component of this framework is the *Trusted Platform Module* (TPM), a hardware-based security module that offers tamper-resistant capabilities for key storage, cryptographic processing, and system integrity verification. The TPM plays a fundamental role in Remote Attestation (RA). This security mechanism allows a remote verifier to assess the integrity of a system by requesting cryptographic evidence of its state. Remote Attestation is particularly valuable in cloud and IoT security, as it ensures that only trusted devices and software configurations can access sensitive resources.

As security threats continue to evolve, standardization efforts have been made to improve the implementation of Remote Attestation across different environments. The *Remote ATtestation procedureS* (RATS) architecture, formalized by the *Internet Engineering Task Force* (IETF) in RFC-9334, provides a structured approach for establishing attestation frameworks that are scalable, interoperable, and adaptable to various platforms. However, despite these advances, gaps remain in aligning existing attestation solutions with the latest standards. This thesis aims to bridge that gap by implementing a standard-based Remote Attestation framework that adheres to RFC-9334.

The key components of Trusted Computing are investigated, different attestation models are explored and existing attestation frameworks such as *Keylime* and *Veraison* are evaluated. By leveraging these insights, the thesis presents a novel, modular, and scalable approach to Remote Attestation that enhances security in cloud and IoT environments. The proposed solution demonstrates its effectiveness in providing a secure, verifiable, and standard-compliant attestation mechanism through performance and functional testing. Ultimately, this research contributes to advancing the field of cybersecurity by offering a flexible and robust Remote Attestation framework capable of meeting the evolving security demands of interconnected digital ecosystems.

Chapter 2

Trusted Computing

The *Trusted Computing Group* (TCG) is a non-profit organization focused on developing open, vendor-neutral, global industry specifications and standards for trusted computing platforms. These platforms are based on a hardware root of trust to ensure security and interoperability. TCG's core technologies include the *Trusted Platform Module* (TPM), *Trusted Network Communications* (TNC), Network Security and Self-Encryption drives. TCG's work aims to enhance the security of computing systems globally [1].

Trusted Computing (TC), based on a hardware root of trust, was developed to protect computing infrastructure and endpoints. The *Trusted Platform Module* (TPM) provides cryptographic capabilities to enforce system behaviours and defend against attacks such as malware and rootkits. TCG's standards are now widely used in enterprise systems, networks, and mobile devices, securing cloud and virtualized environments. Thousands of vendors offer Trusted Computing-based products to enhance system security. These technologies make systems more reliable, less vulnerable to malware, and easier to deploy and manage [2].

The purpose of this chapter is to present the main concepts behind the *Remote Attestation* (RA) procedure such as the *Root of Trust* (RoT) as well as the TPM.

2.1 Trust

In the context of the TCG specifications, *trust* is conceived to convey an expectation of behaviour. Nevertheless, predictable behaviour is not necessarily related to trustworthy behaviour. Determining the identity of a platform is essential for understanding its expected behaviour. Physically distinct platforms can display identical behaviour if they consist of components (both hardware and software) that function equally. As a result, if the behaviour of their components is identical, their trust properties should also be similar [3]. The TCG outlines frameworks for building trust in a platform by identifying its hardware and software components. The TPM offers techniques for gathering and reporting these identities. When incorporated into a computer system, a TPM reports on both hardware and software in a way that allows the determination of expected behaviour and, from that expectation, the establishment of trust.

2.1.1 Transitive Trust

Transitive Trust is a process in which the RoT decides the trustworthiness of an executable operation. The trustworthiness of this operation is then used to state the trustworthiness of the next one [3].

This process can be achieved by

1. knowing that a function enforces a trust policy before it allows a subsequent function to take control of the TCB, or

2. using measurements of the subsequent functions so that an independent evaluation can establish trust.

The TPM may support either of these methods.

Thanks to the *Transitive Trust*, it is possible to ensure the integrity and security of all the subsequent operations that take place from a valid starting point and the possibility to identify untrustworthy behaviour and the consequent loss of confidence.

2.1.2 Chain of Trust

In a *Chain of Trust*, each component must be measured and verified by the previous one before it can be executed. The process relies on trust in the final element of the chain, which serves as a verifier after having been previously validated itself. Overall trust is established transitively, so the *Transitive Trust* process is involved.

Starting from the first element, which cannot be verified and must be trusted by default, (serving as the RoT) it initiates the chain of trust by ensuring that the boot process begins only after confirming the absence of malicious code. Through a trusted boot process, the RoT ensures that any software running on the device is legitimate and has not been compromised, thus maintaining the device's security. In this way, the RoT indirectly guarantees the trustworthiness of all subsequent components executed after it, establishing a secure foundation upon which trust is passed ahead.

2.2 Trusted Computing Base (TCB)

A *Trusted Computing Base* (TCB) refers to the set of system resources (both hardware and software) responsible for enforcing the system's security policy. One of the main characteristics of a TCB is its ability to protect itself from being compromised by any hardware or software which is not part of the TCB. The TPM is not considered the TCB of a system. Instead, it serves as a component that enables an external entity to verify whether the TCB has been compromised. In certain scenarios, the TPM can prevent the system from booting if the TCB is not properly initialized [3].

2.3 Root of Trust

As stated in the TCG specifications, a *Root of Trust* (RoT) is a component that performs different security-specific tasks such as measurement, storage, reporting, verification and updates. The RoT is a system component that must be trusted because any misbehaviour cannot be detected [3]. TCG-defined methods depend on the Roots of Trust. The set of roots specified by the TCG offers the minimum functionality needed to define characteristics that affect a platform's trustworthiness. Although it is not possible to directly verify if a RoT is working properly, it is possible to understand how the roots are implemented. Certificates offer assurances that the root has been implemented in a manner that ensures its trustworthiness.

This certification boosts confidence in the Roots of Trust implemented within the TPM. Additionally, a certificate from the platform manufacturer can assure that the TPM was correctly integrated on a machine that complies with TCG specifications, ensuring that the platform's Root of Trust can be trusted.

The TCG needs three Roots of Trust to cooperate in a trusted platform:

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

2.3.1 Root of Trust for Measurement (RTM)

The RTM conveys integrity-related information (e.g. measurements) to the RTS. Typically, the RTM is the CPU, which is controlled by the *Core Root of Trust for Measurement* (CRTM). The CRTM is the first set of instructions executed when a new chain of trust is established. Upon system reset, the CPU starts with the execution of the CRTM, which then sends values representing its identity to the RTS. This marks the starting point for a *Chain of Trust*.

Static and Dynamic Root of Trust for Measurement (S-RTM and D-RTM)

The *Core Root of Trust for Measurement* (CRTM) is the starting point for measurements, initiating the first measurements of the platform which are then Extended into the *Platform Configuration Registers* (PCR) within the TPM. The Extend operation is basically the update of a PCR made through the appending of a new hash to the existing PCR value and then through the hashing of this intermediate object. More details on the PCRs and the Extend operation are given in the subsection 2.5.1, for now, it is important to notice that after each Extend operation, the PCR value is unique for the specific order and combination of digest values that were Extended. For these measurements to be meaningful, the executing code must control the environment it operates in, ensuring that the values recorded in the TPM accurately reflect the platform's initial trust state [3].

TCG defines two types of RTM:

1. **Static RTM (S-RTM)**: a power-on reset establishes an environment where the platform is in a known initial state, with the main CPU executing code from a predefined starting location. Since at this point the code has exclusive control over the platform, it can measure the platform's state from the first operations executed after power on (usually BIOS or firmware). These initial measurements allow for the creation of the chain of trust measuring the next piece of code before executing it. This chain of trust is constructed until the OS is fully loaded. Since it is established only once during the reset and cannot be altered, it is referred to as a static Root of Trust for Measurement (S-RTM).

2. **Dynamic RTM (D-RTM)**: an alternative method of initializing the platform available on some processor architectures. In this case, the CPU acts as the CRTM and applies protections to the memory portions that are measured after the system has been booted. So a new chain of trust starts without rebooting the platform, as the RTM can be re-stated dynamically this method is called Dynamic RTM (D-RTM).

It is worth noticing that there is a natural connection between these two types of RTM. In particular, the D-RTM depends on the S-RTM as the former could not exist without the base given by the latter. The S-RTM's chain of trust is the starting point thanks to which the D-RTM can be initiated as it is anchored to the former. So, jeopardizing trust in the static chain of trust would affect the binding to the dynamic one.

2.3.2 Root of Trust for Storage (RTS)

The TPM memory is protected from access by any entity other than the TPM itself. Since the TPM is trusted to prevent unauthorized access to its memory, it can function as an RTS. Some information in TPM memory is not considered sensitive, and the TPM does not restrict its disclosure. For instance, non-sensitive data may include the current contents of a platform configuration register (PCR) holding a digest. However, other information is sensitive, and the TPM ensures it remains inaccessible without proper authorization. (e.g. the private portion of an asymmetric key stored in a Shielded Location.) Sometimes the TPM may use the contents of one Shielded Location to control access to another Shelded Location as well. For instance, access to a private key for signing purposes may be conditioned on the PCR holding specific values.

2.3.3 Root of Trust for Reporting (RTR)

The RTR reports on the contents of the RTS. An RTR report is typically conveyed as a digitally signed digest of the contents of the selected values inside a TPM. Typically, the reported values provided by the RTR are

- evidence of a platform configuration in PCR,
- audit logs,
- key properties.

The interaction between the RTR and RTS is crucial. The design and implementation of this interaction must minimize the risk of tampering that could interfere with the RTR's ability to provide accurate reports. An instantiation of the RTS and RTR should be resistant to all types of software attacks, as well as to the physical attacks implied by the TPM's Protection Profile, while delivering a precise digest of all sequences of the presented integrity metrics.

RTR Identity

The TPM holds cryptographically verifiable identities for the RTR in the form of asymmetric aliases (Endorsement Keys or EKs), which are derived from a common seed. Each seed value and its associated aliases should be statistically unique to a TPM, so the likelihood of two TPMs sharing the same EK should be insignificant. The seed can be used to generate multiple asymmetric keys, all of which represent the same TPM and RTR.

RTR binding to a Platform

The TPM reports the platform's state by quoting the PCR values. To ensure that these PCR values properly reflect the platform's state, it is essential to establish the binding between the RTR and the platform. A Platform Certificate can serve as proof of this binding. The Platform Certificate is the assurance from the certifying authority of the physical binding between the platform (the RTM) and the RTR.

Platform Identity and Privacy Concerns

The uniqueness and cryptographic verifiability of an EK raise concerns about whether directly using that identity could lead to the aggregation of activity logs. Analyzing these aggregated logs might expose personal information that a platform user would not want to share with aggregators. In order to prevent unwanted aggregations, the TCG recommends the usage of domain-specific signing keys and restrictions when using the EK. The Privacy Administrator oversees the use of the EK, including the process of binding another key to it. In fact, the Privacy Administrator's control of the EK differs from the Owner's control of the RTS providing separation of the security and identity uses of the TPM. Unless the EK is certified by a trusted entity, its trust and privacy properties are no different from any other asymmetric key that can be generated by pure software methods. Therefore, the public portion of the EK by itself, is not privacy-sensitive.

2.4 Trusted Platform Module (TPM 2.0)

A *Trusted Platform Module* (TPM) is a system component with a state independent from the host system it reports to. The sole interaction between the TPM and the host system occurs through the interface defined in the TCG specification [3].

TPMs are implemented on physical resources, either directly or indirectly. A TPM can be built using physical resources that are permanently and exclusively allocated to it, or using resources

that are temporarily assigned. The physical resources of a TPM may be either contained within a single physical boundary or spread across multiple physical boundaries.

Some TPMs are implemented as single-chip components attached to systems (typically a PC) via a low-performance interface. These TPMs include a processor, RAM, ROM, and Flash memory. Interaction with these TPMs occurs only through the LPC bus and the host system cannot directly modify values in the TPM's memory, except through the I/O buffer that is part of the interface.

Another sensible implementation of a TPM involves running the code on the host processor while it is in a special execution mode. With this approach, the hardware partitions parts of the system memory so that the memory used by a TPM is not accessible by the host processor unless it is in this specific mode. Moreover, when the host processor switches modes, it always starts the execution at designated entry points. This type of TPM shares many characteristics with stand-alone components, with the host being able to alter the TPM's internal state only through well-defined interfaces. Various schemes can be used to achieve mode switching, such as System Management Mode, Trust Zone^{TM} and processor virtualization.

The main objective of the given information is to define the interaction between the host and the TPM. The prescribed commands enable the TPM to perform specific actions on data stored inside it. A key purpose of these commands is to enable the determination of a platform's trust state. The TPM's ability to fulfil its role relies on the correct implementation of the RoT.

The TPM specification has undergone two development phases. Initially, it evolved from version 1.1b to 1.2, gradually incorporating new capabilities as they were identified by the specification committee. This incremental approach led to a final specification that became quite complex. In the second generation, TPM 2.0, the architecture was completely redesigned from scratch, prompted by the cryptographic weaknesses of SHA-1, resulting in a more integrated and unified design [4]. The TPM considered for the implementation of the proposed work thesis is TPM 2.0.

2.4.1 Architecture

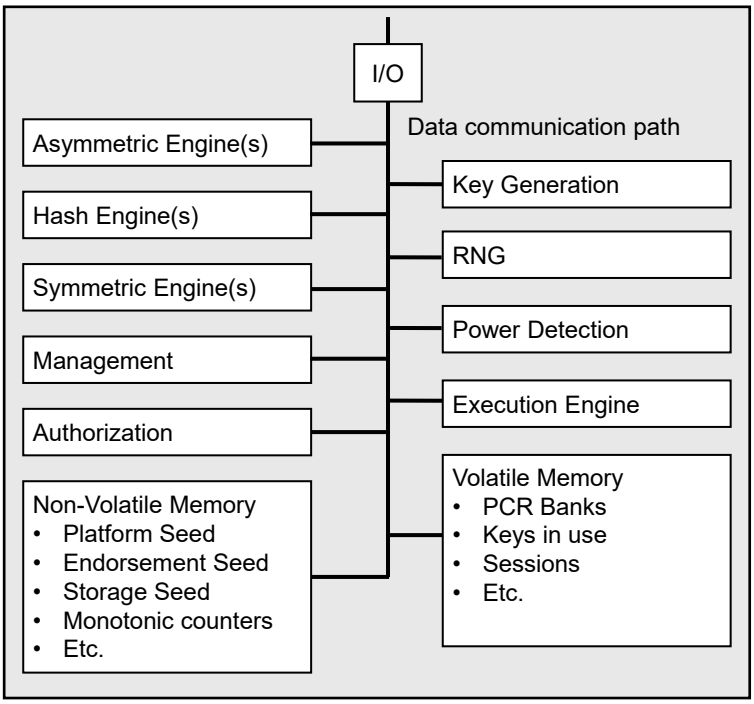


Figure 2.1: Architectural Overview (Source: [3])

TCG defines the TPM standard specifications by dividing its design into distinct components, each outlined to clarify its specific role and functionality within the overall architecture. Documentation of TPM implementation details is a crucial resource and the starting point for designers and manufacturers. Figure 2.1 provides an example of the entire architecture, offering a high-level overview of the individual components and their interconnection [3].

The main components of the TPM architecture are listed and briefly explained below.

I/O Buffer

The I/O buffer functions as the communications area between the TPM and the host system. The system places command data into the buffer and retrieves response data from it.

There is no requirement for the I/O buffer to be physically isolated from other parts of the system, it can be implemented as shared memory as well. However, when the processing of a command begins, the implementation must ensure that the TPM uses the correct values. For instance, if the TPM performs a hash of the command data as part of the authorization process, it must protect the validated data from modification. In particular, the data must be safeguarded from changes before validation and must be in a Shielded Location before modification.

Cryptography Subsystem

The Cryptography subsystem is in charge of implementing the TPM's cryptographic operations. This can be invoked by the Command Parsing module, the Authorization Subsystem or the Command Execution module. The subsystem embodies the **Asymmetric Engine(s)**, the **Hash Engine(s)**, the **Symmetric Engine(s)** and the **Key Generation** module.

The TPM implements conventional cryptographic functions in conventional ways. These operations include

- hash functions;
- asymmetric encryption and decryption;
- asymmetric signing and signature verification;
- symmetric encryption and decryption;
- symmetric signing (HMAC and SMAC) and signature verification;
- key generation.

Random Number Generator (RNG) Module

The RNG is the source of randomness in the TPM. It uses random values for nonces in key generation, and for randomness in signatures.

The RNG is a *Protected Capability* with no authentication, it is composed by

- an entropy source and collector;
- a state register;
- a mixing function (usually, an approved hash function).

The entropy collector gathers entropy from various sources and eliminates any bias. This collected entropy is then used to update the state register, which feeds into the mixing function to generate random numbers.

The mixing function can be implemented using a pseudo-random number generator (PRNG), which produces numbers that appear random from a non-random input (e.g. a counter). By

combining an approved PRNG with an input that has significantly more entropy than a counter, the resulting RNG can have properties that are at least as good as the underlying PRNG, and potentially much better.

The TPM should generate enough randomness for each usage thanks to an internal function. When accessed externally, it must be capable of providing 32 octets of randomness. Larger requests may fail if there is not enough available randomness. Each RNG access produces a new value, regardless of the data's intended use, with no distinction between internal and external accesses.

Authorization Subsystem

The Authorization Subsystem is invoked by the Command Dispatch module at both the start and end of command execution. Before the command is executed, the Authorization Subsystem verifies that the necessary authorization for accessing each Shielded Location has been provided.

There are different types of commands, some of them access Shielded Locations that require no authorizations. Other commands access to some locations that may require a single-factor authorization. Others access some other Shielded Locations that may need the usage of an authorization policy of arbitrary complexity.

The Authorization Subsystem only requires hash and HMAC cryptographic functions. An asymmetric algorithm may be needed if a Signed Policy is implemented.

Random Access Memory (Volatile Memory)

Random Access Memory (RAM) stores TPM temporary data. The TCG specifications states that the TPM RAM data are not mandatory to be lost when TPM power is turned off. Even if data loss is implementation-dependant the TCG specifications will refer to it as volatile and so they are considered in this work thesis.

Following the TCG specifications, when a value has both volatile and non-volatile copies it may be kept in a single location as long as it has the properties of allowing random access and having unlimited endurance.

Not all values in the TPM RAM are in Shielded Locations, a portion of it contains the I/O buffer described previously in 2.4.1. TPM RAM supports three types of storage operations:

- Platform Configuration Registers (PCR), these registers store hashes of measurements taken by external software, and the TPM can later report those measurements by signing them with a specified key; while the details of how the registers function are explained later 2.5.1, it's important to understand that they have a one-way property that prevents spoofing: if the registers provide a representation of a trusted software that operates as expected, then all the register values can be trusted; one useful application of these registers is using them as an authentication signal: you can create a key or other object in a TPM that can only be used if a PCR (or multiple PCRs) is (are) in a specific state [4];

- Object Store, used to store keys and data loaded from external memory, these objects cannot be used or modified unless they are first loaded into TPM RAM using specific commands (e.g. $TPM2_{Load}()$, $TPM2_{CreatePrimary}()$, $TPM2_{LoadExternal}()$, $TPM2_{ContextLoad}()$); the structure used for keys can also be applied to data objects, as long as the access properties are suitable, a special type of data object (called Sealed Data Object) can require specific PCR values or authorization for access, and it is managed like a key; this object must be loaded before use and its context can be saved; TPM also processes other transient structures that are passed in commands but are not stored in the TPM after the command completes; items loaded into the TPM are assigned handles, which allow them to be referenced in future commands;

- Session Store, used to manage and control sequences of operations, sessions can be useful for auditing actions, authorizing actions, or encrypting parameters in commands; a session is created through specific commands and is assigned a handle; the TPM may store session data in a shared memory pool with the object store, or it may have separate dedicated stores for sessions and objects.

TPM RAM must be sufficiently large to manage the transient state, sessions, and objects necessary to execute any implemented command. The minimum RAM requirements for the worst-case scenario in the TCG specifications are:

- two loaded entities (e.g. two keys, a key and a Sealed Data Object, or a hash/HMAC sequence and a key);
- three authorization sessions;
- an input buffer capable of holding the largest command, or an output buffer sufficient for the largest possible response (both depend on the implementation's supported algorithms);
- any vendor-defined state required for operation;
- all defined PCRs.

Non-Volatile (NV) Memory

The NV memory module in the TPM stores persistent state information and includes locations that can only be accessed with Protected Capabilities. Some of this memory is allocated for use by the platform and entities authorized by the TPM Owner. Depending on the vendor's preference, if the specification does not specify where a parameter is stored, it may reside in either RAM or NV memory.

To prevent excessive wear, the TPM checks if data written to NV memory is identical to the current data and avoids writing if so. The platform or OS can define an NV Index for storing persistent data. NV memory can also store loaded objects persistently, and when referenced in a TPM command, the object may be moved into an object slot for more efficient access.

The TRM (TPM Resource Manager) must ensure sufficient RAM for object memory to allow such transparent movements. Additionally, if a command references a persistent object using Transient Object resources, the TRM must ensure that a Transient Object slot is available.

Power Detection Module

This module manages TPM power states in coordination with platform power states. Platform-specific TCG (Trusted Computing Group) specifications must include a requirement for notifying the TPM of any power state changes. The TPM supports only the ON and OFF power states. Any system power transition that requires a reset of the RTM (Resource Manager) will also trigger a reset of the TPM (TPM_{Init}). Similarly, any system power transition causing the TPM to reset will also reset the RTM.

2.5 Entities

A TPM 2.0 entity refers to an item in the TPM that can be directly accessed using a handle. The term is broader than objects, as the objects in TPM terminology refer to a specific subset of entities. This section outlines the different types of TPM entities: *Permanent Entities* (such as hierarchies, the dictionary attack lockout mechanism, and PCRs); *Non-Volatile Entities* (like NVRAM indexes, similar to permanent entities); *Objects* (including keys and data); and *Volatile Entities* (which refer to various session types) [4].

2.5.1 Permanent Entities

A *Permanent Entity* refers to an object whose handle is specified by the TPM specification and cannot be created or deleted.

In TPM 1.2, the only permanent entities were PCRs and the owner. While the storage root key (SRK) had a fixed handle, it was not considered a permanent entity.

TPM 2.0 introduces additional permanent entities, including three persistent hierarchies, the ephemeral hierarchy, dictionary attack lockout reset, PCRs, reserved handles, plaintext password authorization session, and platform hierarchy NV enable. Details are presented below.

Persistent Hierarchies

TPM 2.0 has three persistent hierarchies: platform, storage, and endorsement, each tagged by a permanent handle TPM_RH_PLATFORM, TPM_RH_OWNER, and TPM_RH_ENDORSEMENT. Access to these hierarchies is controlled through authorizations, which consist of an authorization value and a policy, both of which can be modified by the hierarchy's administrator. Although these hierarchies can be disabled, they cannot be deleted. They may also have associated chains of keys and data, which can be wiped out by clearing the hierarchy. Other permanent entities, such as the ephemeral hierarchy and dictionary attack lockout reset mechanism, are also noted as permanent entities that cannot be created or deleted. The following section 2.6 will provide further details on these hierarchies.

Ephemeral Hierarchy

TPM 2.0 includes an ephemeral hierarchy called the `NULL` hierarchy, identified by the permanent handle TPM_RH_NULL. This hierarchy is used when the TPM functions as a cryptographic coprocessor, with both its authorization value and policy set to `NULL`. Similar to the persistent hierarchies, the NULL hierarchy is permanent and cannot be deleted. However, unlike persistent hierarchies, it is automatically cleared after each power cycle of the TPM.

Dictionary Attack Lockout Reset

The dictionary attack lockout mechanism in TPM 2.0 is identified by the handle TPM_RH_LOCKOUT. Like the persistent hierarchies, it has both an authorization and a policy that can be modified by the administrator. Unlike the hierarchies, it does not have a key or object hierarchy. This mechanism is used to reset the dictionary attack lockout or clear the TPM_RH_OWNER hierarchy, and it generally represents the IT administrator of the TPM storage hierarchy.

Platform Configuration Registers (PCRs)

PCRs (Platform Configuration Registers) are secure locations used to record and validate the log of measurements that track a platform's security state. The TPM (Trusted Platform Module) maintains a log of events impacting the platform's security, particularly during the boot process when establishing the Trusted Computing Base (TCB). When new log entries are made, the TPM receives and hashes them into a PCR, allowing for attestation of the log's contents [3].

TPM 2.0 has two possible ways to modify the content of the PCRs:

1. **Reset**, initializes a PCR to a specific value (SET), or clears its contents by setting all the bits to zero (CLEAR); this is usually performed in the early phases of a boot process to ensure that the previous measurements do not affect the current measurement cycle;
2. **Extend**, defined as

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

it is a cumulative hash representing the sequence of all the measurements that have been registered, thus ensuring that any alteration in the sequence will produce a different final hash value.

The TPM has a number of PCRs, which are accessed using their index. They also have an authentication value and a policy, chosen by the specification (generally `NULL`), which may be used to change the value stored in the PCR via a PCR Extend. Reading the value stored in a PCR doesn't require authentication and can be done with a simple $TPM2\_PCR\_READ()$. The TCG PC Client platform specifies a minimum of 24 PCRs [5].

PCR Index	PCR Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8-15	Defined for use by the Static OS
16	Debug
23	Application Support

Table 2.1: PCR usage compliant to the TCG PC Client Platform specifications (Source: [5])

As outlined in Table 2.1, each PCR is usually linked to a specific set of elements for measurement. While it's possible to use a single PCR for all log entries, this approach is not ideal for tracking intermediate system states. A clear example of where this is particularly useful is during the boot process. This association ensures that different aspects of the platform's configuration and state are independently verified. Each PCR extension is linked to an entry in the TCG event log, which allows a challenger to review how the final PCR digests were constructed PCRs can also control access to objects by ensuring the required values are present.

TPMs may support multiple PCR banks (a set of PCRs with the same hash algorithm), each using a specific hash algorithm, but banks can differ in the number of PCRs they contain. Each bank may be used for different applications requiring different security or compatibility needs. Only one bank is mandatory, and programmable to either `SHA-1` or `SHA-256` at boot time [4].

PCRs can be stored in non-volatile memory, though this may affect TPM performance during boot. A TPM must support PCR banks for each algorithm they support and may include special "Resume PCRs" that retain their state across certain TPM sequences, but reset on others.

Reserved Handles

Vendor-specific reserved handles may be included in a TPM if a platform-specific specification requires them. These handles are intended for use by a vendor in the event of a catastrophic firmware security failure, enabling the TPM to testify to the hash of the stored software. As of the current date, no platform specifications have defined any reserved handles.

Password Authorization Session

There is one session that is permanent as well, called a password authorization session at handle TPM_RS_PW (`0x40000009`). A caller uses this handle for plaintext password (as opposed to HMAC) authorization.

Platform NV Enable

The TPM_RH_PLATFORM_NV handle (`0x4000000D`) controls the platform hierarchy NV enable. When disabled, it denies access to any NV index in the platform hierarchy. NV indexes can belong to either the platform or storage hierarchy, with the storage hierarchy controlling NV indexes within it. The platform NV enable separate controls of platform hierarchy NV indexes, allowing independent control of the platform hierarchy (e.g., keys) and platform NV indexes. This separation of controls provides more flexibility. Next some entities that are similar to permanent entities will be discussed: non-volatile indexes, which are non-volatile but not architecturally defined.

2.5.2 Non-Volatile Indexes

An NVRAM index in a TPM is a nonvolatile storage entity that a user can configure with a specific index and set of attributes. Unlike standard TPM objects, NVRAM indexes have more attributes and can be individually controlled for reading and writing. They can be configured to behave like PCRs, counters, or bit fields, and can also be set as "write once" entities. NVRAM indexes have both an authorization value, which can be changed by the owner, and an authorization policy, which cannot be modified once set at creation time. These indexes are associated with a hierarchy, and when that hierarchy is cleared, the NVRAM indexes linked to it are deleted. While objects share similarities with NVRAM indexes (belonging to a hierarchy with data and authorization), they have fewer attributes.

2.5.3 Objects

A TPM object can be a key or data, with a public part and possibly a private part such as an asymmetric private key, symmetric key, or encrypted data. Objects belong to a hierarchy and have both associated authorization data and an authorization policy, which cannot be changed after creation. Commands that use an object can be administrative or user commands, and the user decides which can be executed with authorization data and which requires a policy. However, some commands can only be executed with a policy, regardless of the object's attributes. Like NVRAM indexes, objects belong to one of the four hierarchies: platform, storage, endorsement, or `NULL`, and are cleared when the associated hierarchy is cleared. Most objects are keys, and using them typically requires a non-persistent entity, the session.

2.5.4 Non-Persistent Entities

A nonpersistent entity does not persist through power cycles. While it can be saved using $TPM2\_ContextSave()$, TPM's cryptographic mechanism prevents the saved context from being loaded after a power cycle, ensuring its volatility. Nonpersistent entities come in several classes, with authorization sessions (including HMAC and policy sessions) being the most common. These sessions enable entity authorization, encryption of command and response parameters, and command auditing. Additionally, hash and HMAC event sequence entities store intermediate results of cryptographic operations, allowing the hashing or HMAC of data to be larger than the TPM command buffer. Unlike nonpersistent entities, persistent entities remain intact through power cycles.

2.5.5 Persistent Entities

A persistent entity is an object that the owner of a hierarchy has designated to remain in the TPM through power cycles. Unlike a permanent entity, which cannot be deleted, the owner of a hierarchy can evict a persistent entity. TPMs have a limited amount of persistent memory, so it's important to use persistent entities sparingly. Typically, primary storage keys (like the SRK), primary restricted signing keys (such as the attestation identity key [AIK]), and sometimes endorsement keys (EK) are the primary entities that remain persistent in a TPM. They are discussed in more details in the following section 2.6.

2.6 TPM Keys and Credentials

The TPM is a security device that excels at storing and using keys securely in hardware. It can generate and import both asymmetric and symmetric keys, acting as a key cache for applications with limited memory. The TPM manages three key hierarchies (*Endorsement Hierarchy*, *Platform Hierarchy* and *Storage Hierarchy*), each with security roles controlling parent-child relationships. Each key has specific security features, such as passwords, authorization policies, restrictions on duplication, and usage limits. Keys can also be certified and used to certify other keys, with additional attributes [4].

2.6.1 TCG Base Terminology

This subsection defines the terms used in the context of this work of thesis, compliant to the TCG specifications [6].

Key

The term "key" typically refers to the public key of an asymmetric key pair, either RSA or ECC. When referring to the private key, the term "private key" will be used. Similarly, "symmetric key" will be used when referring to a symmetric key.

Certificate

The term "certificate" refers to an X.509v3 digital certificate. A certificate contains a public key (referred to as "key" above), information about the "subject" (identity), and a signature that covers the entire structure, signed by an authority. Digital certificates, as mentioned here, are ASN.1 structures encoded using DER rules.

When validating a certificate, an entity typically needs a chain of certificates to establish a trust relationship with a trusted anchor (usually a root certificate). This chain is referred to as a "certificate chain" in discussions. Otherwise, the chain is assumed to exist but is not the focus of the current discussion.

Credential

A "credential" refers to a combination of a private key, available for its intended purpose (e.g., creating a signature), and a certificate that links the corresponding public key to a subject (identity) through a signature. With a credential, an entity can present the certificate and use the private key to prove possession of the corresponding private key, typically for authentication purposes.

A private key without a certificate does not qualify as a credential, nor does a certificate without the corresponding private key.

Identity

An "identity" refers to the combination of the `X.509` certificate fields `Subject` and `subjectAltName` within a certificate issued to a device. Identities are unique within the scope of the Certificate Authority that signs the certificate. Furthermore, in an `IDevID/IAK`, both the `Subject` and `subjectAltName` should be unique independently within the scope of the Certificate Authority. This ensures flexibility in creating a unique `LDevID/LAK` derived from an `IDevID/IAK`. The acronyms' meanings are detailed in the following:

- `IDevID`, **I**nitial (factory installed) **Dev**ice **ID**entity certificate;
- `IAK`, OEM or Manufacturer provided AK Key and certificate. The `IAK` certificate contains the public attestation key and the certificate subject matches the `IDevID` certificate;

- `LDevID`, **L**ocally significant (field-installed) **Dev**ice **ID**entity certificate;
- `LAK` (**L**ocal **A**ttestation **K**ey or Certificate), a field-installed certificate containing a public attestation key, with the certificate subject matching the `LDevID` certificate.

2.6.2 TPM 2.0 Key Terminology

The management and classification of TPM keys adhere to the 802.1AR standard, ensuring secure key handling and access restrictions [7]. The TPM's RTS safeguards keys, preventing external access and ensuring a device's identity is securely bound. TPM's Authorization and Policy control further enhance key security by limiting access.

However, the 802.1AR standard does not cover all TPM use cases, particularly those requiring stricter key constraints. Additional keys and certificates are needed to support TCG-specific operations like attestation and verification.

The TCG provides specific definitions and properties for TPM keys to meet its requirements:

- **Signing Key**, a key that can be used only for signing operations, thus serving as `DevID`, given the fact that it is not *restricted*, it can be used for general device authentication purposes if the key is provided together with a certificate, referred to as *Signing Certificate*;
- **Attestation Key (AK)**, a *non-duplicable restricted* signing key, where
  - *non-duplicable* means that the key is created by the TPM and not usable outside of that specific TPM, and
  - *restricted*, means that the key has restrictions on the Sign or Decrypt operations, limiting those operations to the TPM-generated data;
- **Storage Key**, a *restricted* key that is created for encryption and decryption only, storage keys are usually used to protect other keys and never used in device identification contexts;
- **Endorsement Key (EK)**, related solely to the TPM (i.e. not related to the platform on which it is installed), is the most important key which enables the authentication of all the operations performed by the TPM [8];

  the EK is an asymmetric key pair with a public and private key stored in a Shielded Location on the TPM; the public key is accessible from the TPM and is included in the EK certificate, but the private key must remain secret;

  in TPM 1.2, the EK was defined as an RSA 2048-bit key, but TPM 2.0 allows multiple EKs with various asymmetric algorithms; the EK's properties are defined by its public area structure, called the "template", and TPM 2.0 offers default templates for different algorithms; TCG standards provide flexibility by not requiring specific attributes for the EK, however, it is strongly recommended to define it as a *non-duplicable*, *restricted* decryption key; this is suggested because the EK certificate or public EK can serve as a privacy-sensitive, cryptographic identifier unique to a particular platform, details are given in the subsection 2.6.3; platform specifications may guide whether the EK should be persistent when the TPM ships;

  the EK is accompanied by a unique *Endorsement Key Credential*, a certificate issued for the EK that includes additional information about the security features and origin of the TPM. the EK Credential is typically provided by the TPM or platform manufacturer during the manufacturing process.

2.6.3 Key Hierarchies

A hierarchy is a collection of related entities managed as a group, including permanent objects, primary objects at the root, and other objects like keys. NV indexes are part of a hierarchy but not in a tree. Non-permanent entities can be erased as a group.

The cryptographic root of each hierarchy is a seed, a random number generated by the TPM that remains secure. This seed creates primary objects such as storage root keys, which encrypt other objects in the hierarchy. Each hierarchy has a proof value, derived from the seed, used to verify that data supplied to the TPM was generated by it. The TPM may use an HMAC key for authenticity checks on data.

Hierarchies can be persistent or volatile, with each one tailored to specific use cases, such as for platform manufacturers, users, privacy-sensitive applications, or ephemeral needs [4].

Platform Hierarchy

The platform hierarchy in TPM 2.0 is controlled by the platform manufacturer, specifically through the early boot code. Unlike TPM 1.2, where the platform firmware could not rely on the TPM being enabled, TPM 2.0 ensures the platform hierarchy is enabled at reboot with a zero-length password and a policy that can't be satisfied. The platform firmware generates a strong authorization value and may install its policy. Unlike other hierarchies, the platform authorization is entered by the firmware, eliminating the need to store it securely. The platform firmware also controls when the platform hierarchy is enabled or disabled, with the intent that it should always be enabled for use by the firmware and operating system.

Storage Hierarchy

The storage hierarchy in TPM 2.0 is intended for use by the platform owner, such as the enterprise IT department or end user, and is equivalent to the TPM 1.2 storage hierarchy. It has a persistent owner policy and authorization value that are set and rarely changed. The owner can disable the storage hierarchy without affecting the platform hierarchy, allowing the TPM to still be used by platform software. This hierarchy can also be cleared independently by changing the primary seed and deleting persistent objects. Unlike the endorsement hierarchy, which handles privacy-sensitive operations, the storage hierarchy is designed for non-privacy-sensitive tasks.

Endorsement Hierarchy

The endorsement hierarchy in TPM 2.0 is designed for privacy-sensitive operations and is the preferred choice for users with privacy concerns. TPM and platform vendors certify that primary keys (i.e. keys with no parent, the root keys in the hierarchy) in this hierarchy are tied to an authentic TPM and platform. Unlike TPM 1.2, primary keys in this hierarchy can be both encryption and signing keys, with the latter being privacy-sensitive due to its potential to link keys back to a specific TPM. The endorsement hierarchy's enable flag, policy, and authorization value are independent of other hierarchies and are controlled by a privacy administrator, often the end user. This allows users to disable the endorsement hierarchy while still using the storage hierarchy and permitting platform software to access the TPM.

Privacy Concerns

Privacy, in this context, refers to preventing remote parties from correlating TPM digital signatures to prove they came from the same TPM. Users can use different signing keys for different applications to make such correlation difficult. This privacy concern is most relevant for home users who control their platform, while enterprise IT departments may weaken these privacy features. The primary goal is to ensure signing keys cannot be traced back to a single TPM, and the TPM vendor generates endorsement primary seeds and certificates to attest that keys come from an authentic TPM.

If primary keys are signing keys, correlation is easier because all signatures would converge at the same certificate, allowing an attacker to trace them back to the device. To prevent this, primary keys in the endorsement hierarchy are usually encryption keys. When encryption keys are used, creating certificates for descendant keys involves a more complex process called activating a credential. A privacy CA certifies these keys without revealing any correlation, ensuring privacy is maintained.

2.6.4 Key Enrollment and Relationships

Different enrollment use cases for `DevID` certificates arise depending on the platform's manufacturing processes, application, and privacy needs. To prove that a new key belongs to a specific device, the TPM must first be bound to the OEM device, and an Attestation Key (AK) must be bound to the TPM using the EK. The AK certificate serves as the trust anchor for future certificates, linking new keys to the same TPM. The OEM Certification Authority (CA) signs an `IAK` certificate, which asserts that the `IAK` belongs to a specific device, forming a primary security dependency for creating later DevID certificates. The following subsection outlines specific requirements to create the desired binding, with trust for new certificates relying on prior certificates. The trust is illustrated in the associated dependency figure 2.2 [6].

Key Enrollment

A Platform Certificate is a signed manifest that binds the TPM and other installed components to a device through an OEM signature. Unlike a `DevID` certificate, it does not contain a key. An OEM may provide a platform certificate in addition to or instead of `IDevID/IAK` certificates. Defined in the TCG Platform Attribute Credential Profile [9], the platform certificate can be used during the `LAK` certificate enrollment process to offer OEM evidence that a remote device's identifying information and TPM are securely linked to a specific device instance.

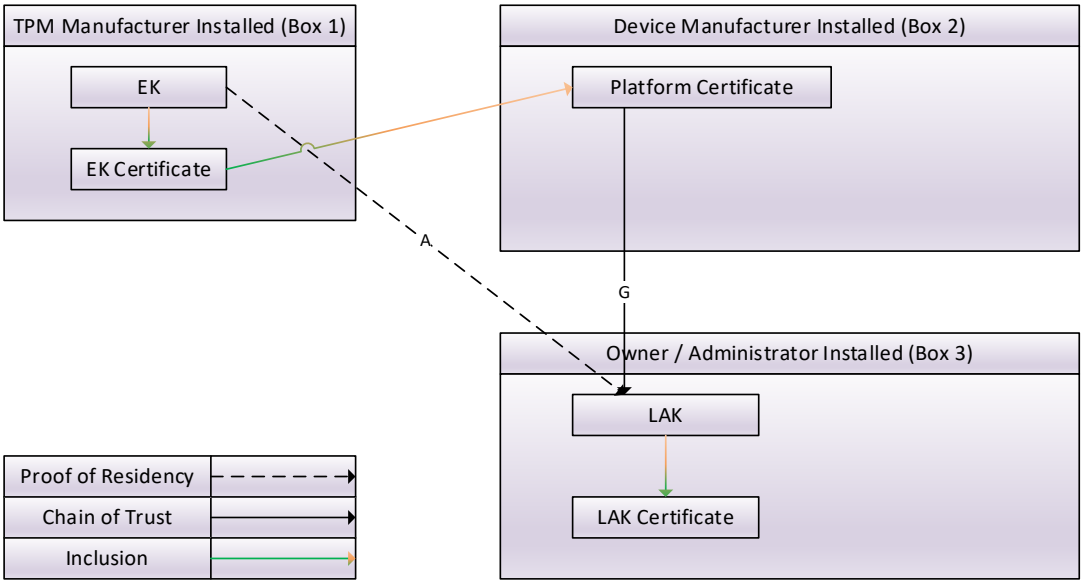


Figure 2.2: Creating LAK Certificate using the Platform Certificate (Source: [6])

The TPM EK Certificate serial number within the Platform Certificate binds the TPM to the device during `LAK` certificate enrollment. The user follows a proof process similar to that of the OEM to ensure the `LAK` is tied to the same TPM as the EK, thus linking it to the device identity (e.g., model and serial number).

- **Box 1**: shows the EK and its certificate, which is signed by the TPM manufacturer to bind the EK to a specific TPM;
- **Box 2**: displays the Platform Certificate, signed by the OEM, which provides information about the device (including a reference to the EK Certificate) and links the TPM to the device;
- **Box 3**: shows the Owner/Administrator-installed `LAK` certificate, where enrollment uses the EK proof and the OEM's assertion that the TPM is installed on the specific device identified in the Platform Certificate;

– line G indicates that information is extracted from the Platform Certificate; this provides device details, including information on the specific TPM installed in the device;

– line A illustrates that the `LAK` created during the enrollment is verified by the Owner's CA to be resident in the same TPM as the EK.

The Platform Certificate may not always contain the device serial number, as it is optional. However, when creating `LDevID/LAK` certificates without an `IDevID/IAK`, the Platform Serial Number must be included to comply with the TCG specifications [6].

Key Relationships

Figure 2.3 illustrates the relationship between certificates installed by the different entities when the device OEM has installed `IDevID` and `IAK` certificates.

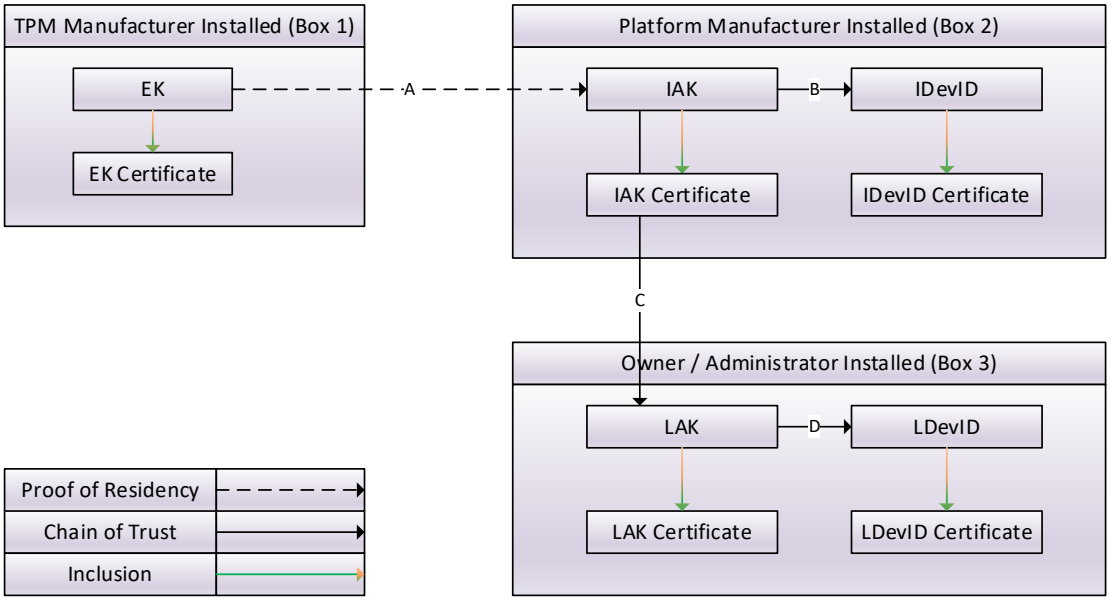


Figure 2.3: Relationships of `IDevID/IAK` Keys and Certificates (Source: [6])

- **Box 1**: shows the EK and EK Certificate stored in the TPM, as installed by the TPM manufacturer, the EK certificate, signed by the manufacturer, binds the EK to a specific TPM from that manufacturer;
- **Box 2**: illustrates the `IDevID` and `IAK`certificates installed by the OEM;
  - line A shows that the `IAK` created during enrollment is verified by the OEM CA to be resident in the same TPM as the EK;
  - line B indicates that the `IDevID` key has been verified to have the correct key properties and to be resident in the same TPM as the `IAK`;
  - (both the `IAK` and `IDevID` certificates are signed by the device OEM's CA);
- **Box 3**: shows the certificates installed by the Owner/Administrator; the OEM CA's assertion that the `IAK` belongs to a specific platform (as identified in the `IAK` Certificate) is used to prove that the new `LAK` key is stored in the same TPM, thereby representing the same platform;
  - as indicated in line C, the `LAK` can be remotely verified by the owner's PKI CA to be in the same TPM as the `IAK`; the `IAK` certificate indicates which device the key resides on;

– line D shows that the `LDevID` key is verified to be in the same TPM as the `LAK`.

These considerations apply only to `X.509` certificates generated using keys created or protected by a TPM for remote device authentication. The use of keys created outside the TPM is not covered. Authentication methods not involving X.509 certificates or a TPM are not addressed.

Chapter 3

Remote Attestation

Remote Attestation (RA) is the process of providing *Evidence* to an appraiser (*Verifier*) over a network, to support a claim about the properties of a target system (*Attester*) [10].

RA process provides distinct phases for the creation of the evidence and its authentication and validation to let the verifier adapt to the changing environment, e.g. rejecting some previously accepted configurations as new vulnerabilities arise.

The development of an *Attestation System* must follow five important guidelines:

1. **Fresh Evidence**, the attestation must be based on temporally fresh information, i.e. computed over the information up to the moment in which the attestation request has been sent;
2. **Comprehensive information**, the attester must provide an entire description of its internal state to the verifier;
3. **Minimal Disclosure**, the attester (or its owner) should manage to limit the disclosure of information about the target (e.g. with an authentication process to put in place policies about the disclosure);
4. **Explicit Semantics**, attestation claims should have explicit semantics to allow decisions to be derived from several claims;
5. **Trustworthiness**, the underlying attestation mechanism must be trustworthy, so the verifier must receive proof of the confidence of the attestation process put in place, as a result, the attester can deliver correct evidence even if it is corrupted.

In the TCG specifications, the term *Attestation* is related to the signature of some TPM data made by the TPM itself. The confidence in the attestation is given by the key used to perform the signature. Maximum confidence is guaranteed by a *fixedTPM* (key cannot be duplicated) restricted signing key created on a TPM with a certificate coming from the TPM manufacturer. Trusted platform properties are supported by a hierarchy of attestations that starts from the TPM, goes through the platform and ends with the set of software/firmware running on the platform.

A *Quote* is a specific type of attestation object which is created by generating a signature over a measurement stored in a PCR (or a set of PCRs), using an AK protected by a TPM. The quote is then sent to a verifier which performs the quote verification by first verifying the signature and then comparing the received PCR values against a set of trusted reference values [3].

RA procedures can be integrated into different scenarios, e.g. in network endpoint assessment, confidential machine learning model protection, confidential data protection, critical infrastructure control, trusted execution environment provisioning, hardware watchdog, FIDO biometric authentication and more [11].

3.1 Logical Workflow

Independently from the specific implementations, a general attestation procedure encompasses the following phases (represented in the figure 3.1):

1. the verifier starts the attestation procedure by sending a request to the attester including a *nonce* (protection from *replay attacks*), and the set of PCRs to be validated;

2. the attester accesses the TPM and reads the selected PCRs' content from the RTS by passing through the RTR supplying the received nonce, then, together with the nonce, it is signed with a certified AK and the final object is returned to the attester;

3. the attester then reads the *Measurement Log* (ML) from the RTM, this contains information about all the operations that have been executed on the target system until the time of the request;

4. the attester sends the attestation response that includes both the signed object (containing the nonce and the set of PCRs) and the ML to the verifier;

5. the verifier performs the following operations on the attestation response:

   - it verifies the signature of the received object to confirm that they come from the system's authentic RoT;
   - it validates the ML's integrity and recalculates all PCR *extend* operations recorded, to verify that the ML extension results match the received PCR values;
   - it further checks the ML in order to confirm that only the authorized operations have been performed by ensuring that all the entries in the log are compliant with the verifier's appraisal policy.

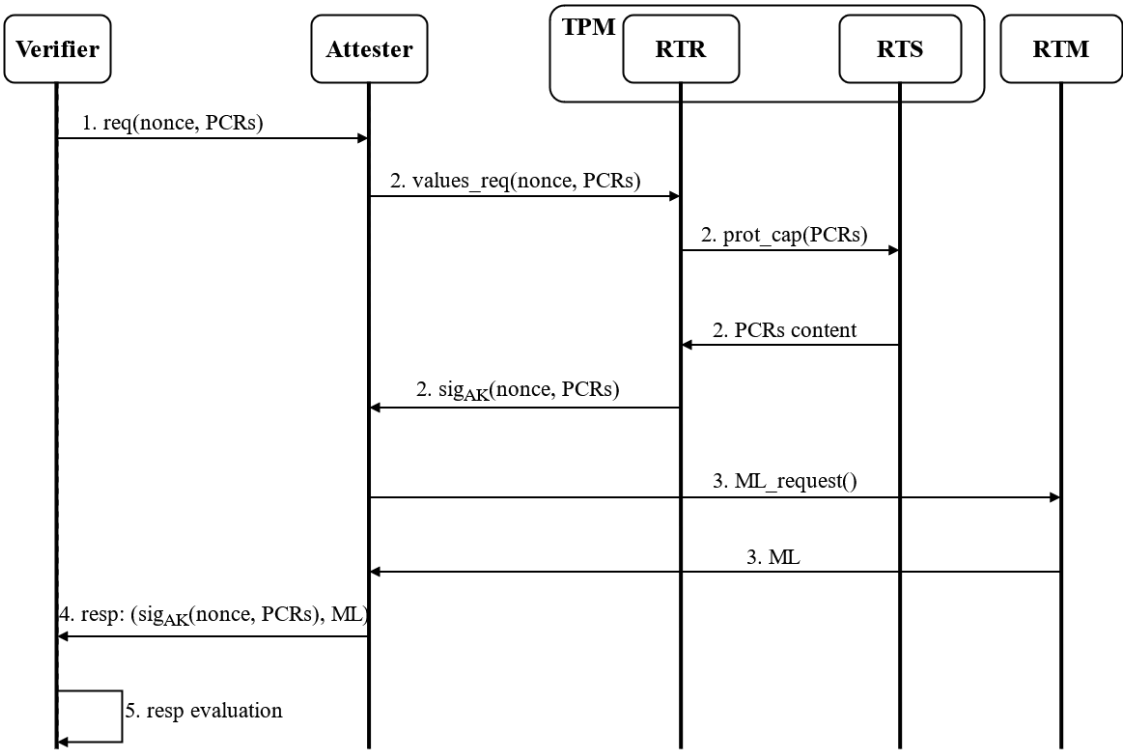


Figure 3.1: Remote Attestation Workflow

3.2 Remote ATtestation procedureS (RATS)

In *Remote ATtestation procedureS (RATS)* there is one system (the "Attester") that provides verifiable evidence about its state to allow another system (the "Relying Party") to assess its trustworthiness. Remote attestation procedures involve a "Verifier" who evaluates the evidence based on predefined policies to help the Relying Party make an informed decision.

The document outlines a flexible architecture with defined roles and interactions, offering a universal set of terms applicable to various attestation methods. It also introduces topological models, such as the "Passport" and "Background-Check" models, to illustrate common data flow patterns. Emphasizing the importance of clear terminology, the document aims to promote semantic interoperability across diverse systems and vendors. It highlights the distinction between trust (a choice made by one system about another) and trustworthiness (an attribute supporting that decision), stressing the need to understand this difference [11].

3.2.1 Architecture

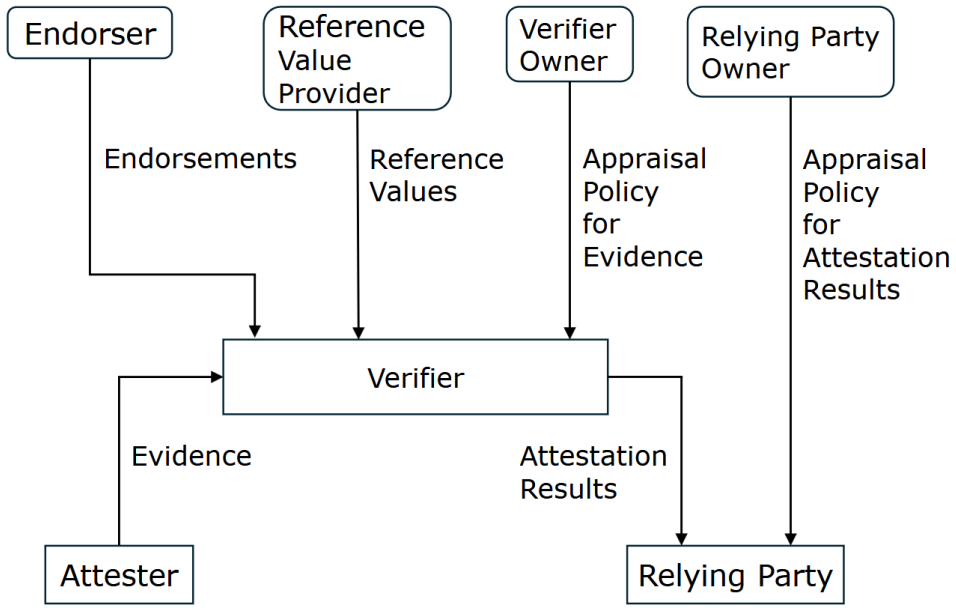


Figure 3.2: Conceptual Data Flow (Source: [11])

Figure 3.2 describes the data exchanged among the different roles, independent of protocol or use case. Roles are assigned to entities, typically system components or devices [12].

The Attester creates an *Evidence* that is sent to the *Verifier*, which uses the *Evidence*, the *Reference Values*, and the *Endorsements* (secure statements that an Endorser vouches for the integrity of an Attester's various capabilities, such as Claims collection and Evidence signing) to assess the Attester's trustworthiness through an *Appraisal Policy for Evidence*. Where this term refers to a set of rules that a Verifier uses to evaluate the validity of information about an Attester. This process is called "appraisal of Evidence". The Verifier then generates the *Attestation Results* for the Relying Party.

The Relying Party applies its own *Appraisal Policy for Attestation Results* (a set of rules that direct how a Relying Party uses the Attestation Results regarding an Attester generated by the Verifiers) to make decisions, such as authorization.

Both the appraisal policies for Evidence and Attestation Results can be obtained or configured through various mechanisms, such as protocols or programming, respectively by the Verifier Owner and the Relying Party Owner [11].

3.2.2 Attester's Environments

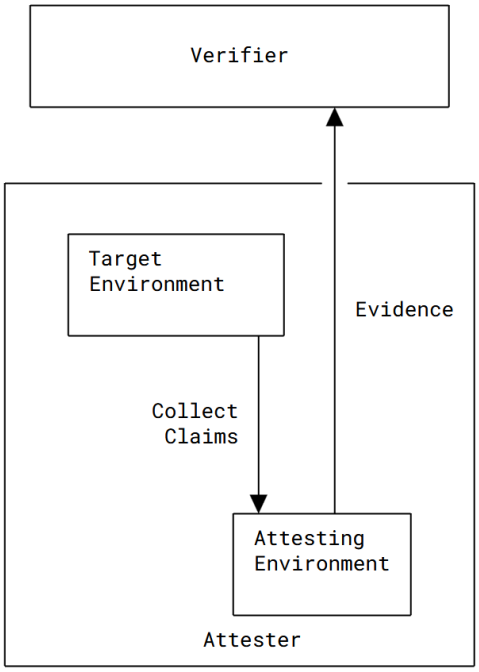


Figure 3.3: Two Types of Environments within an Attester (Source: [11])

As shown in figure 3.3, an Attester consists of at least one Attesting Environment and one Target Environment, which are typically co-located in a single entity. In some implementations, these environments may be combined or have multiple instances.

Attesting Environments collect data from Target Environments by reading system registers, calling subsystems, and measuring various assets like code and memory. This data is then used to create Claims, which are formatted using cryptographic methods, such as signing or encryption, to generate Evidence. Attesting Environments can be implemented using various hardware or software components like TEEs, TPMs [3], or BIOS firmware.

Not all execution environments can collect Claims by default, but those designed for this purpose are called "Attesting Environments". For example, a TPM does not collect Claims independently; it requires another component to supply data, making the combination of the TPM and the data-feeding component the Attesting Environment.

3.2.3 Models

Figure 3.2 illustrates the data flow between an Attester, a Verifier, and a Relying Party. The Attester sends its Evidence to the Verifier, and the Relying Party receives the Attestation Result from the Verifier. This subsection provides details of two reference models for this communication, along with an example composition of them. However, the discussion is meant for illustration and does not limit the interactions between the roles to the models presented.

Passport Model

This model is shown in figure 3.4. The Attester provides Evidence to the Verifier, which checks it against an appraisal policy and returns an Attestation Result that is treated as opaque data, potentially caching it for future use. The Attester can present this result, along with any additional Claims, to one or more Relying Parties, who will then evaluate it against their own appraisal policy.

The process can fail in three ways:

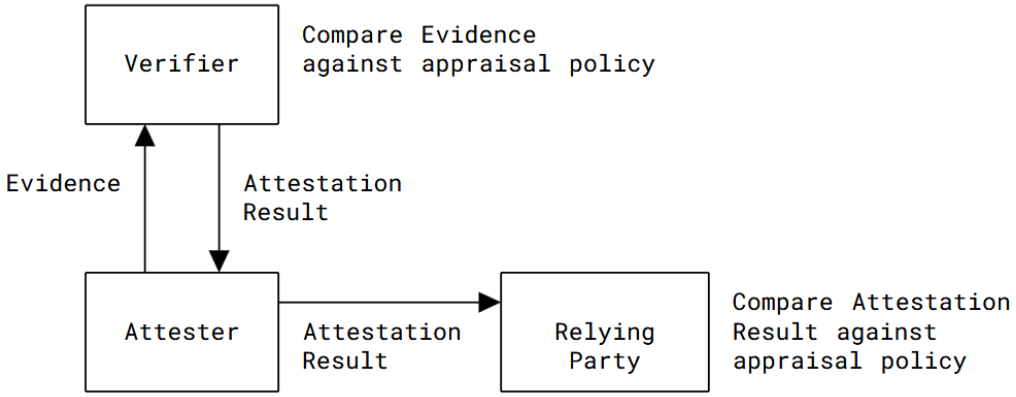


Figure 3.4: Passport Model (Source: [11])

1. the Verifier may not issue a positive Attestation Result if the Evidence does not meet the appraisal policy;
2. the Relying Party may reject the Attestation Result if it does not meet their policy;
3. the Verifier may be unreachable or unavailable.

The Attestation Result can be transferred between the Attester and Relying Party through a resource access protocol, which dictates its serialization format. The Evidence's format is only constrained by the Attester-Verifier attestation protocol. As a result, interoperability and standardization are more critical for Attestation Results than for Evidence.

Background-Check Model

In this model, the Attester provides Evidence to the Relying Party, which forwards it to the Verifier without processing as shown in figure 3.5.

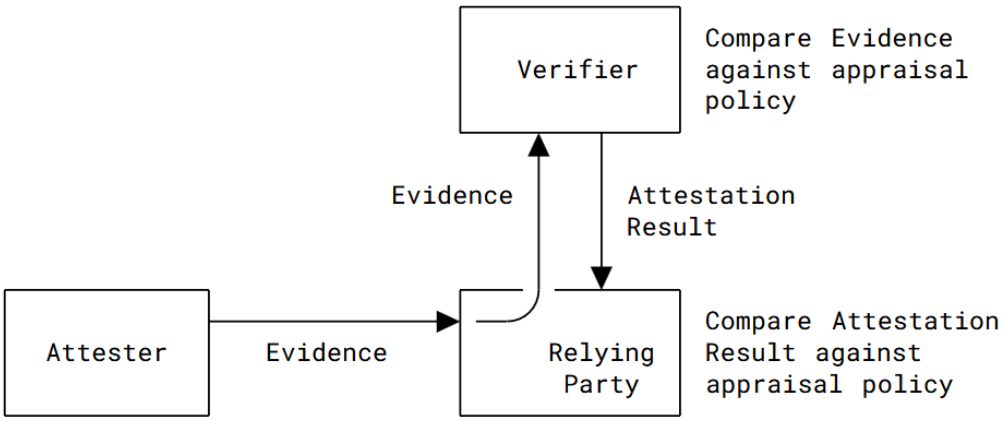


Figure 3.5: Background-Check Model (Source: [11])

The Verifier checks the Evidence against its appraisal policy and returns an Attestation Result to the Relying Party, which then evaluates it against its own policy. The resource access protocol between the Attester and Relying Party includes only the Evidence, which is not processed by the Relying Party.

The Evidence can use any serialization format since the Relying Party does not need to interpret it, as it is simply forwarded. However, the Attestation Result, which the Relying Party

does consume, must be in a format that is compatible with the protocol already supported by the Relying Party. Using a serialization format that the Relying Party already understands minimizes the code footprint and reduces the attack surface, which is especially important if the Relying Party is a constrained node.

Combinations

In a variation of the Background-Check Model, the Relying Party and the Verifier are on the same machine, performing both functions together, eliminating the need for a protocol between them.

The choice of model depends on the specific use case, and different Relying Parties may adopt different topological patterns. A device may use different models to provide Evidence for various Relying Parties or use cases, such as one model to gain network access and another to access confidential data. Both models can be used simultaneously by the same device.

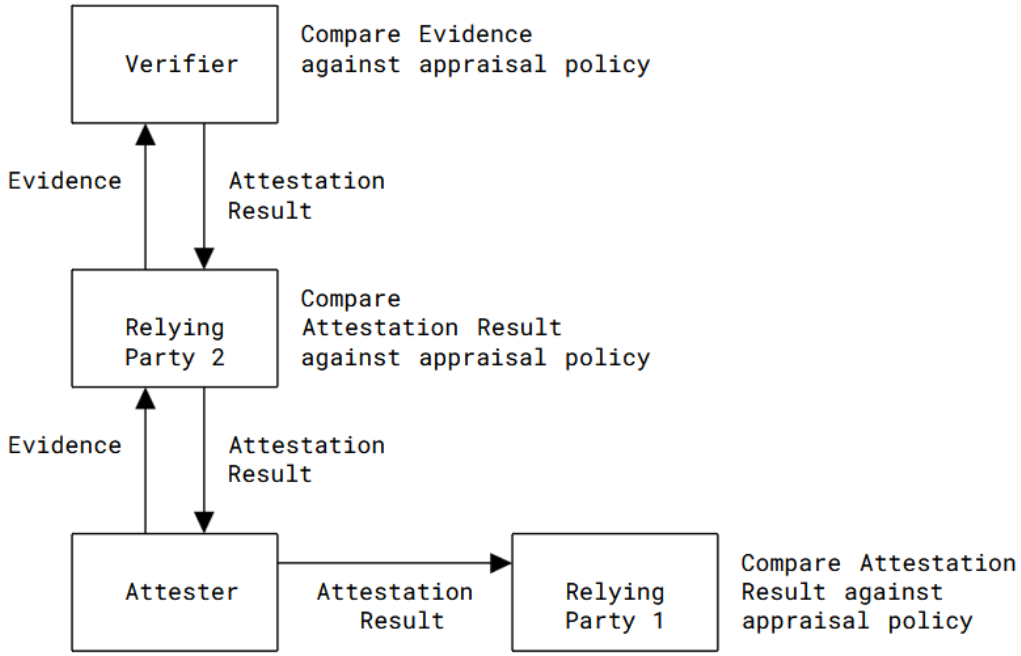


Figure 3.6: Example Combination (Source: [11])

An example is given in figure 3.6 where Relying Party 1 uses the Passport Model, while Relying Party 2 uses an extension of the Background-Check Model. In this extended model, Relying Party 2 provides the Attestation Result back to the Attester, allowing the Attester to use it with other Relying Parties.

3.2.4 Attestation Results

Attestation Results are used by the Relying Party to determine the level of trust it places in an Attester and whether to grant it access or permission to perform certain actions. These results can either indicate simple compliance (a boolean value) or include detailed Claims about the Attester, which the Relying Party evaluates based on its own appraisal policy. The quality of the Attestation Result depends on the Verifier's ability to assess the Attester, and different Attesters can produce results with varying levels of strength being qualitatively different in strength [13].

An Attestation Result that shows non-compliance may suggest that the Attester should not be authorized or needs remediation, and in some cases, the Evidence itself may be deemed unauthenticated. By default, the Relying Party does not trust the Attester but may grant access if

the Attestation Result satisfies its appraisal policy. This policy can range from simple full access to more complex evaluations of the information in the Attestation Result.

Attestation Results can contain detailed, potentially privacy-sensitive information about the Attester, and unlike device-specific Evidence, they can be vendor-neutral, especially if the Verifier can generate agnostic information. This allows simpler, standard-based appraisal policies for the Relying Party, supporting interoperability. Unlike Evidence, which is signed by the device, Attestation Results are signed by the Verifier, so the Relying Party only needs a trust relationship with the Verifier for its appraisal policy.

3.2.5 Freshness

This subsection discusses the importance of determining the "epoch" (point in time) when Evidence or an Attestation Result is produced. This is crucial for Verifiers or Relying Parties to assess whether the Claims are still current and reflect the latest state of the Attester, based on the most recent Appraisal Policy. No protocols are defined but guidance on how to ensure freshness in protocols is provided.

Freshness is an architectural decision that impacts protocol design, including the number of required interactions and overall interoperability. Freshness is evaluated using the Appraisal Policy, which compares the estimated epoch against an "expiry" threshold. However, there is a potential race condition where the Attester's state or policies could change immediately after the Evidence is generated.

A flexible freshness requirement can facilitate the caching and reuse of Evidence and Attestation Results, which is particularly useful in resource-constrained environments. Below are presented three common methods for determining the epoch of Evidence or an Attestation Result.

Explicit Timekeeping

The first approach for determining the epoch is the usage of synchronized and trustworthy clocks, including a signed timestamp [14] with the Claims in the Evidence or Attestation Result. Timestamps can also be applied to individual Claims to differentiate the generation times of the overall Evidence or Attestation Result and each Claim.

The trustworthiness of the clock is typically established through Endorsements and requires additional Claims about the signer's time synchronization. However, in some use cases, a trustworthy clock may not be available, such as in certain Trusted Execution Environments (TEEs), where the clock is only accessible outside the TEE and cannot be trusted by it.

Implicit Timekeeping

This approach assigns the responsibility of timekeeping to the Verifier (for Evidence) or the Relying Party (for Attestation Results), especially when the Attester lacks a reliable clock or time synchronization is compromised. There are two possible ways to implement implicit timekeeping: one using *nonces* and another that relies on *epoch IDs*.

In the first case, the appraising entity sends an unpredictable nonce, which is then signed and included with the Claims in the Evidence or Attestation Result. By verifying that the sent and received nonces match, the entity confirms that the Claims were signed after the nonce was generated. This process associates a "rough" epoch with the Evidence or Attestation Result, where the epoch is considered rough because it applies to the entire set of Claims rather than individual ones, and the time between Claim creation and collection is unclear.

In the second approach, an *epoch ID distributor* periodically sends epoch identifiers (IDs) to both the sender and receiver of Evidence or Attestation Results. Unlike nonces, epoch IDs can be reused and are not tied to specific timestamps, meaning they don't need to convey time information or be monotonically increasing. The approach helps associate a "rough" epoch without requiring

a trustworthy clock or time synchronization, relying only on the epoch ID distributor to manage the timing.

The appraising entity compares the epoch ID in received Evidence or Attestation Results to the most recent one received from the epoch ID distributor to determine if it's within the current epoch. Solutions like counters, retries, or message buffering may be used to handle race conditions when transitioning to a new epoch.

The appraising entity keeps an *epoch window* of recent epoch IDs to prevent false negatives (i.e., discarding valid Evidence as outdated). The window's depth depends on network delays and epoch duration. Unlike the nonce approach, the epoch ID approach minimizes state management, regardless of the number of Attesters or Verifiers, as long as they use the same epoch ID distributor.

Considerations

Hybrid timekeeping mechanisms combine implicit and explicit approaches. For example, if clocks in the Attesting Environment are trustworthy but unsynchronized, a nonce-based exchange can first determine the relative time offset between peers, followed by timestamp-based exchanges.

It's important to note that the values in Claims might have been generated well before they are signed. In such cases, the signer is responsible for ensuring that the values are still fresh when signed. For instance, values created at boot time could be stored securely until network connectivity is established and a nonce is obtained.

3.3 Linux Integrity Measurement Architecture (IMA)

The *Linux Integrity Measurement Architecture* (IMA) is a component of the *Linux kernel*. It includes three main components: *IMA-Measurement*, *IMA-Appraisal*, and *IMA-Audit*. These features are activated according to the actions defined in the *IMA Policy* rules [15].

IMA consists of various modules that offer distinct integrity functions [16]:

- **collect**, measures a file prior to access it;
- **store**, records the measurement in a list maintained by the kernel, and if a hardware Trusted Platform Module (TPM) is available, *Extends* the IMA PCR;
- **attest**, if a TPM is present, uses it to sign the IMA PCR value, enabling remote verification of the measurement list;
- **appraise**, enforces local validation of the measurement by comparing it to a trusted value stored in the file's extended attribute;
- **protect**, protects the file's security extended attributes (including the appraisal hash) from offline attacks;
- **audit**, monitors and logs the file hash values.

Components

IMA-Measurement maintains a log of measurement events and calculates an overall integrity value for this log in a PCR, if the platform includes a TPM chip. Typically, it utilizes PCR 10. A TPM attestation quote serves as a signature over the PCR, indirectly ensuring the integrity of the measurement event log. This feature requires both a TPM and an independent verifier. Measurement is similar to the pre-OS trusted boot process. The initial measurement is the boot aggregate, which is a hash of TPM PCRs 0-9. IMA keeps a record of the measured hash values. If the same hash appears again, it is typically not re-measured (this behaviour can be set in the IMA Policy). IMA Templates define the format and content of each entry collected and stored in

PCR	template-hash	template-name	file-hash	file-path
10	91f3[...]e127	ima-ng	sha1:1801[...]f6b3	boot_aggregate
10	8b16[...]486a	ima-ng	sha256:efdd[...]9954	/init

Table 3.1: Example of two IMA-Measurement log entries with the `ima-ng` template.

the measurement log. The default template in use nowadays is ima-ng, an example is illustrated in the table 3.1 below.

The fields have the following meanings:

- **PCR**, specifies which PCR is used to store the extended hash values;
- **template-hash**, contains the digest calculated from the file fields (file-hash and file-path) of the current entry, which is then extended into the PCR specified in the PCR field;
- **template-name** identifies which IMA template was used during the measurement;
- **file-hash**, ensures the file's integrity by providing a cryptographic hash of the file's contents, representing the digest of the actual event;
- **file-path**, indicates the file's location on the system for tracking and verification.

IMA-Appraisal can verify a file's digital signature or hash and deny access if the signature is invalid or the hash does not match a trusted value stored in an extended attribute. Appraisal is comparable to the pre-OS secure boot process. The IMA-Appraisal feature operates locally and does not require a TPM or an external verifier.

IMA-Audit enhances the system's audit log by including the file hash, which can be useful for analytics and forensic investigations [15].

IMA policy is configured on the attester (the system with a TPM) through a text file located at /etc/ima/ima-policy. It specifies which files on the system will be measured and the conditions under which the measurement will be repeated [17].

Verifying Ima Measurements

The verification process has two main phases [18]:

1. **Measurement Validation**, a process in which each template-hash value in each entry in the log (corresponding to a specific event or file) is (re)calculated and compared against a *whitelist* of acceptable values (Figure 3.7);
2. **Measurement Log (ML) Validation**, a process in which the *extend* operation is performed entry by entry, verifying the ML; for each entry, its *template-hash* is recalculated from the other fields and compared to the stored template hash to check the integrity; the *template-hash* is then extended with the accumulated hash of previous entries, replicating the actual *extend* operation done over the IMA PCR; the initial value extended is the boot aggregate result, starting with a zero-byte sequence, the size of which depends on the specific hash algorithm used by the PCR bank; the final result can be compared with the PCR storing the IMA aggregate (usually `PCR 10`) (Figure 3.8).

IMA can be considered both as an S-RTM and a D-RTM. It functions as an S-RTM by validating the ML with measurements taken during system boot. Additionally, it acts as a D-RTM by allowing verification of applications executed during normal system operation.

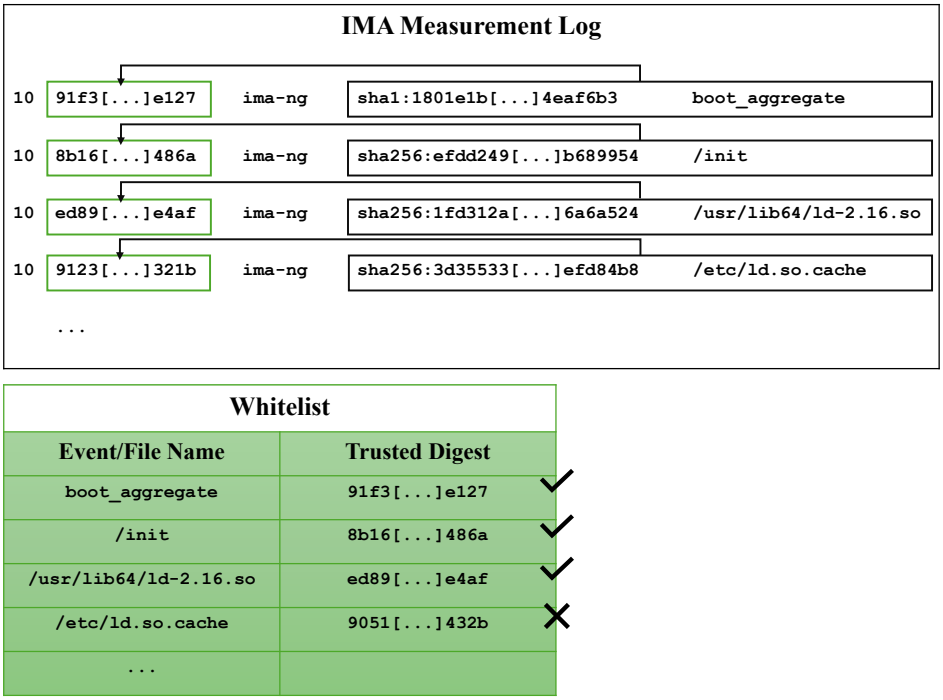


Figure 3.7: Example of IMA Single Measurement Validation

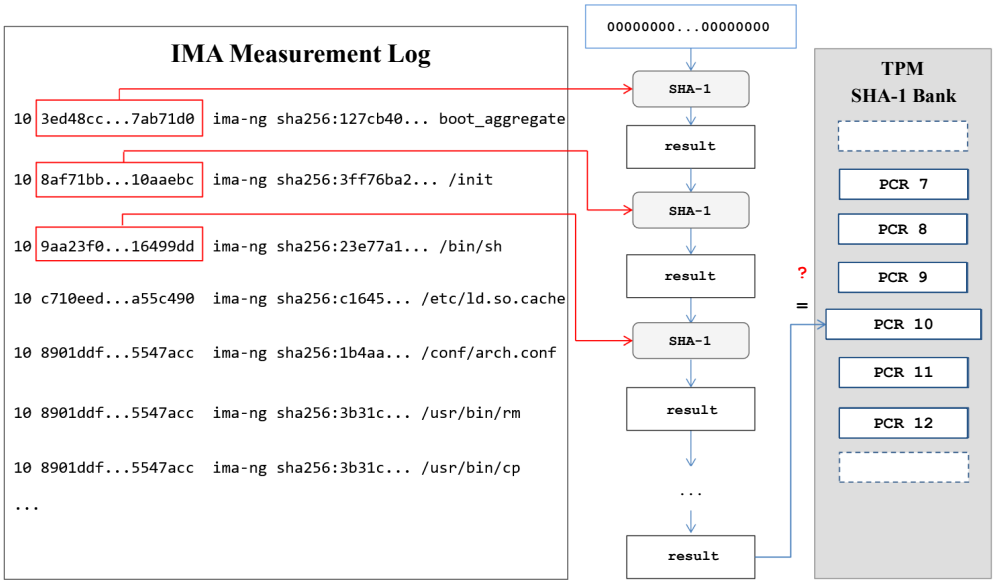


Figure 3.8: Example of IMA Measurement Log Validation (Source: [19], [18])

Chapter 4

Attestation Frameworks

With the rise of Cloud Computing, computations and data storage have shifted from local to cloud environments. While this simplifies many operations, it also comes with risks related to cloud infrastructure administrators and its multi-tenancy nature. As a result, it's crucial to verify the trustworthiness of cloud platforms. Similarly, the increasing use of IoT devices and cyber-physical systems presents risks because manufacturers often prioritize cost and size over security. Connecting these devices to the Internet offers benefits but also exposes them to attacks. Detecting if a device is compromised may be more effective than focusing solely on prevention to mitigate these risks.

A common solution for both cloud and IoT security is *Remote Attestation* (RA), which allows an external entity to verify a system's integrity through an unforgeable proof of its correct configuration. Several organizations, including the *Trusted Computing Group* (TCG) [1], *IETF* [20], and *ETSI* [21], are working to standardize RA procedures. The process of verifying these integrity proofs is handled by a Verification Service, which relies on up-to-date data from authoritative sources to assess the system's status. However, due to the variety of devices involved (from IoT devices to large servers), each deployment requires a tailored service, making RA systems complex and costly [22].

Keylime originated from the security research team at MIT's Lincoln Laboratory and was introduced to the scientific community in December 2016 through the whitepaper titled "Bootstrapping and Maintaining Trust in the Cloud" [23]. It is now a project hosted by the Cloud Native Computing Foundation (CNCF) [24] that offers an open-source solution for both establishing hardware-rooted cryptographic identities for cloud nodes and monitoring the integrity of those nodes through periodic attestation [25].

Veraison [26] is an open-source project focused on developing software components that can be used to build Attestation Verification Services. The name *Veraison* (pronounced "ver-ayy-sjon.") refers to the stage in winemaking when grapes begin to ripen and require regular checks to determine when they are ready for harvest. This term was chosen for the project because it aligns with the project's goal of verifying integrity attestation (rather than grapes). Veraison's flexible design allows it to be easily adapted to meet specific contextual needs, providing customizable Verification Services [22].

4.1 Keylime

To overcome the limitations of current methods, the combination of trusted computing and IaaS is seen as a way to provide a hardware root of trust that tenants can use to establish trust in both the cloud provider's infrastructure and their own systems running on it. The aim is to reduce reliance on the cloud provider and carefully consider all the trade-offs required to enable trusted computing services [23].

Keylime's *Threat Model* is based on the following assumptions:

- Cloud Provider's considerations:
  - semi-trusted, i.e. trusted organizationally but susceptible to compromise or malicious insiders;
  - cloud provider has technical controls and policies to limit the impact of compromise;
- Adversary's capabilities:
  - control over part of the infrastructure, some fraction of cloud provider's resources may be compromised (e.g., rogue system admin controlling specific racks);
  - an adversary can monitor or manipulate compromised parts of the cloud network or storage;
  - no physical tampering, adversary cannot physically tamper with host components like CPU, memory, TPM, etc...;
- Security in Virtualized Environments:
  - cryptographic keys stored in VM memory are protected;
  - hypervisor provider does not deploy hypervisors that can spy on tenant VM memory;
  - TPM and system manufacturers create valid endorsement credentials, verified via signed certificates;
- Adversary's Goal:
  - attacker seeks persistent access to tenant systems to steal, disrupt, or deny data/services;
  - attacker needs to modify the code loading or running process for persistence;
  - load-time and runtime integrity checks (hypervisor, kernel, applications) will detect such modifications.

4.1.1 Architecture

Keylime is made of the following components [27]:

- **Agent**, a service that operates on the operating system being attested; it interacts with the TPM to enrol the *Attestation Key* (AK), generate quotes, and gather necessary data, such as UEFI and IMA event logs, to enable state attestation; once the device is successfully attested for the first time, the agent provides an interface for further provisioning using the secure payload mechanism;
- **Registrar**, oversees the agent enrollment process; this process involves assigning a `UUID` to the agent, collecting the EK_{pub}, EK certificate, and AK_{pub} from the agent, and verifying that the AK is associated with the EK (using `MakeCredential` and `ActivateCredential`); once the agent is registered with the registrar, it is prepared for attestation enrollment; the tenant can use the EK certificate to verify the TPM's trustworthiness;
- **Verifier**, carries out the actual attestation of an agent and sends revocation messages if the agent exits the trusted state; once an agent is registered for attestation (either through the tenant or directly via the API), the verifier continuously retrieves the necessary attestation data from the agent; this data may include a quote over the PCRs, PCR values, NK public key, IMA log, and UEFI event log; after validating the quote, additional validation of the data can be configured;
- **Tenant**, a command-line management tool provided by Keylime to manage agents; it allows adding or removing agents from attestation, validating the EK certificate against a certificate store, and checking the status of an agent; additionally, it provides the necessary tools for the payload mechanism and handling revocation actions.

The core of Keylime's operation is the *Registration Protocol*, which ends with the identification of the system where the Agent is instantiated, making it the subject of attestation by the Verifier.

Registration Protocol

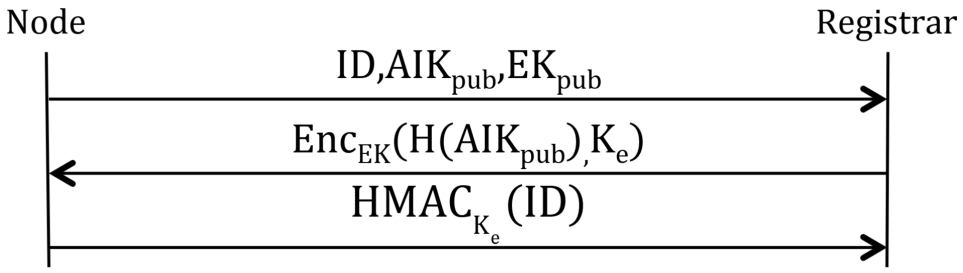


Figure 4.1: Physical Node Registration Protocol (Source: [23])

The *Registration Protocol*, depicted in figure 4.1, can be summarized in the following steps [23]:

1. the Agent sends its `UUID` along with the public elements of the EK and *Attestation Identity Key* (AIK) to the Registrar; the EK `X.509` certificate is also sent and it contains the EK which is immutable and authenticates a genuine TPM, while the AIK is used to sign the quote; the AIK is a unique RSA key stored in the TPM and is used for platform authentication through the TPM's attestation feature;
2. the Registrar checks the validity of the EK certificate, verifying that it was issued by a trusted TPM manufacturer and then creates a challenge to be sent back to the Agent; the challenge is made of an ephemeral key (K_e) and a hash calculated over the public part of the AIK (AIK_{pub}), both encrypted with the EK_{pub};
3. the Agent will solve the challenge properly by using its private part of the EK and the AIK (respectively EK_{priv} and AIK_{priv}) that will be given as input, together with the challenge, to the `ActivateCredential` command of the TPM; if the command succeeds, the proper decrypted K_e will be used to calculate a keyed-hash message authentication code (HMAC) over the `UUID`;
4. the Registrar re-computes the $\text{HMAC}_{\text{K}_\text{e}}$(UUID) and checks it against the received one, if they are equal, it marks the Agent as active, stores the AIK of the active Agent and will later use it to validate the remote attestation procedure.

4.2 Veraison

Veraison is a set of libraries and tools designed to improve consistency in developing a verification service. The architecture of Veraison is organized as depicted in figure 4.2: the *Attestation Scheme* includes an *Endorsement Handler* and an *Evidence Handler* that are used respectively for providing endorsements and trust anchors, and performing verifications as well (i.e. by interacting with the *Provisioning* and *Verification* frontends). The *Veraison Trusted Services* (VTS) backend delivers essential services to the Verification and Provisioning components (it is the actual component that interacts with the evidence handler). The *key-value store* (K-V store) serves as Veraison's storage layer, handling both endorsements and trust anchors. Different methods can be used for this, depending on the configuration.

4.2.1 Configuration

The configuration of each Veraison component is contained within a *ParamStore*, which holds the parameter definitions. Each parameter definition specifies the type of the parameter, used for

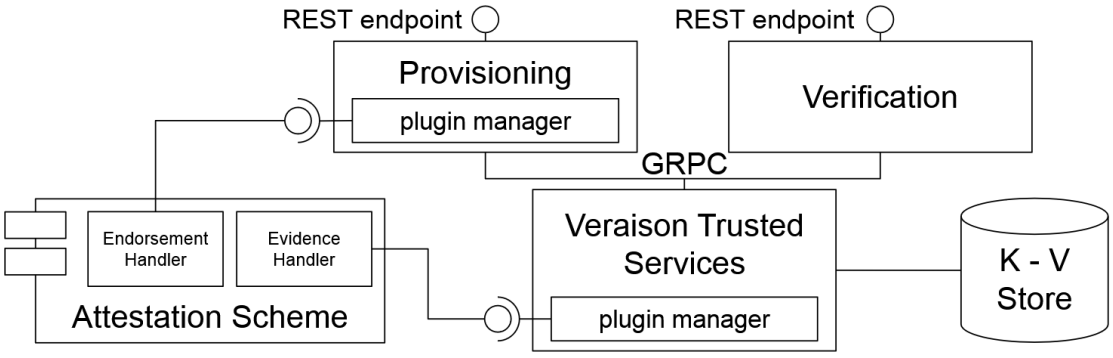


Figure 4.2: Veraison Summary Scheme (Source: [22])

validation, as well as its configuration path. The overall configuration for a Veraison deployment is managed by a *Config*, which aggregates the ParamStores for the components included in that deployment. ParamStores are created and initialized with parameter definitions specific to their respective components. These ParamStores are then provided to the Config, which assumes responsibility for them and populates them with values retrieved from external sources. The values are validated according to the defined parameters. After validation, the ParamStores are passed to the `Init()` methods of the components (figure 4.3).

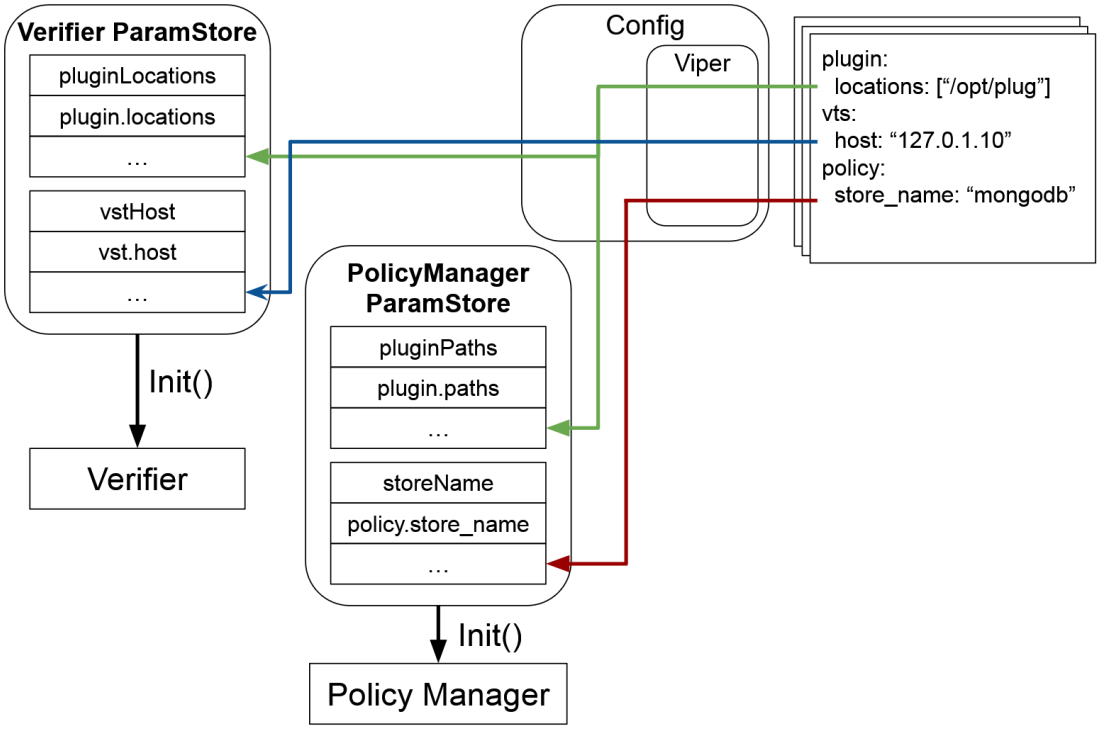


Figure 4.3: Veraison Verifier Configuration Scheme (Source: [22])

Veraison services use Viper [28] for configuration management. Viper is an open-source library for configuration handling in Go applications. In this context, the implemented config loader adds a validation layer on top of Viper, allowing pre-processing of the configuration obtained by Viper before it is applied. This ensures thorough validation before any further processing takes place.

4.2.2 Provisioning

Providing endorsements and trust anchors to the Veraison service is crucial. All data is transmitted in the *Coincise Reference Integrity Manifest* (CoRIM) format [29]. CoRIM is a signed document in CBOR format (COSE) [30]. *Concise Binary Object Representation* (CBOR) [31] is a data format that adjusts encoding type and message size without requiring negotiation. The underlying data model is an extended version of `JSON` [32]. Composite devices are represented through a combination of concise module identifiers (CoMID) and concise software identifiers (CoSWID), which are included within the CoRIM document. The CoRIM data are processed, decoded, and stored (figure 4.4). The Provisioning service is responsible for decoding, while the Veraison Trusted Services component generates the keys and stores the provisioned data.

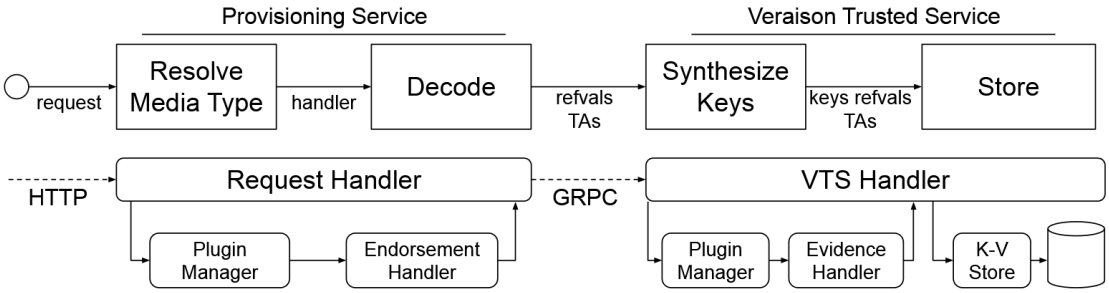


Figure 4.4: Veraison Provisioning Scheme (Source: [22])

4.2.3 Verification

The Verifier receives an *Entity Attestation Token* (EAT) [33] and generates an *EAT Attestation Result* (EAR) [34], which indicates whether the token is properly structured and has been verified against the provisioned endorsements (figure 4.5).

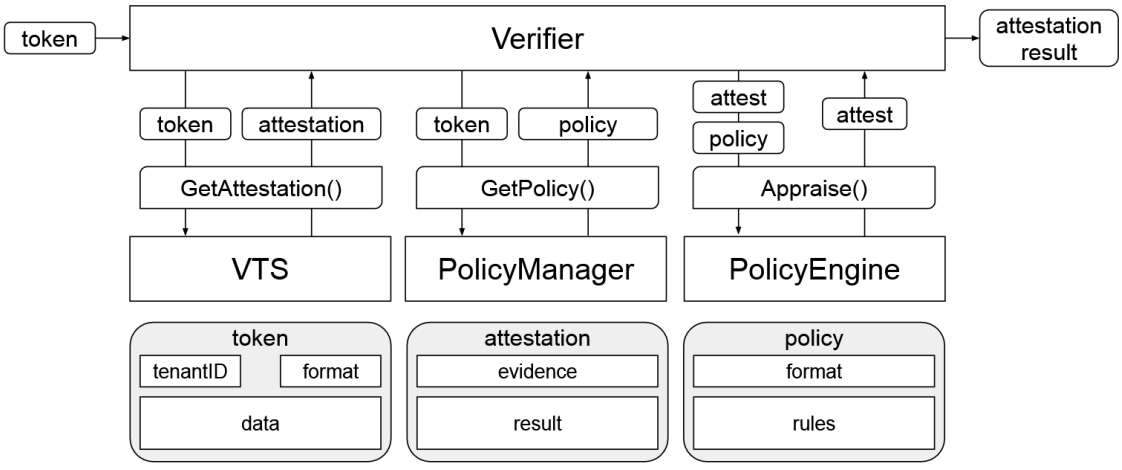


Figure 4.5: Veraison Verification Scheme (Source: [22])

An EAT contains a set of attested claims that describe the state and characteristics of an entity. This entity could be a device, such as a smartphone, IoT device, or network equipment. The set of claims within an EAT is used by a relying party, a server, or a service to assess the type and level of trust to be placed in the entity. EATs exist in two formats: as a *CBOR Web Token* (CWT) [35] or a *JSON Web Token* (JWT) [36]. Both formats include attestation-oriented claims. The EAR is used by the Verifier to encode the outcome of evaluating an Attester's evidence. The Verifier then returns a signed attestation result as an EAT document. Most of the complex processes are handled by the Veraison Trusted Services (VTS) component (figure 4.6).

Veraison Trusted Services (VTS)

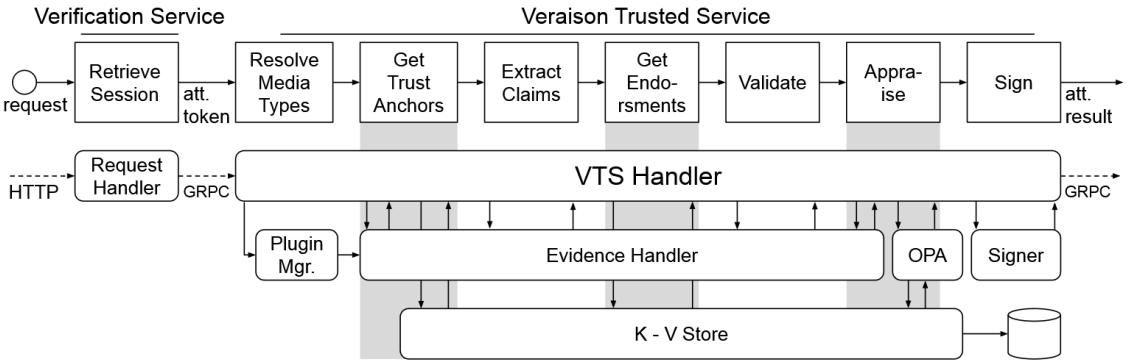


Figure 4.6: VTS Verification Scheme (Source: [22])

The *Veraison Trusted Services* (VTS) component (figure 4.6) can run within the same process as the Verifier or in a separate process (for high-trust isolation). It consists of:

- **Schemes**, that contain the knowledge of how to parse a specific token to extract evidence, how to validate the token against a trust anchor, and how to evaluate the evidence against endorsements to populate the *Trust Vector* (a method to indicate the trust degree of a component);
- A **Trust Anchor Store**, used to maintain provisioned trust anchors (e.g., keys, certificates) for token validation;
- An **Endorsement Store**, to store the provisioned endorsements, which serve as "golden values" for verifying claims or may include additional claims that are associated with, but not part of the token (e.g., certification information).

Its main duties are summarized as follows:

1. Evidence Extraction, extracts the evidence from the token;
2. Verification, verifies the token against the trust anchor;
3. Endorsement Retrieval, retrieves associated endorsements;
4. Trust Vector Population, assesses evidence against endorsements based on the attestation scheme to populate the Trust Vector;
5. Policy Application, if applicable, the Verifier applies the registered policy to the attestation, updating the Trust Vector accordingly;
6. Attestation Result Creation, initializes and generates an attestation result.

4.3 Considerations

After evaluating the Keylime and Veraison frameworks, Veraison has been opted for due to its alignment with modern attestation standards and its modular, flexible design.

While Keylime offers a robust attestation mechanism and has been widely used in cloud security, it presents several significant limitations that make it less suitable for a standard-based, scalable solution. Firstly, Keylime does not natively implement the Remote ATtestation procedureS (RATS) standard, which is a critical requirement for interoperability and future-proofing

attestation solutions. Compliance with the IETF RFC-9334 standard is essential for ensuring compatibility across different platforms, and Keylime lacks built-in support for this framework.

Another major drawback of Keylime is its limited scalability and adaptability. Due to its design, modifying its workflows and adapting it to diverse environments requires extensive modifications. This makes it difficult to integrate with a variety of platforms and infrastructures, particularly in heterogeneous IoT and cloud ecosystems, where flexibility is paramount. Additionally, Keylime's modularity is constrained, making it challenging to add or remove components for additional functionalities. Unlike Veraison, which provides a well-structured and extensible attestation service, Keylime requires significant re-engineering to accommodate new use cases.

Moreover, Keylime's workflow modification process is complex and rigid. Changing formats or adjusting attestation procedures within Keylime is not straightforward, as it relies on predefined mechanisms that are not easily extensible. This presents a significant hurdle for organizations or research initiatives aiming to implement custom attestation policies or integrate new verification mechanisms. Additionally, managing different attestation models within Keylime is more cumbersome, whereas Veraison provides a customizable architecture that can easily accommodate multiple attestation schemes.

In contrast, Veraison emerges as a more suitable solution due to its high degree of modularity and its adaptability to different environments. Unlike Keylime, Veraison is built to support RATS-compliant workflows, ensuring future-proof compatibility with evolving attestation standards. Additionally, its architecture is highly flexible, making it an excellent choice for integrating new attestation methods, adapting to diverse hardware and software configurations, and supporting a broad range of use cases.

For these reasons, the decision to move forward with Veraison instead of Keylime was based on a combination of technical and strategic considerations. Standard compliance, scalability, modularity, and ease of integration were the key factors influencing this choice. By leveraging Veraison, this thesis ensures the development of a secure, standard-based Remote Attestation framework that is both scalable and adaptable to future cybersecurity challenges.

Chapter 5

Design

Now that all the primary concepts about the *Remote Attestation* procedure have been presented, it is possible to dive into the proposed solution. The architecture is presented in figure 5.1.

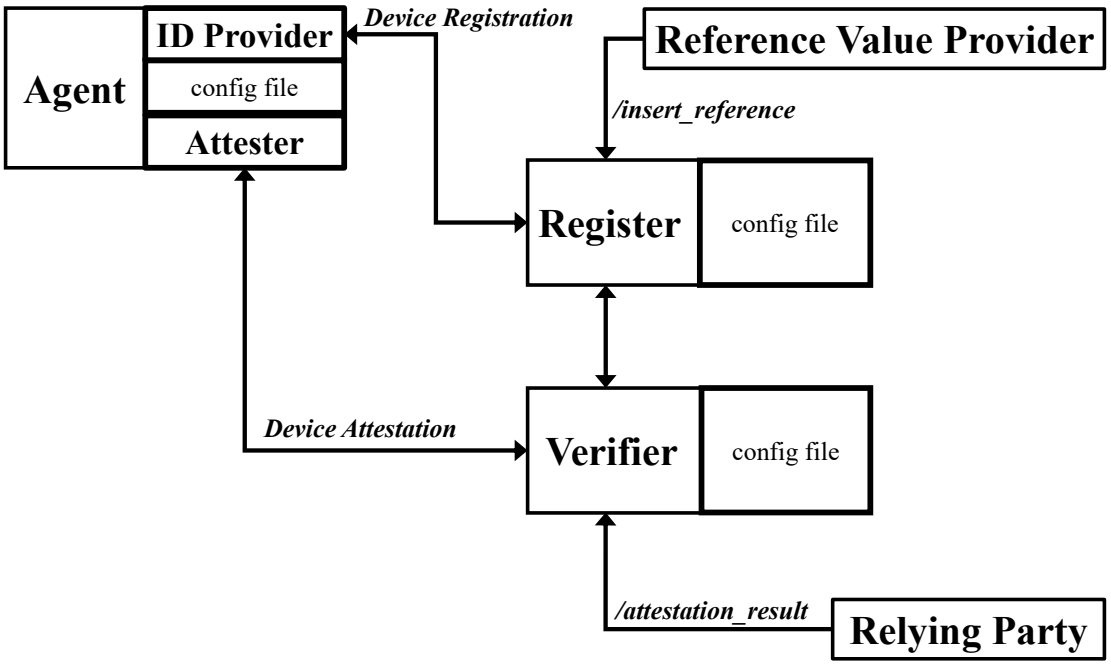


Figure 5.1: Architecture

It has been decided to consider the device willing to perform the remote attestation (*Agent*) composed of two modules, the *ID Provider* and the *Attester*. The former is in charge of performing the *Device Registration*, in order to give an *ID* to the agent that will be used to store some important information related to it (i.e. IP address, authentication key), while the latter is used to perform the *Device Attestation* procedure and is meant to happen after the registration.

From the verification side, there are two separate entities involved, one called *Register*, to which the agent's ID Provider sub-module talks in order to perform the registration, and the *Verifier*, involved during the device attestation (thus interacting with the agent's Attester sub-module). Register and verifier can communicate to exchange some useful information, but this communication happens through some data stored in two different databases, which will be discussed later.

All of the modules that have been presented have a *configuration* file (*config file* in figure 5.1)

through which is possible to define some important parameters (like the internal sub-network's IP, the hash algorithm used for the quotes, the templates for the key hierarchies, the period of attestation, ...).

To let the *Reference Value Provider* give new golden values, a specific `API` can be invoked (`/insert_reference`). The same happens for the *Relying Party* when the attestation results are needed, in this case, the `API` to call is `/attestation_result`. Now let's dive into the device registration and attestation's details.

5.1 Device Registration

The *Device Registration* is a procedure performed to verify that the device has a valid TPM with a valid EK certificate and also to let the agent prove the possession of the private key related to the EK certificate and resident to the TPM.

In the proposed solution, there are two ways to start this procedure:

- initiated by the agent;
- initiated by the register.

5.1.1 Initiated by the Agent

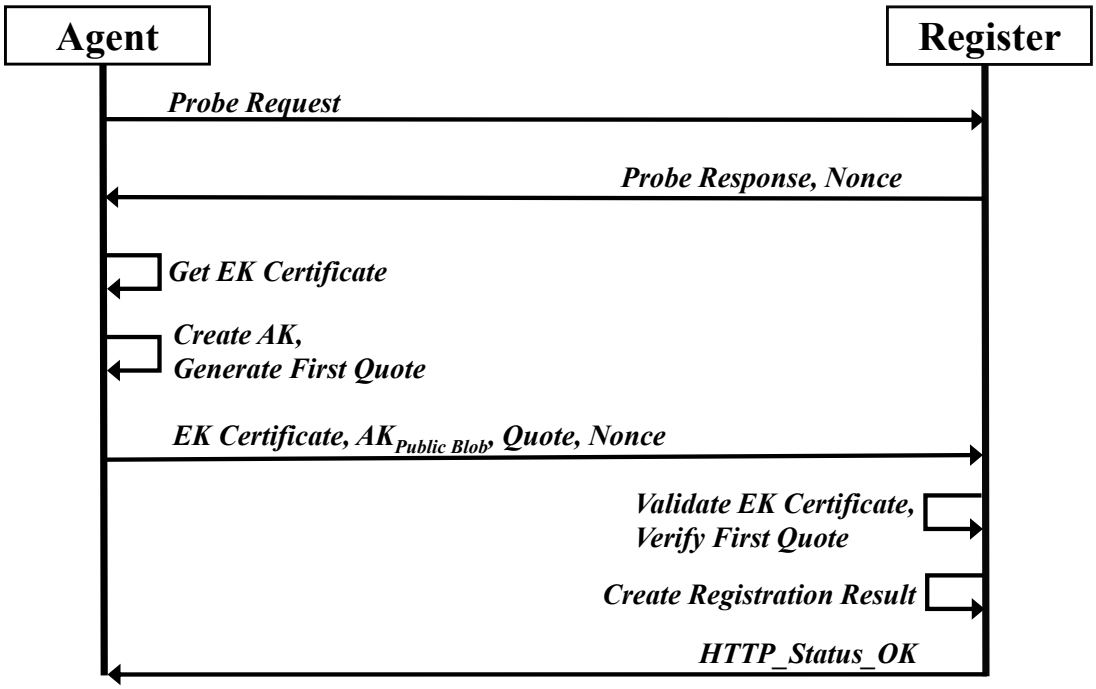


Figure 5.2: Device Registration Procedure Initiated by the Agent

As illustrated in the figure 5.2, the agent starts the procedure by sending a *Probe Request* to the register which replies with a *Probe Response* and a *Nonce* to be used in combination with the *First Quote* to guarantee the freshness of the data and protection against replay attacks.

So, at this point the agent will interact with the TPM in order to retrieve the *EK Certificate*, to create the *AK* to be used for the attestation procedure and to generate the *First Quote* which

is a quote calculated over the first ten `PCRs` (from 0 to 9) and is also called *Identity Quote* and signed with the just created AK.

Then the agent sends these data to the register, in particular, the $AK_{Public\ Blob}$ is sent, which is an object that contains the byte encoding of the public area of the AK, not only the public part of the key couple.

Now the Register will perform the *Validate EK Certificate* procedure in which the certificate's issuer is compared against a set of stored trusted issuers and the certificate's chain is validated. Then there is the verification of the first quote and the generation of the *Registration Result*. If every procedure is executed properly, an `HTTP_Status_OK` code is returned to the agent.

5.1.2 Initiated by the Register

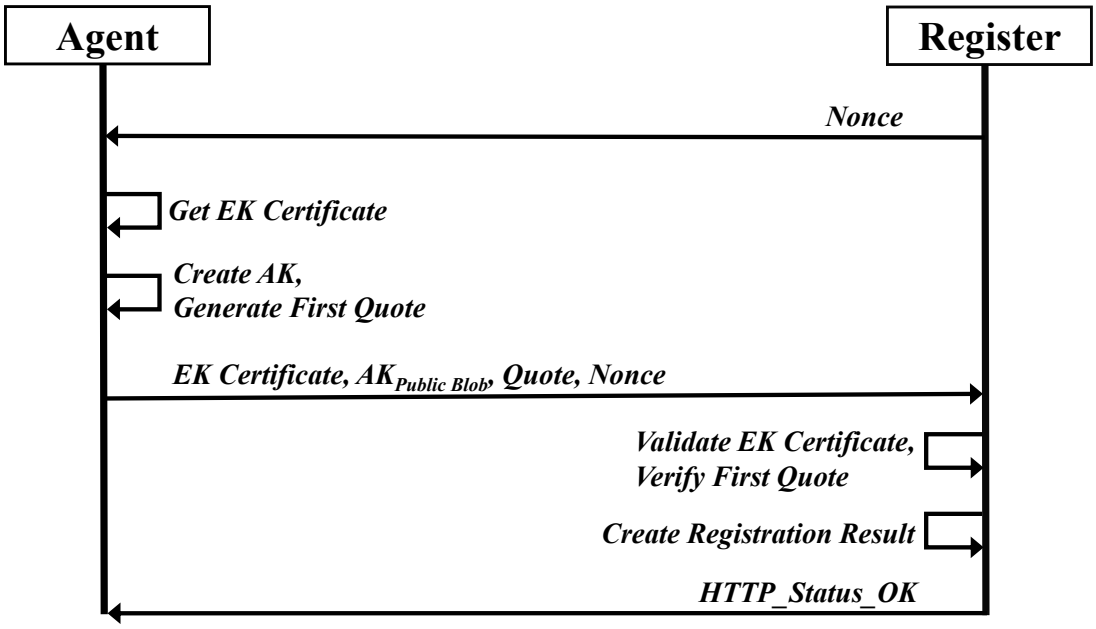


Figure 5.3: Device Registration Procedure Initiated by the Register

In this case (figure 5.3), the procedure is started by the register when he sends a *Nonce*. So, both entities are waiting and when the registration needs to start, a command from the outside is given to the Register to contact the proper agent. From this point on, the operations performed are the same as discussed in the sub-section 5.1.1.

The order in which the procedure is initiated can be set in the configuration files.

Continuation

The rest of the procedure is shown in figure 5.4, there is no distinction in which entity has initiated the device registration.

The agent interacts again with the TPM in order to retrieve the $AK_{Public\ Blob}$, the public part of the EK (EK_{Public}) and the $AK_{Name\ Data}$. This object is a hash calculated over the public blob of the AK and is used in the first stage of the second part of this procedure to guarantee that the public blob has not been altered. The $\mathrm{EK}_{\mathrm{Public}}$ is sent by the register to generate the challenge that will prove the possession of the private part in the TPM installed by the agent.

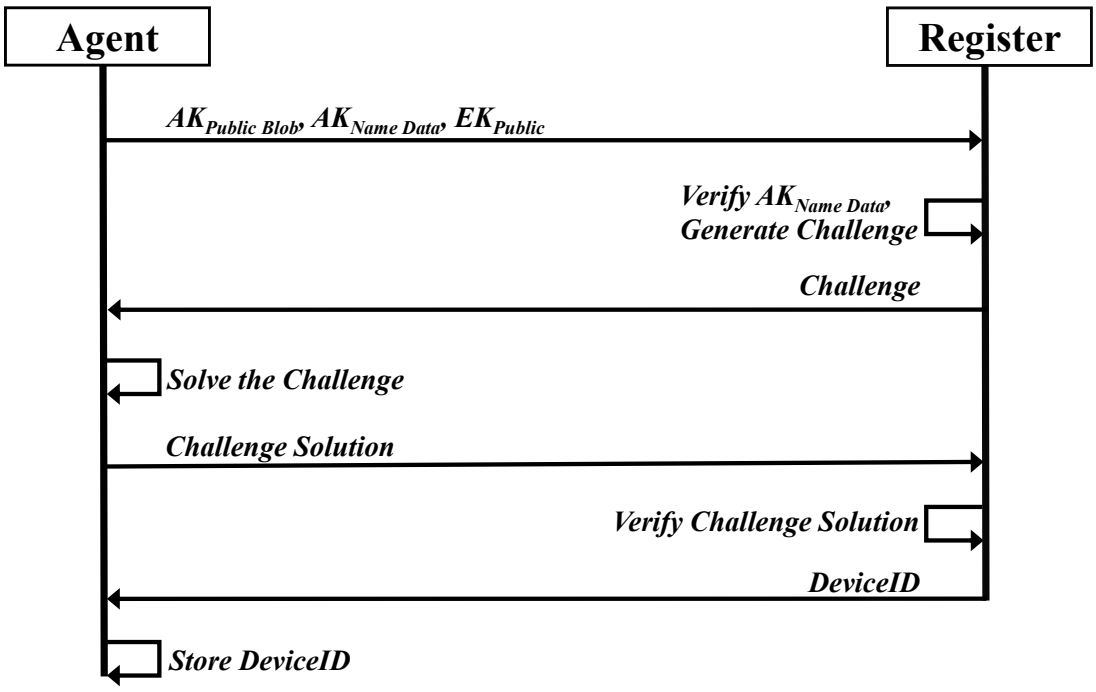


Figure 5.4: Continuation of the Device Registration Procedure

Now the register verifies the name data by recalculating the hash over the public blob and checking if it matches the received data. If so, it calculates the *Challenge* which is basically a secret encrypted with the EK_{Public}, and sends it back to the agent.

If the agent has the proper private part of the EK, then it will be able to solve the challenge and return to the register the *Challenge Solution* that will be verified by it.

After all these operations have been executed properly, the register generates a *DeviceID* (i.e. `UUID` [37]) and sends it to the agent that stores it inside a *Non-Volatile* register within the TPM (*Store DeviceID*).

5.2 Device Attestation

Following the proper registration of the device, there is the *Device Attestation* procedure in which a device is already assigned an ID and will use it to be recognized by the verifier. In this case, the procedure can also be started by both the agent and the verifier.

5.2.1 Initiated by the Agent

First of all, the agent needs to be acknowledged, so it interacts with the TPM to retrieve the *DeviceID* that was saved in a non-volatile register at the end of the registration phase. This ID is then sent to the verifier.

The verifier, after receiving the ID, sends back a *Nonce* used in order to guarantee the freshness of the data and to avoid reply attacks.

Now the agent interacts again with the TPM regain the data needed to create the *Attestation Object*, to which the received nonce is attached.

The verifier receives the *Attestation Object*, at first it performs some checks on the *AK* used to sign it, then it verifies the attestation object itself (against some "golden values" as well), and

eventually it creates the *Attestation Result* and returns an `HTTP_Status_OK` code to the agent. Details on the attestation object are given in the following chapter.

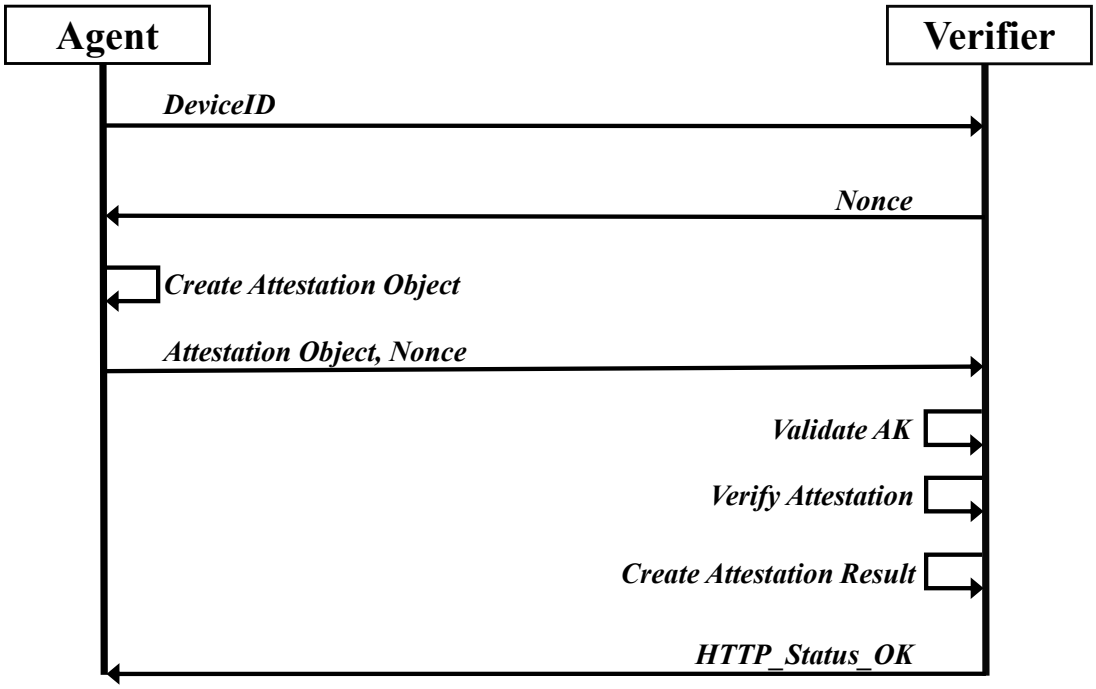


Figure 5.5: Design Attestation Procedure Initiated by the Agent

5.2.2 Initiated by the Verifier

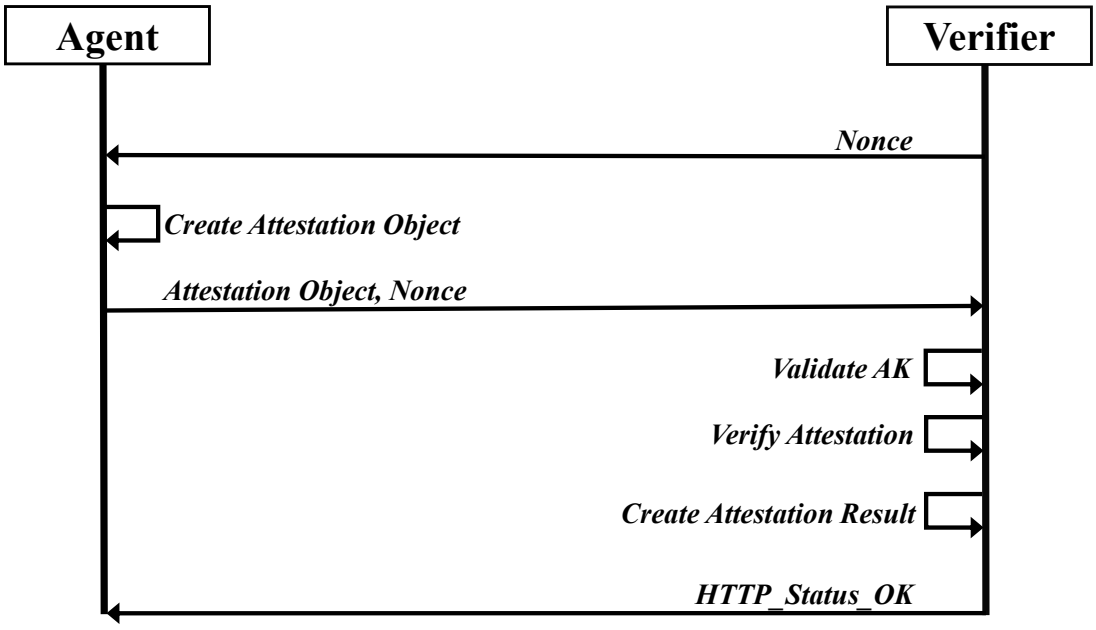


Figure 5.6: Design Attestation Procedure Initiated by the Verifier

Here the procedure is initiated by the verifier that sends a *Nonce* (figure 5.6). It is implicit that in this case, the verifier is somehow being told from the outside the device that needs to be attested so it does not wait to be contacted. More details about this are given in the next section. The rest of the procedure is the same as the one described in the sub-section 5.2.1.

Similar to the device registration, the procedure's start order can be set in the configuration files.

Chapter 6

Implementation

This chapter aims to present more details about how the design described previously has been implemented in practice. More information is given about the communication that happens between the *Register* and the *Verifier* as well as the procedures (*Device Registration* and *Device Attestation*) needed to perform the remote attestation.

First of all, the focus is on the communication between the register and the verifier and the data that are exchanged. As it is possible to notice in figure 6.1 there is not a direct connection between the two entities but they rely on two different databases.

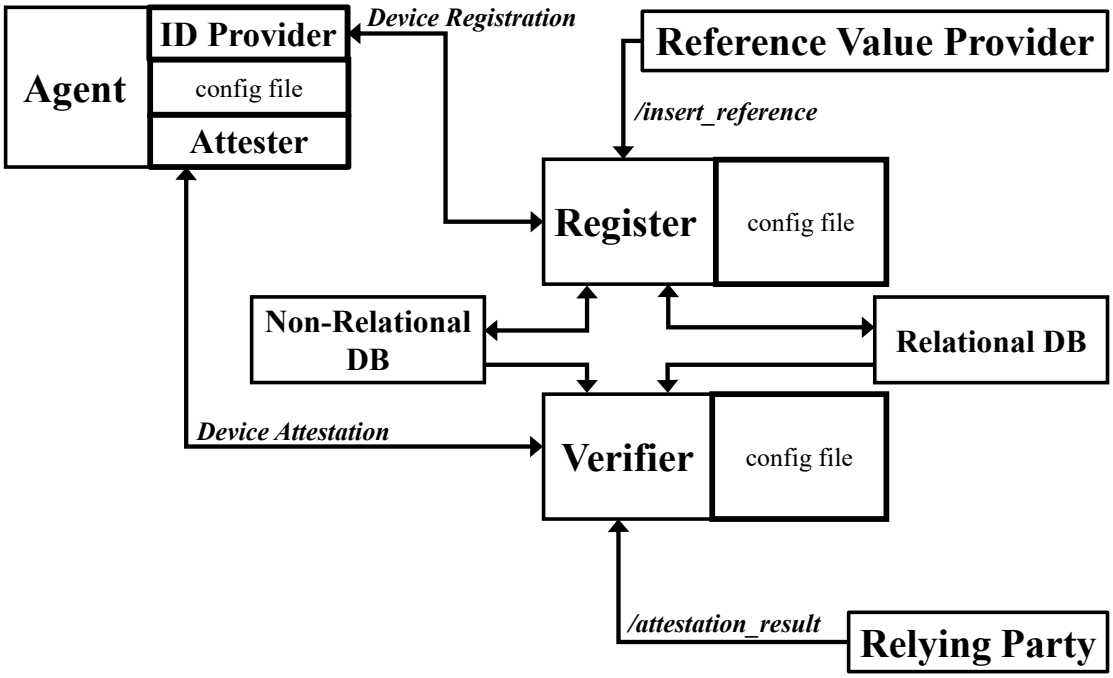


Figure 6.1: Implementation of the Architecture

The *Relational DB* is used to store information about the agents while the *Non-Relational DB* is used in order to store the reference values (i.e. "golden values") that will be used during the *Device Attestation* procedure. This choice has been made because the information about the agents that needs to be available is always the same while the reference values are more variable and a non-relational database can handle them in an effortless way.

During the registration phase, the register needs to ensure that the TPM within the device is trusted. To do so, it needs two tables in the relational database, one related to the trusted TPM certificates with the following fields:

- **certificate ID**, to recognize locally the certificate;
- **common name** of the subject;
- **certificate** in PEM format [38];

and one related to the trusted TPM vendors with the following data:

- **vendor ID**, to recognize locally the TPM vendor;
- **name**, basically the brand (e.g. AMD, Intel, Huawei, ...);
- **TCG identifier**, as stated in the "TCG TPM Vendor ID Registry" [39];
- **platform model**;
- **firmware version**.

So, once the device registration has been properly performed, the register will store the following information in the relational database:

- **device ID**, to make the device recognizable locally;
- **IP address** and **port number**, to contact the agent;
- **authentication key**, to perform validation before going into the attestation procedure;
- **whitelist ID**, an ID assigned to access the reference values which the device refers to during the attestation procedure;
- **state**, to keep track of the device's state and put remedies if needed.

For how much concerns the non-relational database, it stores only two pieces of information:

1. **whitelist ID**, needed for the same purpose explained for the relational database instances;
2. **reference values**, a huge list against which the values sent by the agent will be compared.

The register has reading and writing permissions to the *Relational DB* as well as to the *Non-Relational DB*. The verifier has only reading permission to retrieve the information stored in both databases. The following sub-sections explore in more depth how the *Device Registration* and *Device Attestation* procedures have been implemented. More details on these two databases can be found in the appendix A.

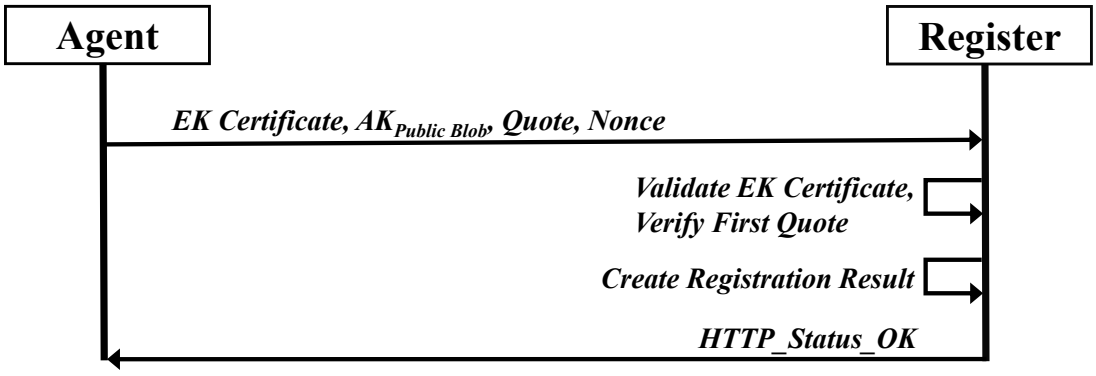


Figure 6.2: Device Registration's end of First Phase

6.0.1 Device Registration

In this sub-section the objective is to explore the operations that are performed on the data received by the register during the *Device Registration* procedure. As shown in figure 6.2, the focus is given on the main operations executed whether the agent or the register starts the procedure.

In particular, the register receives the *EK Certificate*, the $AK_{Public\ Blob}$ and the *First Quote*. At first, some checks are performed over the certificate and for this purpose, the relational database is accessed. The operations are illustrated in figure 6.3.

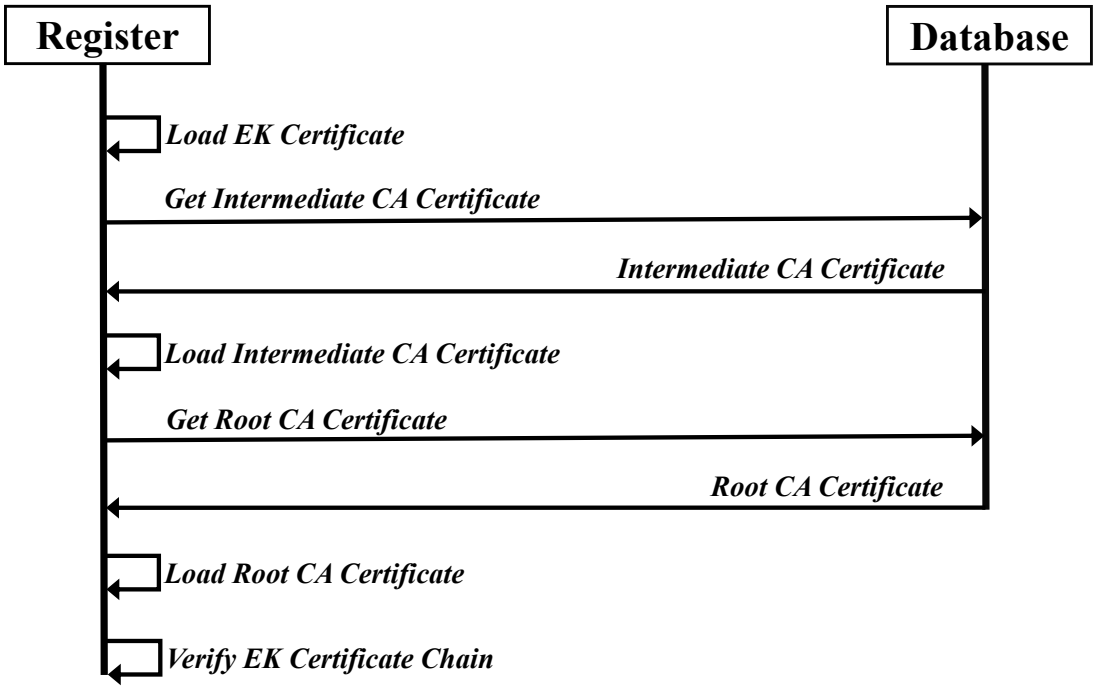


Figure 6.3: Interactions with the Relational DB during the EK Certificate Validation in the Device Registration Procedure

To work more suitably, some operations named "load" are performed on the certificates in which their internal format is changed. After the *EK Certificate* is loaded, the database is accessed in order to retrieve the *Intermediate CA Certificate* and the *Root CA Certificate*, which are loaded and are vital elements in the verification of the certificate chain.

During the verification of the certificate chain, the database is re-accessed to get the TCG identifier, the TPM platform model and the TPM firmware version. In particular, these fields are retrieved from the *EK Certificate* to perform a query toward the relational database to check if the TPM is among the trusted ones. If no result is given, then the agent is considered untrusted.

After the operations on the certificate are properly completed, there is the verification of the *First Quote*. This verification comprises two phases:

1. verify if the hash algorithm used for signing the quote is among the ones supported by the TPM;
2. signature verification of the quote performed with the AK_{Public} (taken from the received $AK_{Public\ Blob}$);

The last two operations are *Create Registration Result*, where it is stated that the boot phase of the device has been approved, and the sending of the `HTTP_Status_OK` to the register that will proceed with the registration.

Then, as shown in figure 5.4 of the previous chapter (subsection 5.1.2), the registration procedure continues. The register receives the $AK_{Public\ Blob}$, the $AK_{Name\ Data}$ and the EK_{Public} as it needs to verify that the received AK is resident on a proper TPM that is owned by the agent (so, that it possess the $EK_{Private}$). To accomplish this operation, the register at first verifies the $AK_{Name\ Data}$, which is the hash of the $AK_{Public\ Blob}$, then it generates a challenge which is basically the encryption of a secret performed with the received $EK_{Public\ Key}$. This is needed because the EK can only be used to encrypt/decrypt data, as if it was used also for signing, it would cause privacy concerns as it is directly linked to a TPM and so to the device on which the TPM is mounted.

The challenge is sent back to the agent that, if it is in possess of the $EK_{Private}$, is able to solve it and send the *Challenge Solution* that is then verified by the register. If this verification goes fine, the *DeviceID* is assigned to that agent in the database, the state is updated (`Registered`, `Registration Failed`). More details can be found in appendix A.

6.0.2 Device Attestation

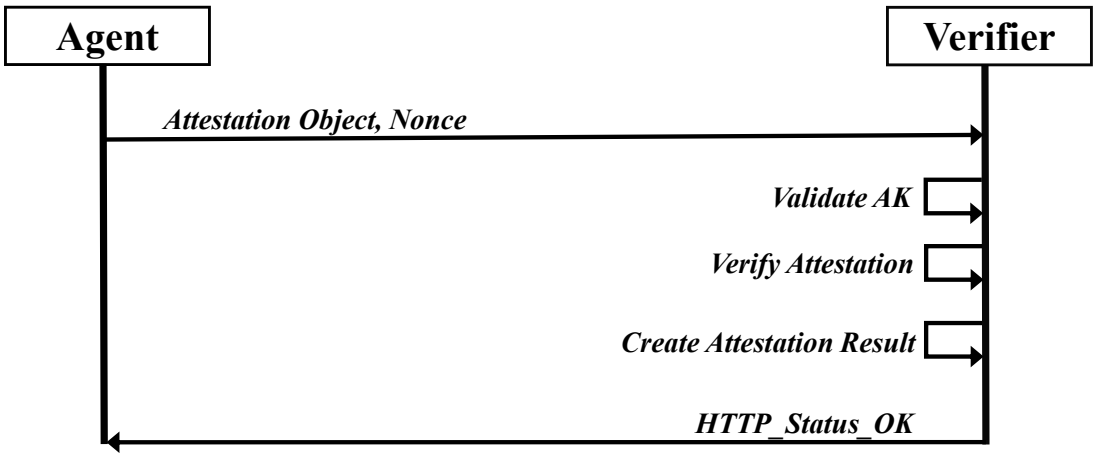


Figure 6.4: Device Attestation's End

Now, the implementation of the *Device Attestation* is discussed. Figure 6.4 shows the main operations executed independently from who starts the procedure. At the beginning, the agent sends the *Attestation object* in which the previously received *Nonce* is inserted.

The *Attestation Object*, as stated in the RFC-9864 [40] standard, contains the following data:

- the AK used to sign the quotes, in particular the public area;
- the quotes;
- the event log;
- the integrity measurement architecture file (Section 3.3).

And three optional fields:

- the instance info (information about a Google Compute Engine [GCE] instance, unused outside of GCE);
- the AK Certificate;
- the intermediate certificates for verifying the AK Certificate, optional;
- the trusted execution environment attestation, a secondary platform attestation that the machine is running within a particular confidential environment.

On receiving the *Attestation Object* and the *Nonce*, the verifier validates the AK, which means that it accesses the relational database to check if there is a device with an ID associated with the AK_{Public} of the object (figure 6.5).

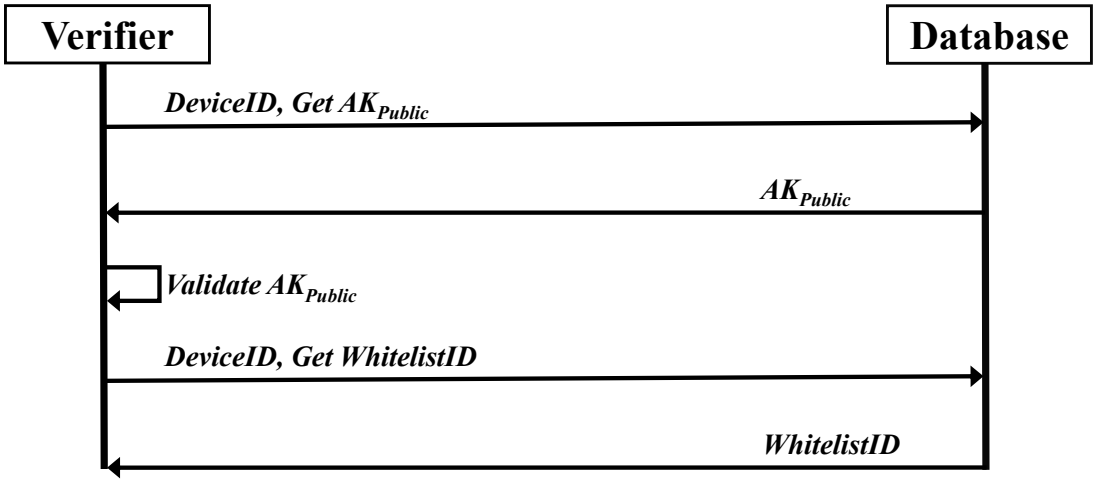


Figure 6.5: Interaction with the Relational DB before and after the AK Validation in the Device Attestation Procedure

After the AK validation, there is the verification of the *Attestation Object*. The following checks are performed:

- the provided signature is generated by the trusted AK;
- the signature signed the provided quotes data;
- the quote data starts with `TPM_GENERATED_VALUE`;
- the quote data is a valid `TPMS_QUOTE_INFO`;
- the quote data was taken over the provided PCRs;
- the provided PCR values match the quote data internal digest;
- the provided *Nonce* matches the one in the quote data;
- the provided event log matches the provided PCR values.

Then, in order to check the integrity measurement architecture (IMA) file, the reference values are involved. To get these "golden values" the verifier needs to contact the non-relational database and submit the *WhitelistID* received at the end of the registration phase (figure 6.6). However, before that, the relational database is accessed to recover the proper *WhitelistID* associated with the device (figure 6.5).

The IMA verification comprises the operations discussed in the sub-section 3.3 to assure its integrity. In the end, an *Attestation Result* is created and an `HTTP_Status_OK` code is sent back to the device.

The result can be accessed with the API `/attestation_result` (figure 6.1) by the relying party, but this is also used by the register to update the state of the device in the relational database (`Attested`, `Attestation Failed`) in accordance to the *Attestation Result*.

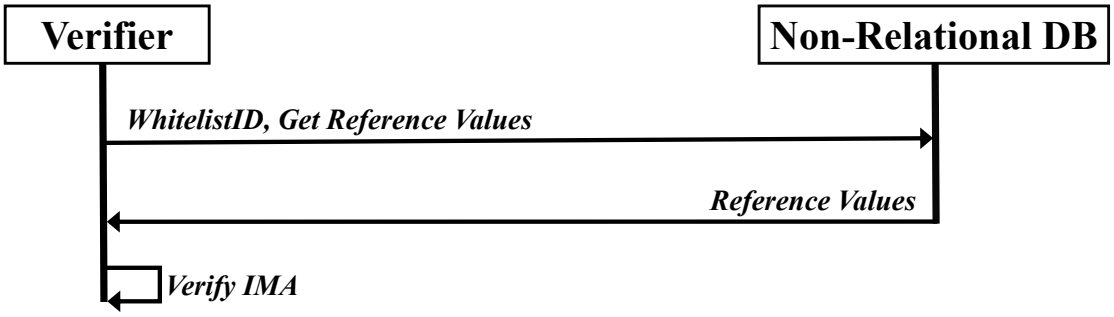


Figure 6.6: Interaction with the Non-Relational DB after the AK Validation in the Device Attestation Procedure

6.0.3 Standards Adoption

The entire implementation has followed the architecture and the specifications described in the RFC-9334 [11]. The data structure of the attestation object has followed the one described in the RFC-9864 [40]. Moreover, also the data types used for the reference values and the attestation results are compliant to them, as well as to the *Veraison Project* [22].

The reference values are provided to the register in CoRIM format [29]. These are also handled and stored by the register using the same format and by assigning an ID following the UUID [37] standard. This standard is adopted also to assign an identifier to the agents after performing the registration properly.

The attestation results are available in the form of *Entity Attestation Result* (EAR) [34]. In particular, the verifier signs an *Entity Attestation Token* (EAT) [33] created as a *JSON Web Token* (JWT) [36].

Chapter 7

Tests

Comprehensive tests have been performed to ensure that the implementation meets the requirements depicted in the design. This chapter presents the approach and methodology employed to perform functional and performance tests.

The tests have been executed on separate machines, which are two Intel NUC devices, each equipped with an *Intel Core i5-5300U* processor, 16 GB of RAM and an integrated TPM 2.0 chip. The two environments are configured as follows:

- a Verifier node running on Debian GNU/Linux 12 (Bookworm) with kernel version 6.1.0;
- an Agent node running on Ubuntu 22.04 LTS (Jammy Jellyfish) with kernel version 6.8.0-52-generic.

7.1 Functional Tests

The functional tests outlined in the following subsections aim to verify the proper operation of the functions needed to perform the main attestation tasks. The tests have been executed taking into account the different possible states in which the machines can be found.

7.1.1 Reference Value Provisioning

Before performing any kind of checks, the Reference Values against which the IMA file is validated are needed. These reference values are formatted in CoRIM format [29]. This test is divided into two main cases:

1. reference values formatted properly;
2. reference values formatted in the wrong way.

Correct Reference Value Formatting

To send the reference values properly, a Reference Value Provider must send a POST request to the node that executes the register module of the Verifier endpoint `/insertreference`. This endpoint is expecting a JSON object [32] with the syntax written as follows. It is possible to notice that the "environment" variable which stores the identifying information about a target or attesting environment at the class, instance and group scope, can also be null.

```
{
    "environment": {},
    "measurements": [
        {
         "value": {
            "digests":
            ["sha-256;e2Q2sMmPYjgIZtlDLCrw7gjOFqFxvaaVGuzZXuEwfWE="],
            "filename": "boot_aggregate"
            }
        },
        {
         "value": {
            "digests":
            ["sha-256;2dF3XWQ/b3ChpvZG3+AjBSdl9VihZ+xYlRrS8QE7PkY="],
            "filename": "/init"
            }
        },
        {
        "value": {
            "digests":
            ["sha-256;9fJdoyH5NBRiIF2P1uyRi7qJrV6l3J9th11RdNGTZCI="],
            "filename": "/usr/bin/sh"
            }
        },
        {
        "value": {
            "digests":
            ["sha-256;WVvNwwaZlxGlqceH5dAWGPlOuSxMCaF46HWjJPEHLYk="],
            "filename": "/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2"
            }
        }
        ...
    ]
}
```


Listing 7.1: Example of Valid Reference Value Measurements

The "filename", which is vital for IMA validation, is added as an extension to each measurement value, as actually there is no other variable intended to store this piece of information in the structure defined by Veraison.

After the measurement validation has been successfully completed, an ID is assigned to the reference values. Here is the message written in the logs:

```
register | 2025/02/12 15:23:21 Inserted object with the following
    whitelistID: 50111534-e42f-49df-bfd4-90622cbf1ec9
```

Listing 7.2: Register logs, successful insertion

Wrong Reference Value Formatting

To perform this test, it has been necessary to create some wrong reference value measurements to take into account the failure of the validation that happens when they are received by the API `/insertreference` of the Register endpoint at the Verifier side. The validation can fail in two cases:

1. empty measurements;
2. hash value of the wrong length.

In the following there are two examples with the respective logs messages.

```
{
    "environment": {},
    "measurements": []
}
```

Listing 7.3: Example of Empty Reference Value Measurements

```
register | 2025/02/12 16:35:05 Error validation new Reference Value: no
    measurement entries
```

Listing 7.4: Register logs, insertion failure caused by empty measurements

```
{
    "environment": {},
    "measurements": [
        {
         "value": {
            "digests":
                ["sha-256;e2Q2sMmPYjgIZtlDLCrw7gjOFqFxvaaVGuzZXuEwfWE="],
            "filename": "boot_aggregate"
            }
        },
        {
         "value": {
            "digests": ["sha-256;2dF3XWQ/b3ChpvZG3+AjBSdl9VihZ+xY"],
            "filename": "/init"
            }
        },
        {
        "value": {
            "digests":
                ["sha-256;9fJdoyH5NBRiIF2P1uyRi7qJrV6l3J9th11RdNGTZCI="],
            "filename": "/usr/bin/sh"
            }
        },
        {
        "value": {
            "digests":
                ["sha-256;WVvNwwaZlxGlqceH5dAWGPlOuSxMCaF46HWjJPEHLYk="],
            "filename": "/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2"
            }
        }
        ...
    ]
}
```

Listing 7.5: Example of Reference Value Measurements with one wrong measurement

```
register | 2025/02/12 16:45:27 Error validation new Reference Value:
    measurement at index 1: length mismatch for hash algorithm sha-256:
    want 32 bytes, got 24
```

Listing 7.6: Register logs, insertion failure caused by measurement with wrong hash value length

7.1.2 Registration

Now that hypothetically the measurements of the golden values have been sent to the register, the agent can perform the Registration. To do so, the agent needs to contact the register of

the Verifier endpoint with different APIs after exchanging the nonces properly. The first API is needed to validate and verify the first quote, the second to create the challenge and send it back to the agent, and the third to verify it.

Proper First Quote Object

At first, the agent sends the first quote together with the EK certificate, the Nonce and the whitelist ID assigned to the golden values previously by the register. The agent must send a POST request to the node that executes the register module of the Verifier endpoint `/firstquote`, which is expecting a JSON object structured as follows:

```
{
    "EKCertificate": "-----BEGIN CERTIFICATE-----
    MIIElTCCA32gAwIBAgIEQzo1xzANBgkqhkiG9w0BAQsFADCBgzELMAk
    GA1UEBhMC\nREUxITAfBgNVBAoMGEluZmluZW9uIFRlY2hub2xvZ2ll
    BBRzEaMBgGA1UECwwR\nT1BUSUdBKFRNKSBUUE0yLjAxNTAzBgNVBAM
    EluZmluZW9uIE9QVElHQShUTSkg\nUlNBIE1hbnVmYWN0dXJpbmcgQ0
    hUnNhTWZyQ0EwMDAzMB4XDTE2MDEwMTEzMDk1MloXDTMxMDEw\nMTEz
    [...]
    xwpMtDI+3g6l+qlWMu8AeI5GGSzbSeLcUB\nMthS7hCMrO4tT6TUVTO
    sCACMvx3XMRiRix5+BEtt2i3tDd33Z9fW1M2RYY4\n2OUyNrIcmP6x/
    O2exZD8d2Y1THbyjYE63nP6zY92RdV3+OO5YAM9BMyJHAu4D6mz\n2K
    YAM9BMyJHAu4D6mz\n2KvvPF99qjnUR06D/NaI+VZFc/eFio5TZoA==
    -----END CERTIFICATE-----
    ",
    "Quote": [
        {
            "quote": "/1RDR4AYACIAC1K+2mQSGaq95am/72ZHyjO1CCVsFGLbJ...",
            "raw_sig": "ABYACwEADsYB8CwoRRFUIxrNzKl6NlmmAv5OkIw3Yuv...",
            "pcrs": {
                "hash": 11,
                "pcrs": {
                    "0": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "1": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "2": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "3": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "4": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "5": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "6": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "7": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "8": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "9": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="
                }
            }
        },
        ...
    ],
    "AKPubBlob": "AAEACwAFBHIAAAAQABYACwgAAAAAAAEAl9FwiS15df12T9xQB...",
    "Nonce": "uuN7blRrOAzz7LuDxRl1Ivr34Dke/DLQsdNQWqqyC6o=",
    "WhitelistID": "c90486dc-b218-4cd0-a420-3bd484cecac4"
}
```

Listing 7.7: Example of properly structured First Quote object

It is valuable to notice that the "Quote" element inside the JSON object is an array, so the quotes can be calculated considering different hash algorithms and appended inside this array. It is the register's duty of the Verifier side to set the proper hash algorithm's quote to be considered during its validation.

At the end of this API, when properly executed, it will be sent an `HTTP_STATUS_OK` code to inform the agent that it can proceed with the call of the next API.

Wrong First Quote Object

For these tests, wrong values of the "quote", "pcrs", "AKPubBlob", "Nonce" and "WhitelistID" elements have been considered. The quote can be signed in three manners, using:

1. `RSASSA PKCS#1 v1.5` [41];
2. `RSASSA PSS` [41];
3. `ECDSA with curve P-256` [42].

If the quote is wrong, three different messages are written in the logs based on the used algorithm.

```
register | 2025/02/18 15:28:27 Failed to verify the first quote:
    RSASSA-PKCS1v1.5 signature verification failed: crypto/rsa:
    verification error
```

Listing 7.8: Register logs, wrong quote signed with RSASSA PKCS#1 v1.5

```
register | 2025/02/18 15:30:11 Failed to verify the first quote:
    RSASSA-PSS signature verification failed: crypto/rsa: verification
    error
```

Listing 7.9: Register logs, wrong quote signed with RSASSA PSS

```
register | 2025/02/18 15:32:17 Failed to verify the first quote: ECC
    signature verification failed
```

Listing 7.10: Register logs, wrong quote signed with RSASSA PSS

For the pcrs, a case in which fewer PCRs are selected has been considered. So, a JSON object with the pcrs element inside the "Quote" built as follows:

```
{
    ...,
    "Quote": [
        {
            "quote": "/1RDR4AYACIAC1K+2mQSGaq95am/72ZHyjO1CCVsFGLbJ...",
            "raw_sig": "ABYACwEADsYB8CwoRRFUIxrNzKl6NlmmAv5OkIw3Yuv...",
            "pcrs": {
                "hash": 11,
                "pcrs": {
                    "0": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "1": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "2": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "3": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "4": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "5": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "6": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="
                }
            }
        },
        ...
    ],
    "AKPubBlob": "AAEACwAFBHIAAAAQABYACwgAAAAAAAEAl9FwiS15df12T9xQB...",
    "Nonce": "uuN7blRrOAzz7LuDxRl1Ivr34Dke/DLQsdNQWqqyC6o=",
    "WhitelistID": "c90486dc-b218-4cd0-a420-3bd484cecac4"
}
```

Listing 7.11: Example of First Quote object with fewer PCRs

The related message written in the logs is the following:

```
register | 2025/02/18 16:51:02 Failed to verify the first quote: given
    PCRs and Quote do not have the same PCR selection
```

Listing 7.12: Register logs, wrong quote with unexpected number of PCRS

The AKPubBlob contains public information related to the AK, not only the public part of the key couple. When it is received by the register it is decoded to get the public part of the AK used to verify the first quote. If AKPubBlob is not correct, the following error message is shown in the logs:

```
register | 2025/02/18 16:58:23 Failed decoding public blob: decoding
    TPMT_PUBLIC: unexpected EOF
```

Listing 7.13: Register logs, wrong AKPubBlob

The nonce is used to guarantee the freshness of the data and to avoid replay attacks, if it is not the expected one, the logs will appear like this:

```
register | 2025/02/18 17:00:42 Failed to verify the first quote: quote
    Nonce G2bDUiujDREzz5XNXNzDhBZ+8Z71bktmr2SCTjqJ6qk= did not match
    expected Nonce
```

Listing 7.14: Register logs, wrong Nonce

Correct Challenge Creation Data

After successfully verifying the first quote, the agent needs to use another API to continue the registration procedure. It has to send the necessary data to create the challenge used to demonstrate that the AK used to sign the first quote is resident on a TPM with a valid EK. To do this, the agent must send a POST request to the node that executes the register module of the Verifier endpoint `/registration`, which expects a JSON objects structured as follows:

```
{
    "AKPubBlob": "ACMACwAFAHIAAAAQABgACwADABAAIOSss9JAWBTYMC/d...",
    "NameData": "ACIAC/xbpnrmR2I1PeUxypyv3R5KZOQonOye5nadIEGv3ChG",
    "EKTPMPub": "AAEACwADALIAIINxl2dEhLP4GpDMjUal1yT9UtduBlILZPKh2
    hszFGmqAAYAgABDABAIAAAAAAAABALCN2PaQVTTkTj2EVkWUs+Tdmri5fkZhhZA
    uQrOl3LN6HcJP2XQLAB67wzWkliMMNW9IecVrhc9qav89IZZpV015WaoOZtSLA
    vpBjUySaLUZGITOZecydbflgv/+Bkztvkjca/TcqVhvrhSqeq5O3Ew04r0RbUp
    /7Ww4z/x0piVL+gPkZCug71gkCpUf9s2lYHxulqDIqMPCxktEAhNkGOZBjW1uC
    SXFQwuHWAwn/0Bp3WAtkajdm9GVTp9FGP7/Jyyou88mbW4TpOMLE9KMFfchG9h
    NaG0Yaj1BMP4C0bAh11/9uSA574jg1o4Z+w2iOV8X/yxf4wGRyHDye5eOV0="
}
```

Listing 7.15: Example of properly structured object for creating the challenge

At the end of this API, if the data are correct, the register sends back the challenge to be solved by the agent.

Wrong Challenge Creation Data

The wrong values of the “AKPubBlob”, “NameData” and “EKTPMPub” have been considered for these tests.

The case in which AKPubBlob has a wrong value causes the following error message in the logs:

```
register | 2025/02/19 17:00:42 Name was not for AK public blob
```

Listing 7.16: Register logs, wrong AKPubBlob or NameData

In fact, the hash over the AK public blob (which is the NameData) is recalculated and compared against the received NameData. As they are not equal, it means that the AKPubBlob is wrong or that the NameData is properly formatted but has a wrong digest value.

The case in which the NameData is wrongly formatted causes an error during the decoding of its bytes:

```
register | 2025/02/19 17:05:03 Error unpacking name: unexpected EOF
```

Listing 7.17: Register logs, wrong NameData formatting

If then, the EKTPMPub is wrong, the error message in the logs is the following:

```
register | 2025/02/19 17:07:55 Error decoding ek public key: decoding
    TPMT_PUBLIC: EOF
```

Listing 7.18: Register logs, wrong EKTPMPub formatting

In all of these cases, the code that creates the challenge, which relies on the NameData and the EKTPMPub, is not reached and the challenge is not created.

Correct Challenge Solution

If the previous API is properly executed, then the agent has received the challenge. Now, to send the challenge, it has to execute a POST request to the register module running on the Verifier endpoint `challenge` containing a JSON object formatted as written in the following:

```
{
    "Solution": "dv3Rk1FtrzEsCaYX1nTA0pWdQUazkYFFBJYuOCIg7Ds=",
    "WhitelistID": "f13ceb44-5de7-4e25-ac72-975253823048"
}
```

Listing 7.19: Example of a proper Challenge Solution object

It is valuable to notice that the WhitelistID is sent both here and in the First Quote object to properly update the relational database whether the challenge is correctly solved (state "Registered"), or whether the first quote is not properly verified (state "Registration Failure").

Wrong Challenge Solution

Three cases have been considered for these tests. A case in which the solution is wrong, a case in which there is no WhitelistID and a case in which the WhitelistID is wrong.

Here are the corresponding error messages written in the logs:

```
register | 2025/02/19 17:12:49 Wrong challenge solution
```

Listing 7.20: Register logs, wrong Challenge Solution

```
register | 2025/02/19 17:14:10 Null Whitelist ID... insert one
```

Listing 7.21: Register logs, no WhitelistID sent

```
register | 2025/02/19 17:16:47 Received WhitelistID is not in the DB:
    mongo: no documents in result
```

Listing 7.22: Register logs, wrong WhitelistID

7.1.3 Attestation

After the device registration, the procedure for attestation can proceed. The procedure in which the nonce is exchanged is not considered for the tests. To perform the attestation, the agent must interact with the Verifier endpoint. It executes a POST request to the `/verifyattestation` API.

Proper Attestation Object

The POST request must contain a JSON object structured as follows:


```
{
    "DeviceID": "8ad78306-f7c3-4c11-96a8-19889f0749eb",
    "Attestation": {
    "ak_pub": "ACMACwAFAHIAAAAQABgACwADABAAIHC5DVh/B5f7ibZ8hB6HR/
    U0g3Zo61ndbF7C4NMDoa2zACC3qvDMHYoWJizZcYAMFRNhCy9NSY6w8jyDNy
    9IayO/9w==",
    "quotes": [
        {
        "quote": "/1RDR4AYACIAC9voKui0+9Z+kDfUeXaaBsPs0tJ4q+G/DJU
        rS4+Fs8qtACDY9gREJ5DilNboQvnJIvCkoebvU7If4fq7nS4+PsO+tgA
        AAAAAAJOyjGNSQxsJoEEBzyKXp1/qieAAAAABAAQD/wcAACAzED/FJOg
        QWTsxS2YJpgX9uf5RO8VmzYs+zqHUv8+cqQ==",
        "raw_sig": "ABgACwAg7q1WvmquURGtx69hksDO9hqccbxOrkikC68U9
        F5/oH8AIIsizdyZwfPc0VpdcwvDoSHj/S0Z+vGkq+pHg1kLM/oG",
        "pcrs": {
            "hash": 4,
            "pcrs": {
                "0": "PcrqJdyGVU2UuUqlvI9zWkkhKvg=",
                "1": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "10": "aoVxiToW8TQdqF2txPNz6CK4zNM=",
                "2": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "3": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "4": "Q9wQ7XSc8q3vImsKhkRswmcmS2c=",
                "5": "R0QbpDEBK4YPrg9IolH6KPCH1Fc=",
                "6": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "7": "vXnshUFQBHKIhjHaKLggqMBcEfE=",
                "8": "6DcnHoQJr7wawosjtucv7yHmrKM=",
                "9": "au2zvtxG4uzylVHjIDyn6Co7FOA="
                }
            }
        },
        {
        "quote": "/1RDR4AYACIAC9voKui0+9Z+kDfUeXaaBsPs0tJ4q+G/DJUr
        S4+Fs8qtACDY9gREJ5DilNboQvnJIvCkoebvU7If4fq7nS4+PsO+tgAAA
        AAAAJRljGNSQxsJoEEBzyKXp1/qieAAAAABAAsD/wcAACB2BcRVfPieqY
        yttWk65Qoamtg7GS/seaM/b1IzwyeIFA==",
        "raw_sig": "ABgACwAgnztsQx64mRfm8/AQTD0b5Wscie4DOkVfWDOlcI
        dcdyQAIPNHKYcXknyDNcxUKqK4LA6dMvV1jws9vbzdXbhZe/6m",
        "pcrs": {
            "hash": 11,
            "pcrs": {
                "0": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "1": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "10": "CB1xYi//JPJIhmCi+Fd/y+sBJQh0WW1bNdbHOByMYIw=",
                "2": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "3": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "4": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "5": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "6": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "7": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "8": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "9": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="}
            }
        }],
```

```
        "event_log": "AAAAAAgAAADEL+2tJoIAyx0V+XhBw0Tnna4zIBAAAAAe+2
        tUDB1VQKStTvS/F7g6BwAAAAEAAIAvIBEqP1U5iyCODEJoE4m0y1sYIzQAA
        ABh3+SLypPSEaoNAOCYAyuMCgAAAAAAAAAAAAAAAAAAAFMAZQBjAHUAcgBl
        AEIAbwBvAHQABwAAAAEAAICbE4cwbrt/+OeV5753VjZmu/RRbiQAAABh3+S
        ...",
        "TeeAttestation": null
    },
    "Ima": [
        "10 6bdad[...]5dcdd ima-ng sha256 7b643[...]07d61 boot_aggregate",
        "10 0b5a1[...]1e1ac ima-ng sha256 d9d17[...]b3e46 /init",
        "10 fbc98[...]28c6f ima-ng sha256 f5f25[...]36422 /usr/bin/sh",
        "10 a4c34[...]67e79 ima-ng sha256 595bc[...]72d89
         /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2",
        "10 a1c2f[...]10189 ima-ng sha256 bc375[...]967c7 /etc/ld.so.cache",
        "10 226af[...]87113 ima-ng sha256 5955d[...]15fa7
         /usr/lib/x86_64-linux-gnu/libc.so.6",
        "10 0f8b2[...]9e9a6 ima-ng sha256 91f24[...]0d9fe /conf/arch.conf",
        "10 4ca27[...]0e510 ima-ng sha256 5235c[...]592c2
         /conf/initramfs.conf",
        "10 010d8[...]6339c ima-ng sha256 1b575[...]71a94 /scripts/functions",
        "10 9debe[...]7a04f ima-ng sha256 83464[...]74942
         /scripts/init-top/ORDER",
        ...
    ]
}
```

Listing 7.23: Example of properly structured object for creating the challenge

Wrong Attestation Object

Wrong values of "DeviceID", "ak_pub", "quote", "event_log" and of one measurement in "Ima" are considered for this test. In the following are shown the error messages written in the logs:

```
verifier | 2025/02/22 9:56:58 Error retrieving AK from the DB: no AK for
    the inserted device ID: sql: no rows in result set
```

Listing 7.24: Verifier logs, wrong DeviceID

At first, the verifier retrieves the stored AK from the database to check if it is equal to the received one.

Then if the DeviceID is correct but the AK is different, the error message is:

```
verifier | 2025/02/22 10:12:22 Error: stored AK is different
```

Listing 7.25: Verifier logs, wrong ak_pub

For the wrong quote:

```
verifier | 2025/02/22 10:15:23 Error: failed to verify the attestation:
    failed to verify quote: signature decoding failed
```

Listing 7.26: Verifier logs, wrong quote

If the event_log is wrong, then:

```
verifier | 2025/02/22 10:23:57 Error: failed to verify the attestation:
    failed to validate the PCClient event log: (PANIC=Error method:
    unimplemented)
```

Listing 7.27: Verifier logs, wrong quote

In the end, if one Ima measurement is wrong, the logs print the file with the wrong hash:

```
verifier | 2025/02/22 10:32:21 Error: failed to validate the integrity of
    ima: boot_aggregate
```

Listing 7.28: Verifier logs, wrong Ima measurement

7.2 Performance Tests

Performance Tests have been performed in order to evaluate the time needed for the Registration and Attestation procedures. In these times, it has been discriminated against the different overheads that occur during the procedures. In particular, the following:

- for the Agent, the overhead caused by the TPM interactions;
- for the Register and the Verifier, the overhead caused by the databases interactions.

For this reason, two subsections are presented considering the two sides of the procedures. In both cases a sample of twenty tests have been considered.

7.2.1 Agent

In this subsection only the Agent performance is discussed, considering the different ways in which the procedure can be started:

- Agent starts Registration, Agent starts Attestation;
- Agent starts Registration, Verifier starts Attestation;
- Register starts Registration, Agent starts Attestation;
- Register starts Registration, Verifier starts Attestation.

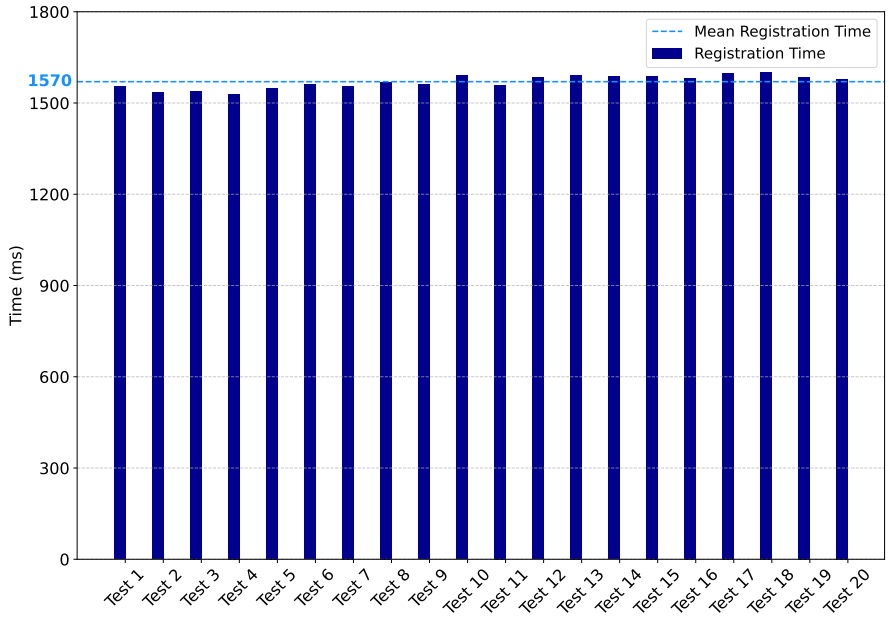


Figure 7.1: Registration Times when Agent Starts both Procedures

During the Registration, the Agent interacts with the TPM four times: it retrieves the EK certificate, creates the AK, creates the Quote, solves the challenge and stores the Device ID.

While during the Attestation phase, the Agent overheads are the TPM interaction to create the attestation object and the reading of the IMA file from the kernel of the operating system.

Started by Agent, Continued by Agent

This is the case in which the Agent starts the procedure and so the Registration phase, and then it also starts the Attestation.

In Figure 7.1, it is possible to see the times needed to complete a registration. The average time is 1.570 seconds.

On the other hand, Figure 7.2 shows the times needed to perform an Attestation. The average time is 411 milliseconds.

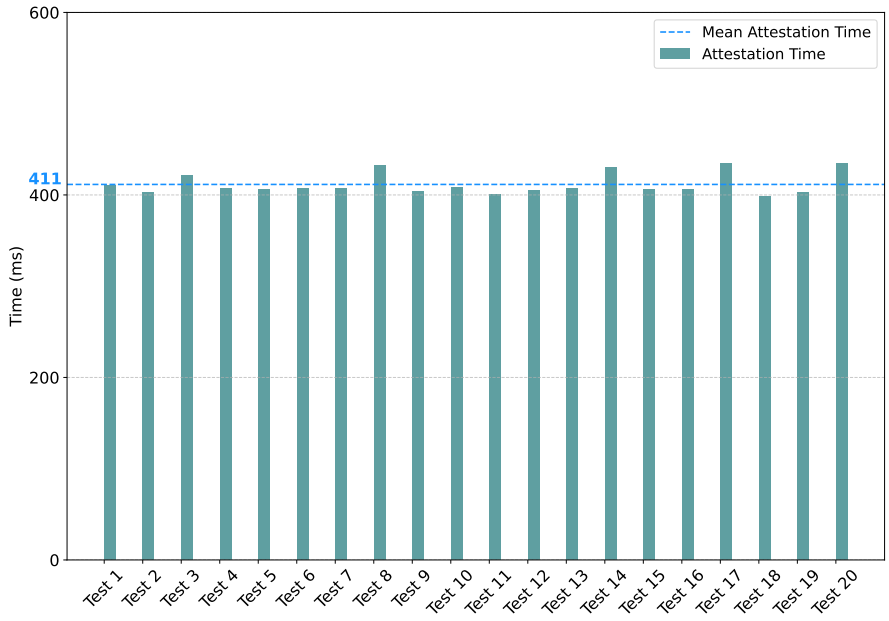
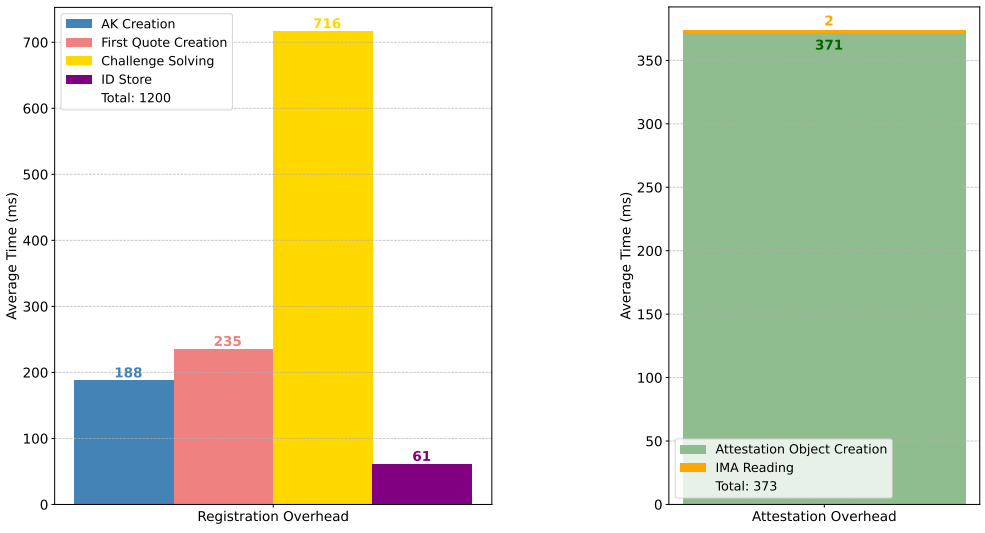


Figure 7.2: Attestation Times when Agent Starts both Procedures



(a) TPM Interactions during Registration

(b) TPM Interactions during Attestation

Figure 7.3: Agent Average TPM Overheads in Registration and Attestation Procedures

The average overhead times are reported in Figure 7.3. In particular, in Figure 7.3a, it is possible to notice that the time of EK retrieval has not been considered because it has been discovered to be some nanoseconds, which is negligible. The same consideration is applied for the further cases.

So, considering the TPM interactions, it is also possible to discriminate the average time needed for network communication during both procedures. This is shown in Figure 7.4.

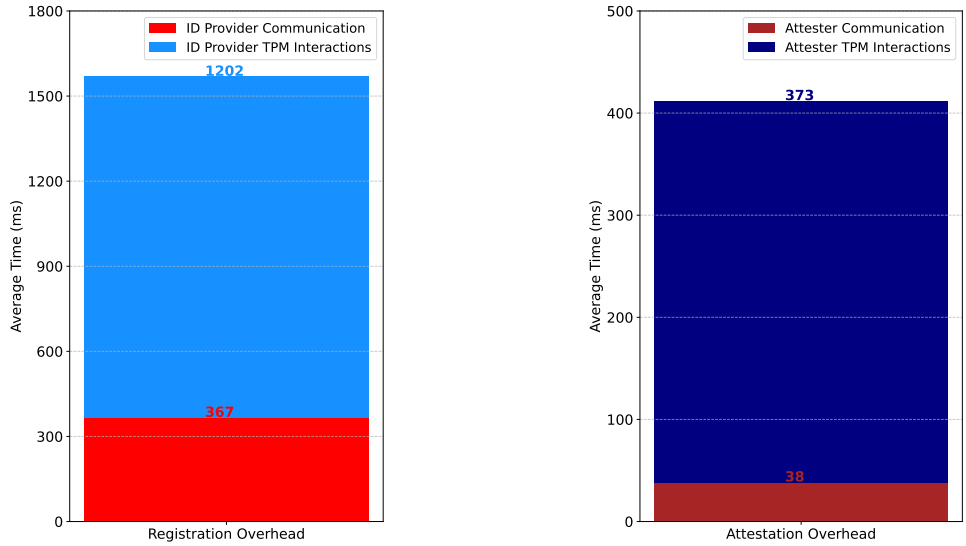


Figure 7.4: Agent Average Overheads Comparison in Registration and Attestation Procedures

In the end, most of the time for both procedures is spent interacting with the TPM. In fact, considering the average Registration time of 1570 milliseconds, 1202 of it are exploited in TPM interactions. While considering the average Attestation time of 411 milliseconds, 373 of it are needed to interact with the TPM.

Started by Agent, Continued by Verifier

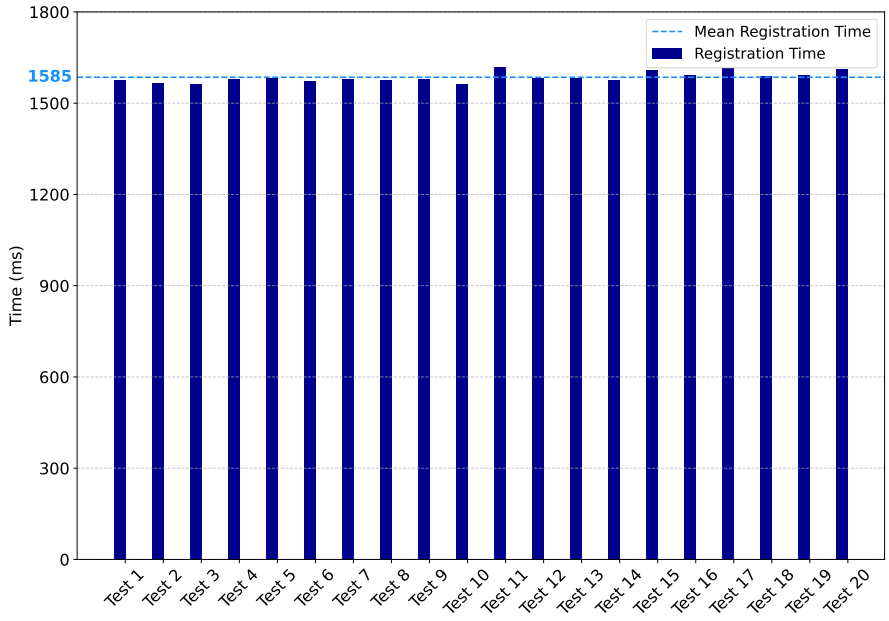


Figure 7.5: Registration Times when Agent Starts Registration Procedure only

In these tests, only the Registration is initiated by the Agent. Figure 7.5 shows that in this case, the average Registration time is 1.585 seconds.

The Attestation phase is carried out by the Verifier and in Figure 7.6 it is possible to notice that it requires 409 milliseconds on average.

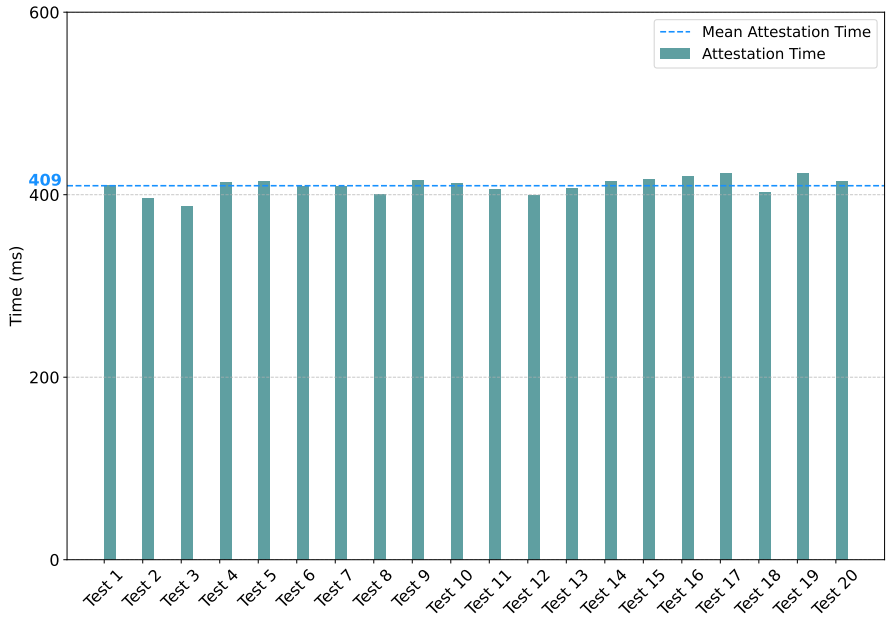
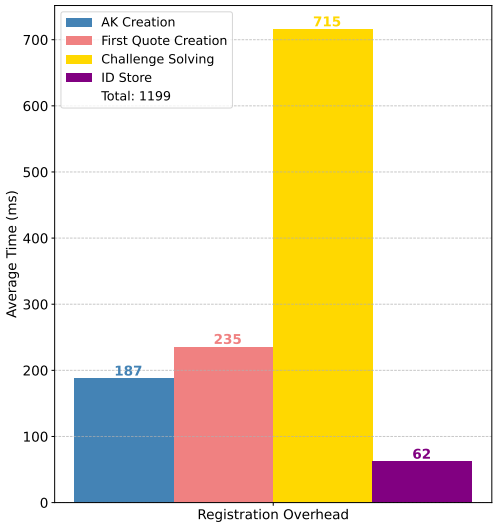
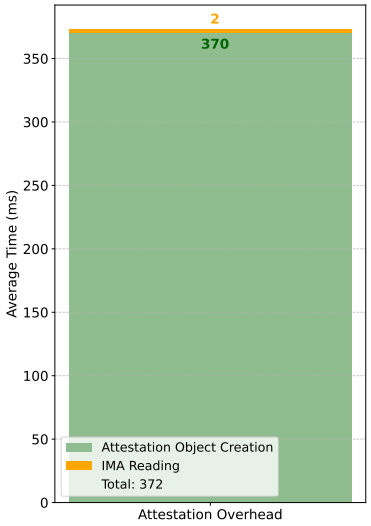


Figure 7.6: Attestation Times when Agent Starts Registration Procedure only

The overheads caused by the interactions with the TPM are shown in Figure 7.7.



(a) TPM Interactions during Registration



(b) TPM Interactions during Attestation

Figure 7.7: Agent Average TPM Overheads in Registration and Attestation Procedures

Also in this case, considering the TPM interactions, it is possible to discriminate the time required by the network communication, which is shown in Figure 7.8.

Overall, in relation to the previous case, these tests have been affected by network communication overhead as it takes 383 milliseconds on average. The Registration and Attestation phases require the same time needed for the previous case.

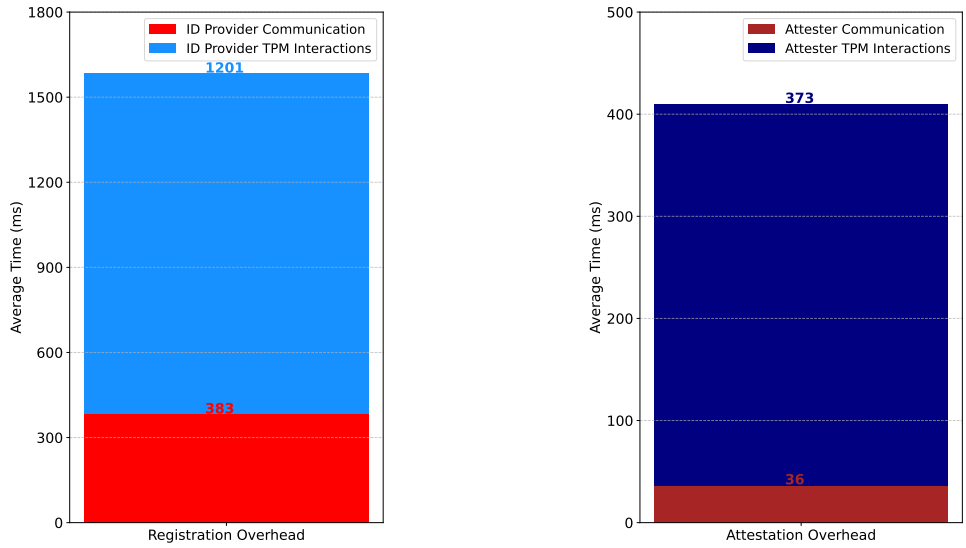


Figure 7.8: Agent Average Overheads Comparison in Registration and Attestation Procedures

Started by Register, Continued by Agent

In the tests performed when the Registration procedure is initiated by the Register, it is possible to notice the effect given by one missing message during the communication.

The total procedure is started by the straight sending of the Nonce by the Register and the Request Probe is never sent by the Agent during the Registration phase.

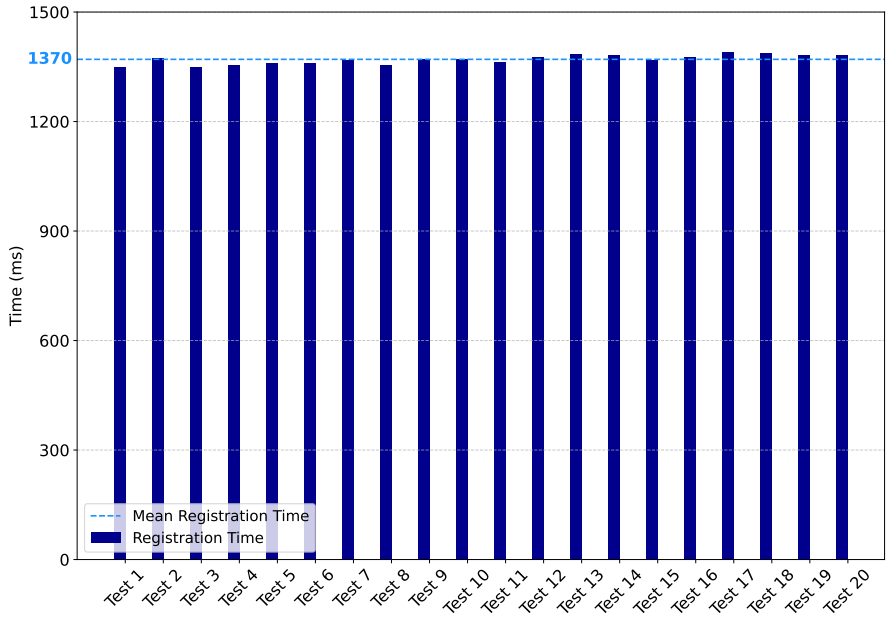


Figure 7.9: Registration Times when Agent Starts Attestation Procedure only

As Figure 7.9 shows, this fact causes a loss of 200 milliseconds on average in relation to the previous cases as the average Registration procedure takes only 1.370 seconds.

The Attestation is then carried out by the Agent that sends its Device ID to the Verifier. With respect to the previous cases, 10 milliseconds on average are added, thus increasing the average Attestation time to 426 milliseconds as shown in Figure 7.10.

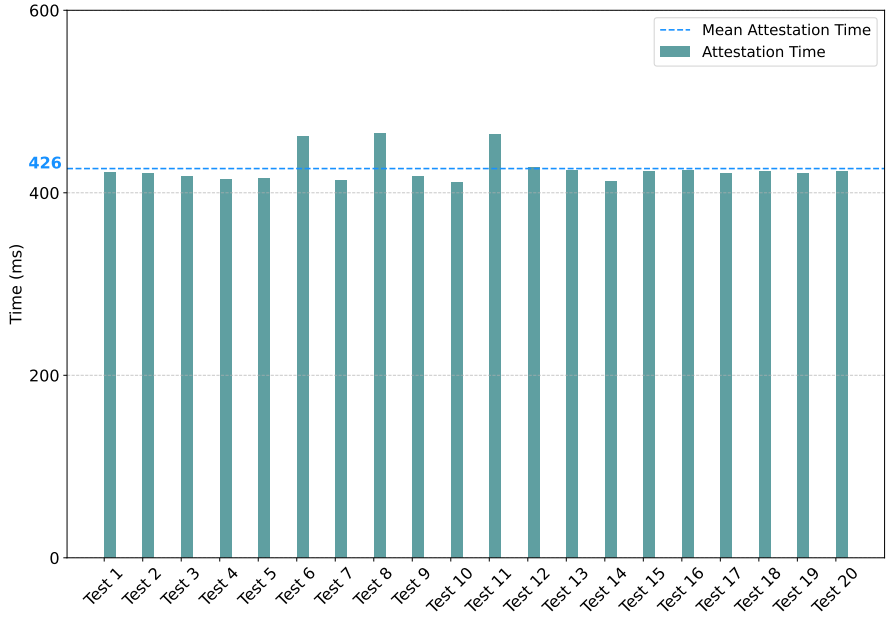
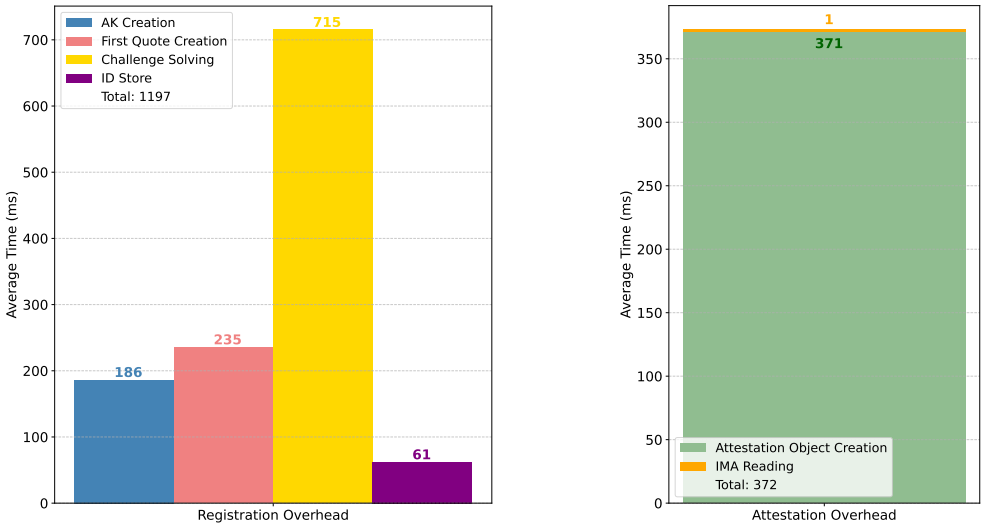


Figure 7.10: Attestation Times when Agent Starts Attestation Procedure only

The average TPM interactions and network communication overheads are shown in Figure 7.11 and Figure 7.12 respectively.



(a) TPM Interactions during Registration

(b) TPM Interactions during Attestation

Figure 7.11: Agent Average TPM Overheads in Registration and Attestation Procedures

As it is possible to notice in Figure 7.12, also in this case most of the time is spent interacting with the TPM as it takes 1199 milliseconds on average during the Registration phase and 373 milliseconds on average during the Attestation one.

The main cause of the addition of 10 milliseconds on average in the Attestation phase can be traced back to network communication, as well as the loss of 200 milliseconds on average concerning the Registration.

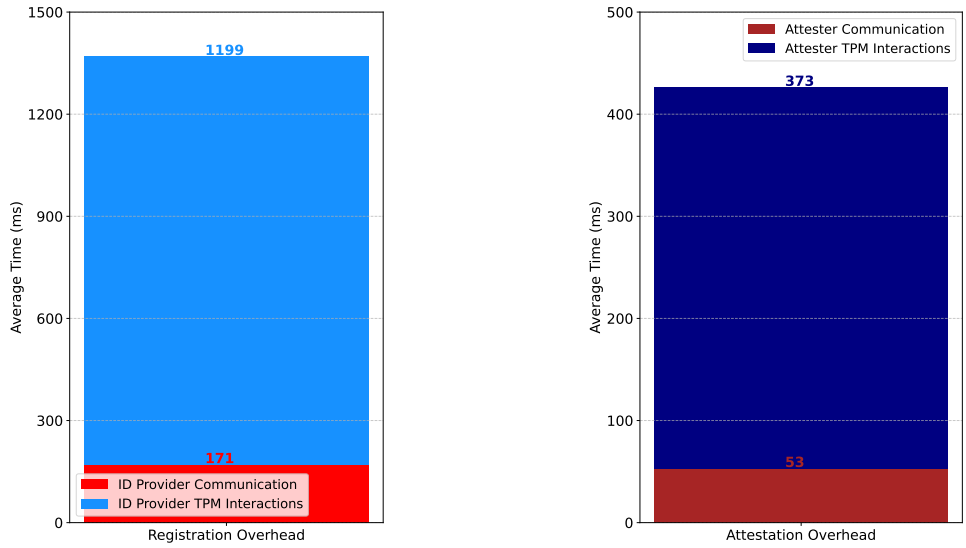


Figure 7.12: Agent Average Overheads Comparison in Registration and Attestation Procedures

Started by Register, Continued by Verifier

In this case, there is an improvement in both the average Registation and Attestation times with respect to the first two cases. During the former, no Request Probe is sent by the Agent, during the latter neither the Device ID is sent by the Agent.

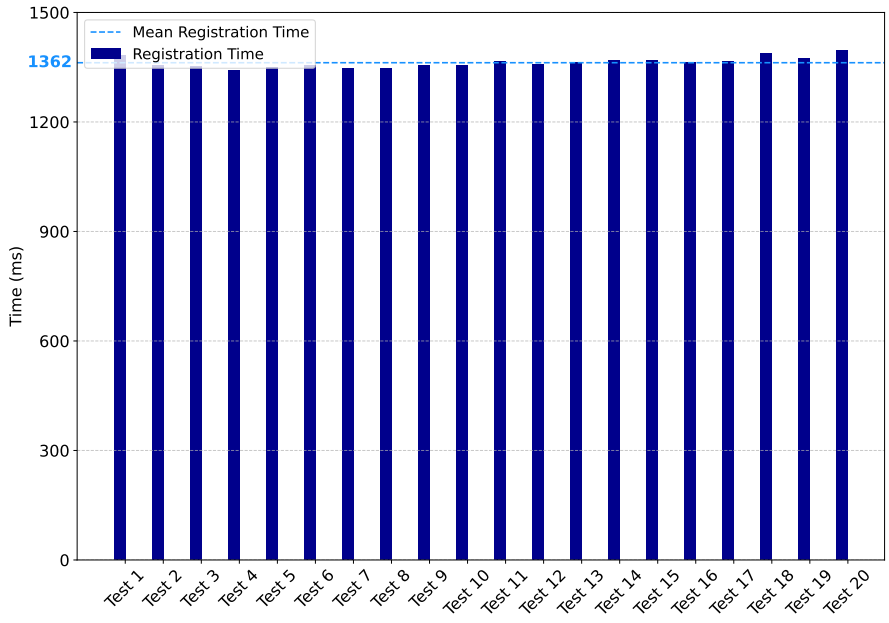


Figure 7.13: Registration Times when Agent does not start any Procedure

These two missing messages during the communication decrease the average Registration time to 1.362 seconds (Figure 7.13) and the average Attestation time to 410 milliseconds (Figure 7.14).

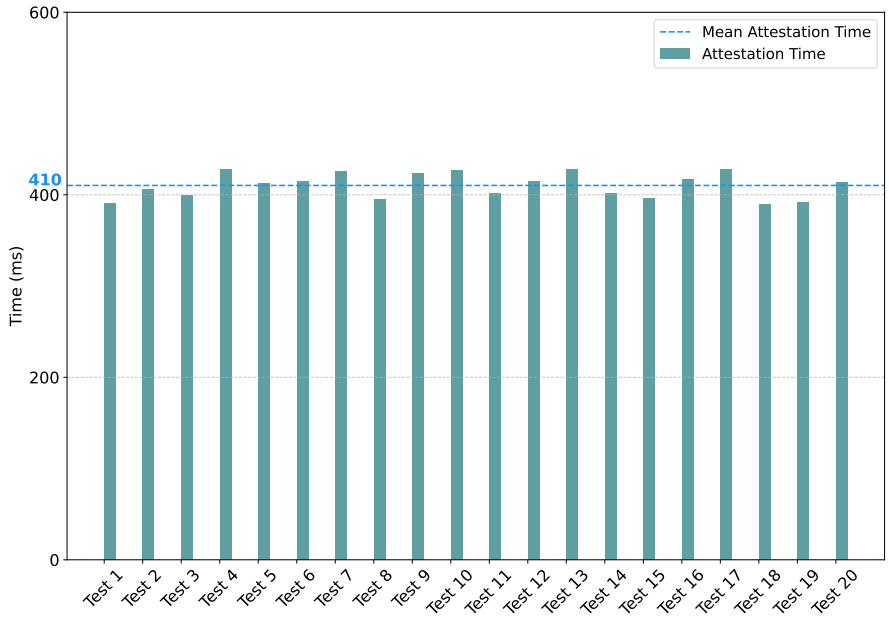
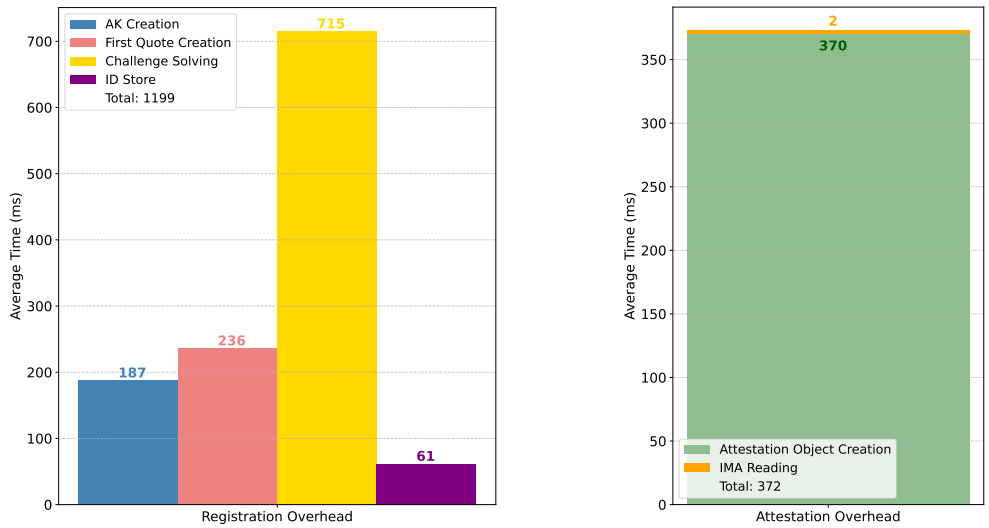


Figure 7.14: Attestation Times when Agent does not start any Procedure

The average overheads given by the TPM interactions are depicted in Figure 7.15 and are very similar to the overheads of the previous cases.



(a) TPM Interactions during Registration

(b) TPM Interactions during Attestation

Figure 7.15: Agent Average TPM Overheads in Registration and Attestation Procedures

In the end, the network communication overhead is considered as well as shown in Figure 7.16. Overall, the tests have highlighted that this is the case with the best average performance for the Agent and the less overhead caused by the network communication.

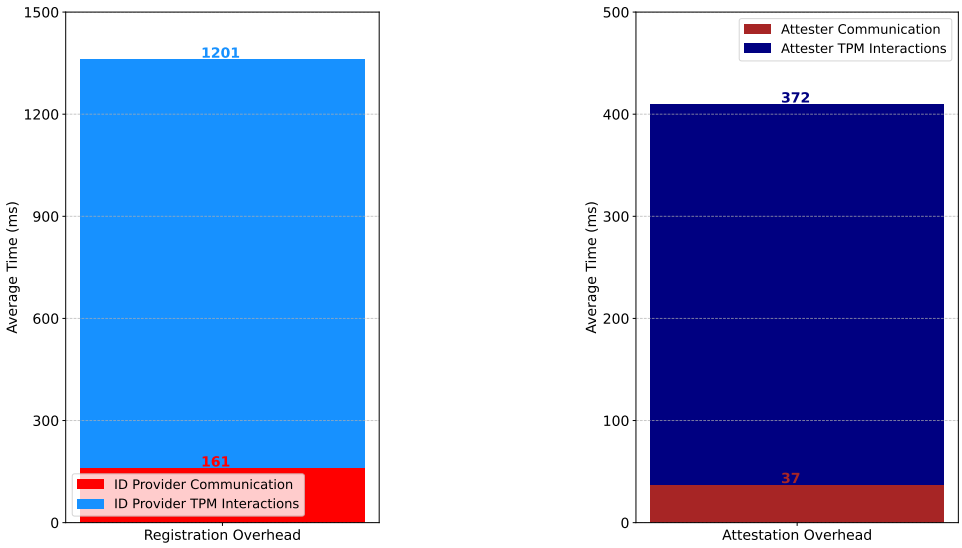


Figure 7.16: Agent Average Overheads Comparison in Registration and Attestation Procedures

7.2.2 Register and Verifier

In this subsection the Register and Verifier performances are discussed, considering the different ways in which the procedure can be started:

- Agent starts Registration, Agent starts Attestation;
- Agent starts Registration, Verifier starts Attestation;
- Register starts Registration, Agent starts Attestation;
- Register starts Registration, Verifier starts Attestation.

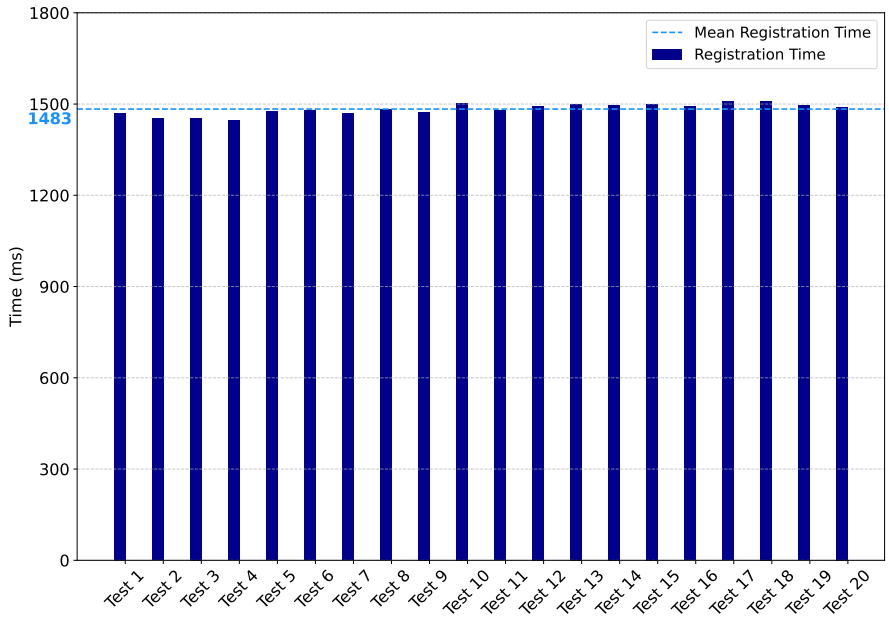


Figure 7.17: Registration Times when Agent starts both Procedures

The main operation carried out by the Register and the Verifier is the one of verification of the data sent by the Agent. The overheads are caused by the time required to access the databases and the network communication.

For the Register, the Registration phase ends when it sends the Device, so before with respect to the Agent where the procedure ends when the Device ID is received and stored within the TPM. On the other hand, for the Agent the Attestation procedure ends when the attestation object is sent, while for the Verifier the procedure ends when it verifies the attestation object and produces a result.

Started by Agent, Continued by Agent

The different points in time in which the two procedures end, can be immediately noticed in Figure 7.17 where the average Registration time is 1.483 seconds while the Agent takes almost 100 milliseconds more in the same case.

On the contrary, the average Attestation time is 478 milliseconds (Figure 7.18) which is more than 50 milliseconds with respect to the Agent considering the same case. This is due to the verification of the attestation object.

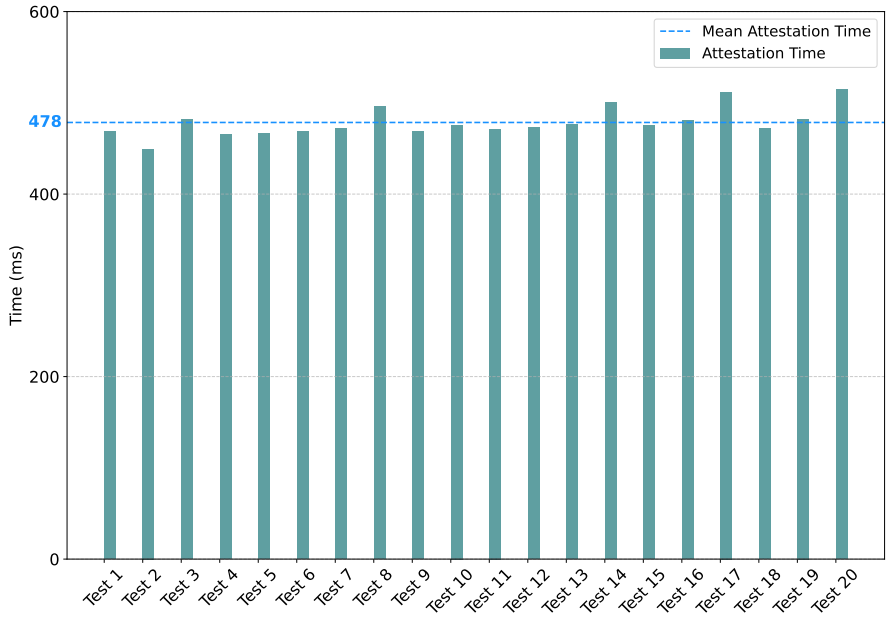


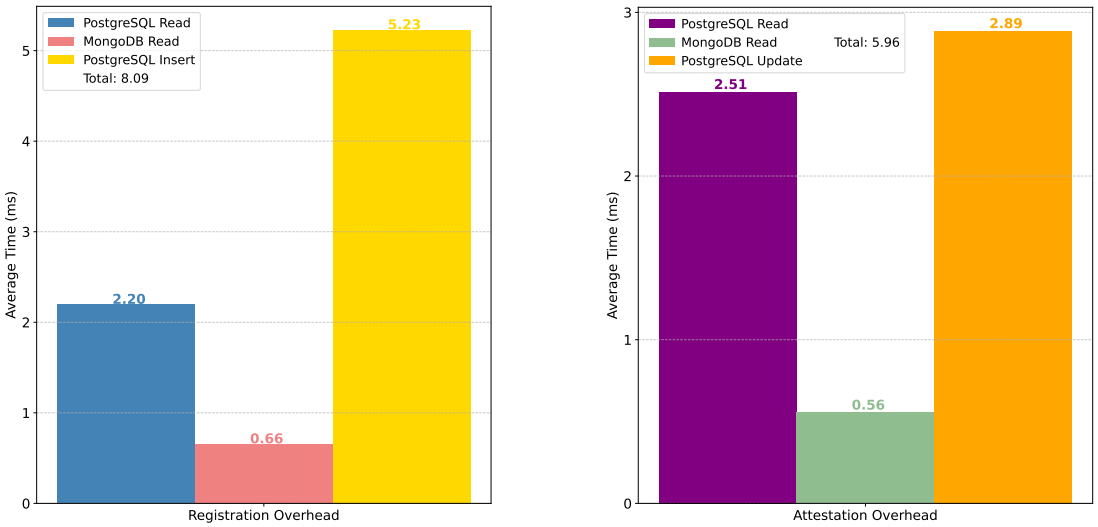
Figure 7.18: Attestation Times when Agent starts both Procedures

The Register accesses one time for a reading operation to the Non-Relational database and twice to the Relational database, once for a reading operation and once for an insert operation.

Instead, during the Attestation phase, the Register accesses once to the Relational database to update the state of the Agent, and the Verifier accesses once to the Relational database to retrieve the ID of the reference values and once to the Non-Relational database to read the reference values.

The average databases interactions times are reported in Figure 7.19, where PostgreSQL is the Relational database and MongoDB is the Non-Relational database.

It can be noticed how the time required to access the databases, in general, is very low, and in particular also how the interactions with the Non-Relational database are faster than the Relational database one, as expected.



(a) Databases interactions during Registration

(b) Databases interactions during Attestation

Figure 7.19: Average Register and Verifier Databases Interactions Overheads in Registration and Attestation Procedures

Considering that the average Registration takes 1483 milliseconds and the average Attestation takes 479 milliseconds, while the overhead caused by the interactions with the databases is 8.09 milliseconds in total and 5.96 milliseconds in total, respectively for the two phases, the communication and verification time compared to the databases interactions overhead is not reported as basically all the time is spent for these purposes, and a graph has been considered not worthy. The same consideration is applied in the following cases.

Started by Agent, Continued by Verifier

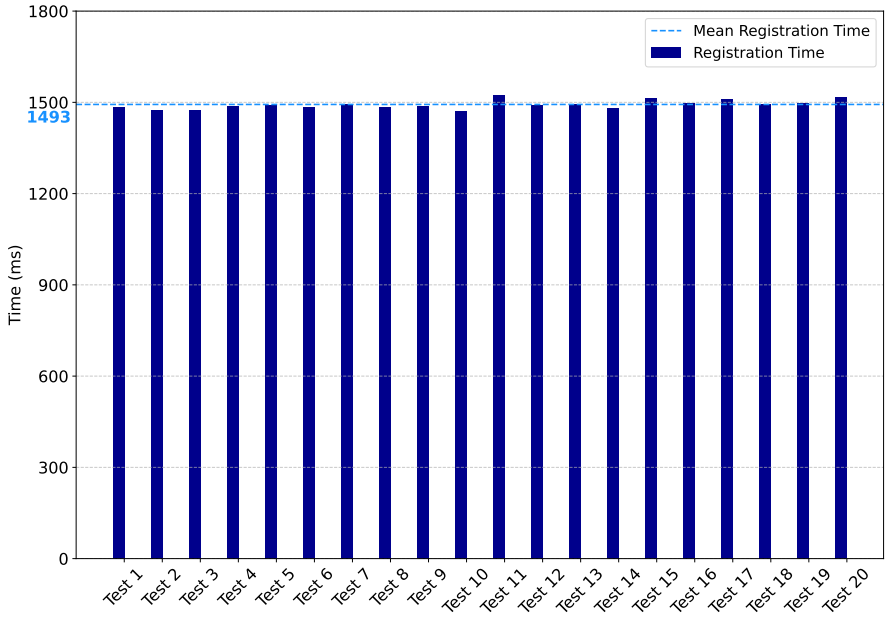


Figure 7.20: Registration Times when Agent starts Registration Procedure only

The Registration phase takes a very similar time to the previous case, this can be noticed in Figure 7.21 where the average time is 1493 milliseconds.

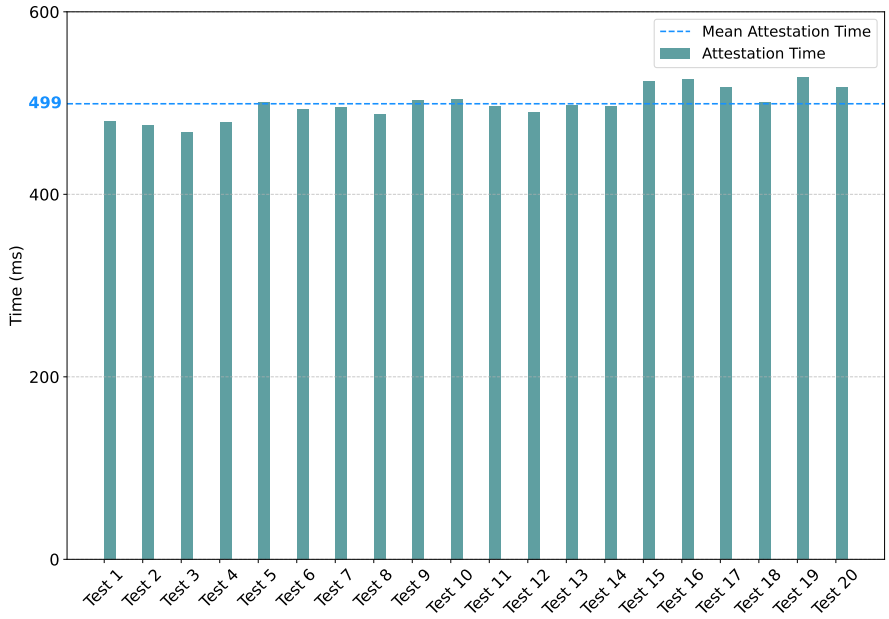
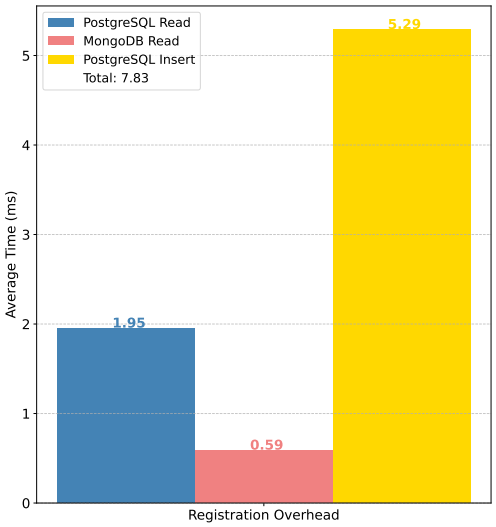
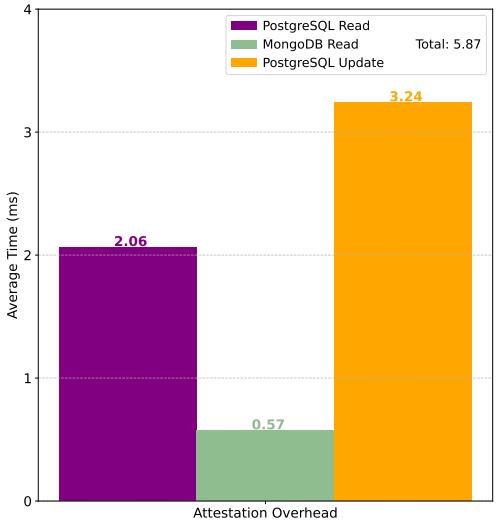


Figure 7.21: Attestation Times when Agent starts Registration Procedure only

The Attestation phase, on the other hand, is initiated by the Verifier, so it sends a further message, with respect to the previous case. This is an `HTTP_STATUS_OK` to the Agent, needed to make him available again and perform the next Attestation round. The effect of this, can be seen in Figure 7.21 where the mean Attestation time is 499 milliseconds, while in the previous case it is 478 milliseconds.



(a) Databases interactions during Registration



(b) Databases interactions during Attestation

Figure 7.22: Average Register and Verifier Databases Interactions Overheads in Registration and Attestation Procedures

The average overheads are basically the same as the previous case with a total of 7.83 milliseconds during Registration, and 5.87 in total in the Attestation phase.

Started by Register, Continued by Agent

Figure 7.23: Registration Times when Agent starts Attestation Procedure only

In this case, the procedure is initiated by the Register, so the Request Probe is not sent by the Agent. The effect of the missing message can be seen in Figure 7.23 where the mean Registration time is 1087 milliseconds, while in the previous cases it is at least 400 milliseconds longer.

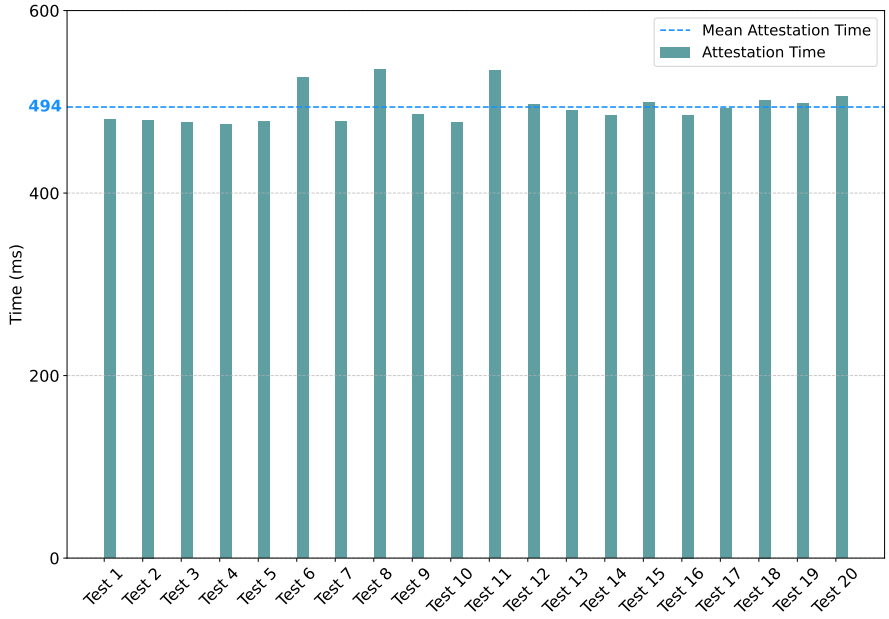
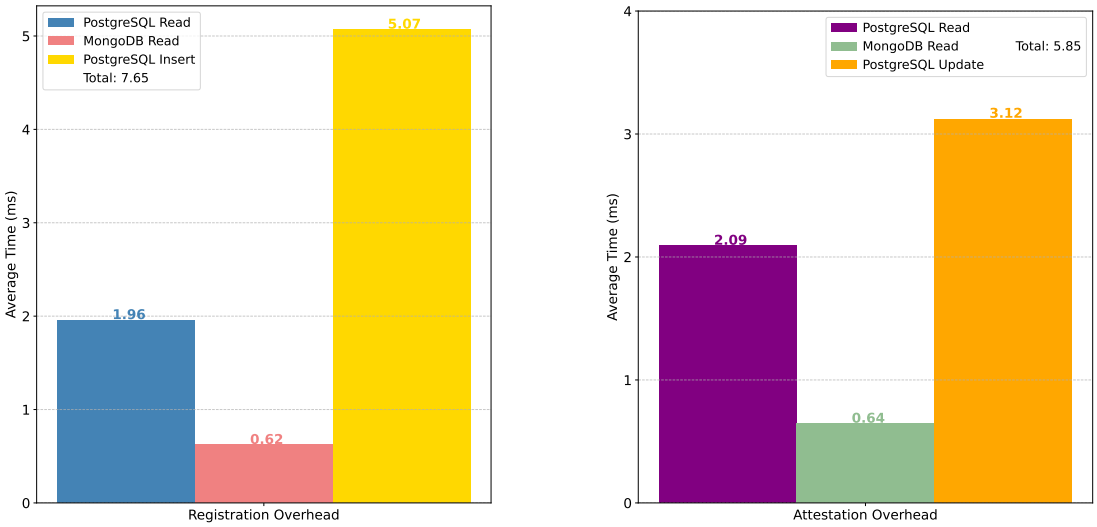


Figure 7.24: Attestation Times when Agent starts Attestation Procedure only

The Attestation phase is similar to the previous case where the Verifier starts the procedure as well. The average Attestation time is 494 milliseconds and is shown in Figure 7.24.

The average overheads are shown in Figure 7.25, also, in this case, they are similar to the previous cases and negligible.



(a) Databases interactions during Registration

(b) Databases interactions during Attestation

Figure 7.25: Average Register and Verifier Databases Interactions Overheads in Registration and Attestation Procedures

Started by Register, Continued by Verifier

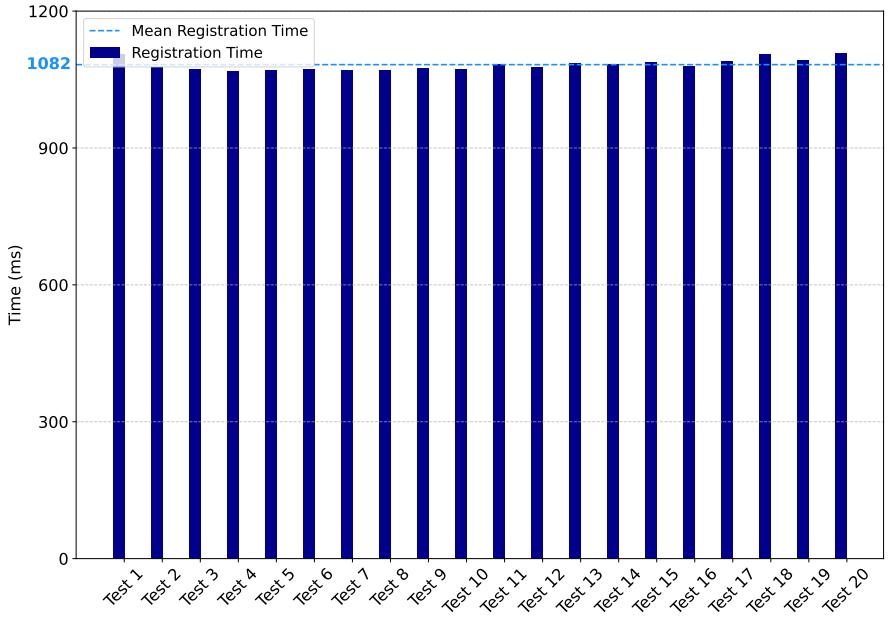


Figure 7.26: Registration Times when Agent does not Start any Procedure

As expected, considering that also in this case the procedure is started by the Register, the mean Registration time is 1082 milliseconds and similar to the previous case (Figure 7.26).

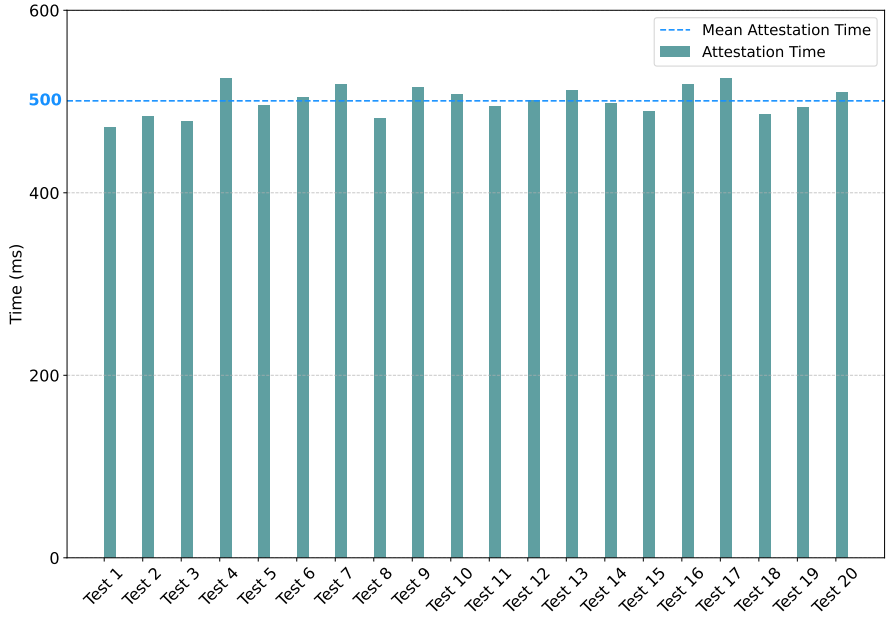
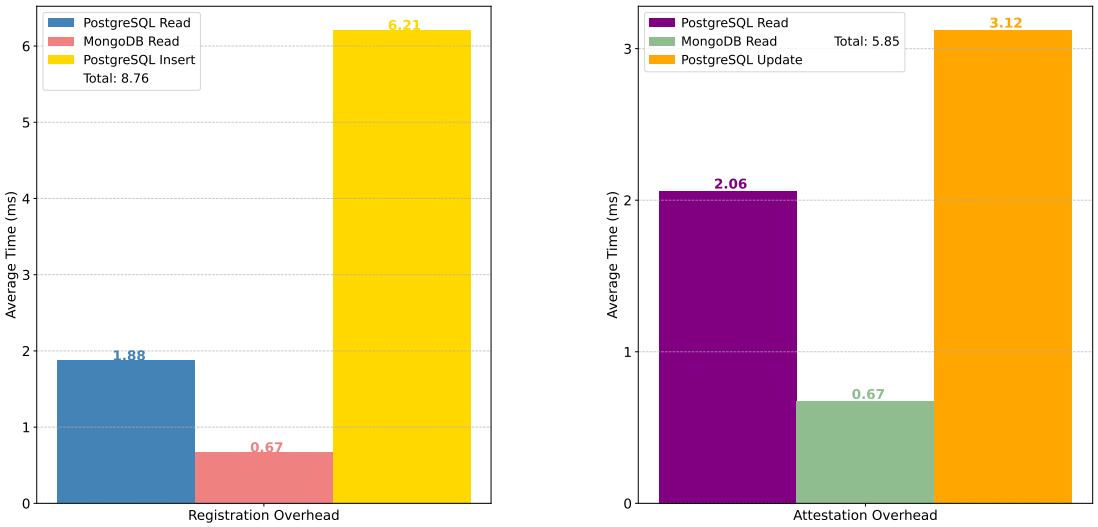


Figure 7.27: Attestation Times when Agent does not Start any Procedure

In this case, as previously stated (sub-section 7.2.2), there is an additional message in the Attestation procedure, thus increasing the mean time to 500 milliseconds, as Figure 7.27 shows.

The average overhead times are shown in Figure 7.28 and there are no differences with respect to the previous cases.



(a) Databases interactions during Registration

(b) Databases interactions during Attestation

Figure 7.28: Average Register and Verifier Databases Interactions Overheads in Registration and Attestation Procedures

7.2.3 Considerations

In the previous sub-sections details about performance tests in each configuration have been given. In this sub-section the aim is to sum up and give some final considerations about the performances.

The time to perform the Registration and the Attestation has been taken, and the TPM interactions cause some non-negligible overhead. Figure 7.29 shows the average Registration time

with the overheads coming from TPM interactions and network communication. It is possible to distinguish the case in which the procedure is started by the Agent, and the case in which it is started by the Register. In particular, the former is executed within 1576 milliseconds on average, and the latter in 1366 milliseconds on average.

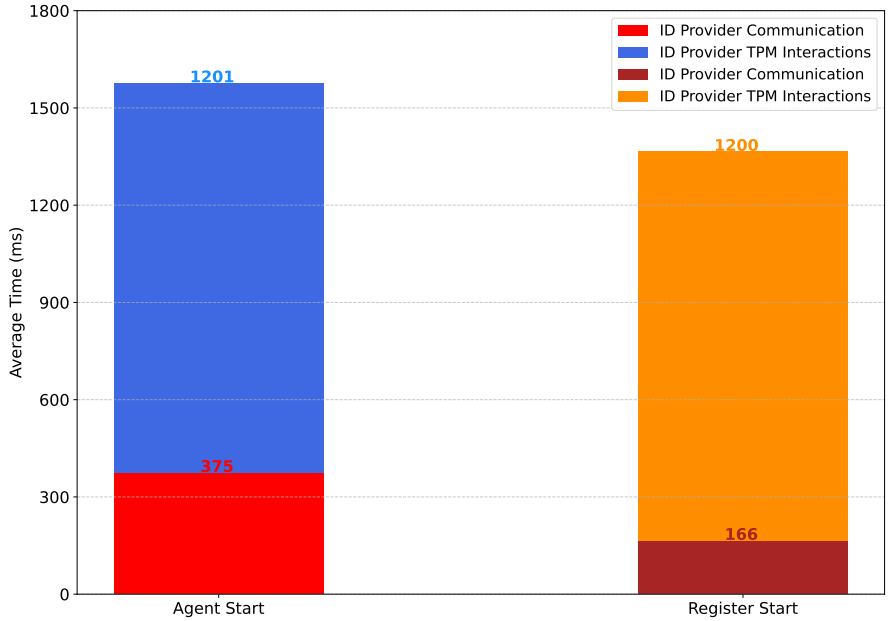


Figure 7.29: Average Agent's Registration Time and Overheads distinguished on who Starts the Procedure

The same has been done for the Attestation phase, and when it is started by the Agent, it takes 418 milliseconds on average, when it is started by the Verifier, it takes 409 milliseconds.

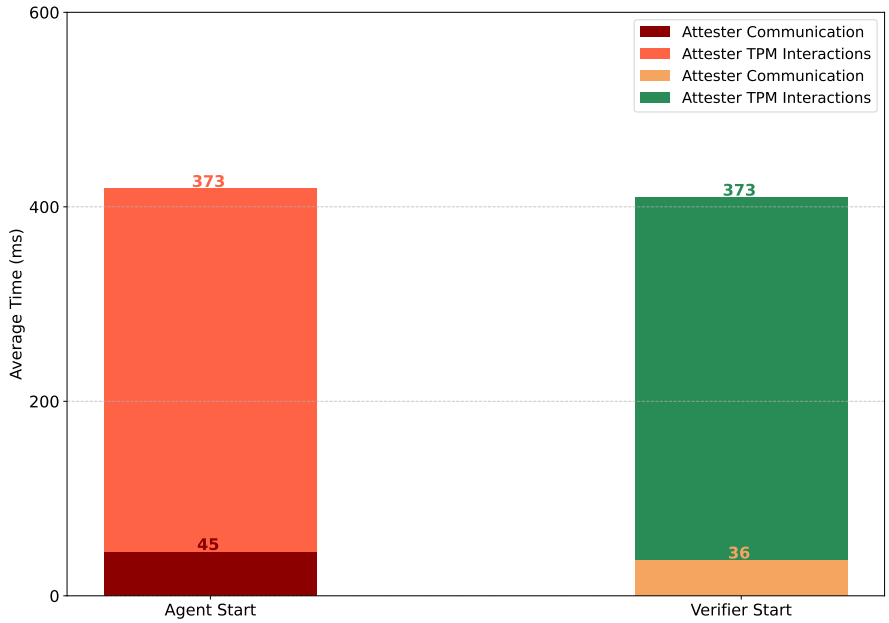


Figure 7.30: Average Agent's Attestation Time and Overheads distinguished on who Starts the Procedure

The fact that when the procedure is started by the Register or the Verifier takes less time, is given mainly by the fact that it is sent one less message in the communication. In fact, during

the Registration, the Agent does not send the Request Probe, and during the Attestation, it does not send its Device ID, but in both cases it is directly contacted by the Register and the Verifier, respectively.

To sum up, in Figure 7.31 is shown the average Registration and Attestation times without distinctions. So, the average Registration and Attestation times considering all the ways in which the two procedures can be started are 1472 milliseconds and 414 milliseconds respectively.

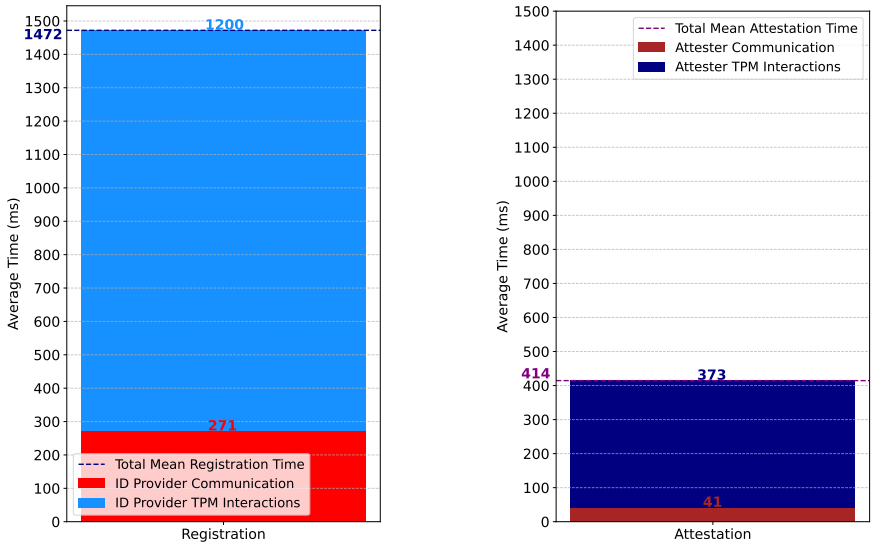


Figure 7.31: Average Agent's Registration and Attestation Times with Overheads

For the Register and Verifier sides, the Registration and Attestation times have also been taken. But, in this case, the time to access the databases which has been considered has overhead is negligible as it is of the order of some milliseconds. Figure 7.32 shows average times for the cases in which the Registration is initiated by the Agent and by the Register.

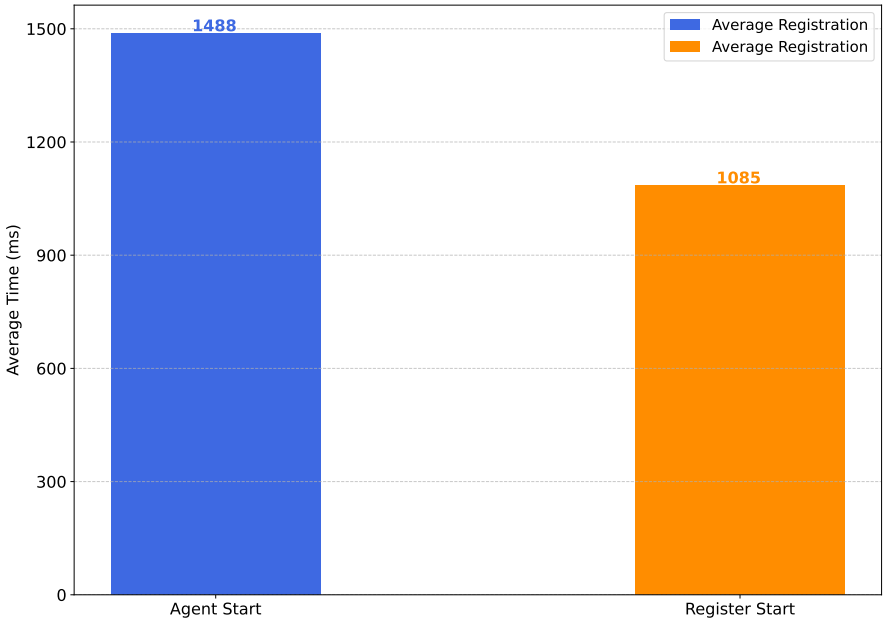


Figure 7.32: Average Register's Registration Time and Overheads distinguished on who Starts the Procedure

As for the Agent case, also the graph (Figure 7.32) highlights that the case in which the procedure is carried out by the Register takes less time. It takes even less time than the Agent as it receives the Device ID and stores it inside the TPM, and this operation takes time. While the Register only sends the Device ID and finishes the procedure.

Figure 7.33 then, shows the average Attestation times as for the Registration ones. It can be noticed how the Attestation takes additional time with respect to the Agent as it only sends the attestation object and finishes. While the Verifier performs the verification of the attestation object after receiving it, and this operation takes time.

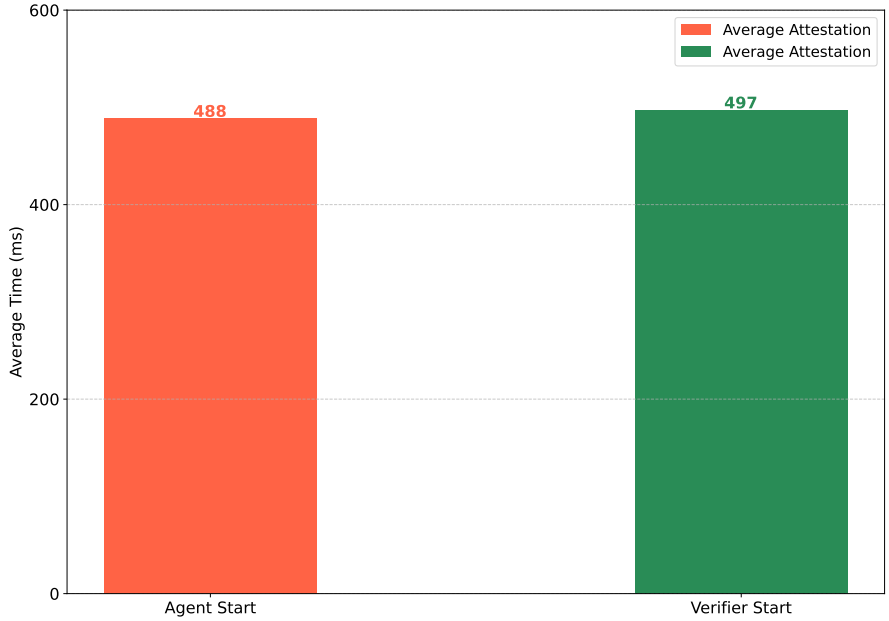


Figure 7.33: Average Verifier's Attestation Time and Overheads distinguished on who Starts the Procedure

In the end, Figure 7.34 show the average Register's Registration time and the average Verifier's Attestation time considering all the ways in which is possible to start the procedures: 1286 milliseconds and 493 milliseconds respectively.

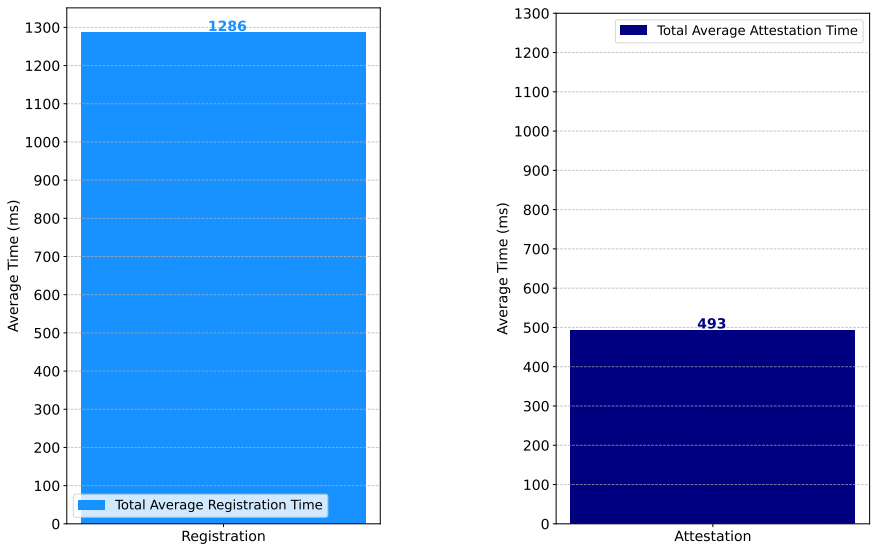


Figure 7.34: Average Register's Registration and Verifier's Attestation Times

Chapter 8

Conclusions

This thesis has explored the challenges and advancements in *Remote Attestation* (RA), particularly in the context of *Trusted Computing* and standard-based approaches. The growing complexity and ubiquity of interconnected devices, especially in cloud computing and IoT environments, have made security and trust crucial concerns. By leveraging the principles of Trusted Computing, and more specifically the *Trusted Platform Module* (TPM) and *Remote ATtestation procedureS* (RATS), this work aimed to contribute to the development of a robust, scalable, and standardized attestation framework.

The work began by introducing trusted computing and its fundamental components, particularly the Trusted Platform Module (TPM), which serves as a cornerstone for ensuring system integrity. The thesis examined the remote attestation process, including its logical workflow and critical aspects such as freshness, attestation results, and verification models. The study further explored existing attestation frameworks, including Keylime and Veraison, evaluating their strengths and areas for improvement.

A core contribution of this work was the design and implementation of a standard-based remote attestation framework. The design phase outlined a structured approach for device registration and attestation, ensuring that each component adhered to the latest standards. The implementation phase demonstrated how the proposed framework successfully aligned with RFC-9334, incorporating modular and scalable design choices. Extensive testing and validation confirmed the effectiveness of the system, with functional tests ensuring correct behaviour and performance tests evaluating efficiency under real-world conditions.

The objectives outlined in the introduction have been successfully achieved. The research has developed an attestation mechanism compliant with RFC-9334, addressing existing gaps in security and trust for cloud and IoT environments. Additionally, the modular nature of the framework allows for flexibility in adapting to various use cases while maintaining interoperability with existing attestation solutions.

Despite these accomplishments, there remains room for future work. As security standards evolve, continuous alignment with future updates to RFC-9334 and other emerging standards will be necessary. Further enhancements could include integrating additional components of Veraison to extend the framework's capabilities, improving interoperability with various hardware and software environments. Additionally, exploring new attestation models and optimizations could enhance performance and scalability, making the solution even more robust for large-scale deployments. Another promising direction is the integration of the proposed solution into other attestation frameworks, allowing for broader applicability and strengthening the overall security of interconnected systems.

In conclusion, this thesis has contributed to the advancement of remote attestation by implementing a secure and standard-compliant framework, addressing critical security needs in modern digital ecosystems. The results provide a strong foundation for further research and development in the field of trusted computing, reinforcing the security of interconnected devices in an ever-evolving technological landscape.

Bibliography

[1] Trusted Computing Group (TCG), https://trustedcomputinggroup.org/about/

[2] Trusted Computing Group (TCG), https://trustedcomputinggroup.org/trusted-computing/

[3] Trusted Computing Group (TCG), "Trusted Platform Module Library Part 1: Architecture", November 29, 2023, https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Library-Family-2.0-Level-00-Revision-1.81-Part-1-Architecture_8December2023-Public-Review.pdf

[4] W. Arthur, D. Challener, and K. Goldman, "A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security", Apress, 2015, ISBN: 978-1-4302-6584-9

[5] Trusted Computing Group (TCG), "TCG PC Client Platform Firmware Profile Specification", June 3, 2019, https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientSpecPlat_TPM_2p0_1p04_pub.pdf

[6] Trusted Computing Group (TCG), "TPM 2.0 Keys for Device Identity and Attestation", October 8, 2021, https://trustedcomputinggroup.org/wp-content/uploads/TPM-2p0-Keys-for-Device-Identity-and-Attestation_v1_r12_pub10082021.pdf

[7] IEEE, "802.1AR-2018 - IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity", June 14, 2018, https://standards.ieee.org/standard/802_1AR-2018.html

[8] Trusted Computing Group (TCG), "TCG EK Credential Profile for TPM Family 2.0; Level 0", January 26, 2022, https://trustedcomputinggroup.org/wp-content/uploads/TCG-EK-Credential-Profile-V-2.5-R2_published.pdf

[9] Trusted Computing Group (TCG), "TCG Platform Attribute Credential Profile, Specification Version 1.0, Revision 16", January 16, 2018, https://trustedcomputinggroup.org/wp-content/uploads/TCG-Platform-Attribute-Credential-Profile-Version-1.0.pdf

[10] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of Remote Attestation", International Journal of Information Security, vol. 10, April 23 2011, pp. 63–81, DOI 10.1007/s10207-011-0124-7

[11] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan, "Remote ATtestation procedureS (RATS) Architecture", RFC-9334, January 2023, DOI 10.17487/RFC9334

[12] R. W. Shirey, "Internet Security Glossary, Version 2", RFC-4949, August 2007, DOI 10.17487/RFC4949

[13] National Institute of Standards and Technology (NIST), "Strength of Function", https://csrc.nist.gov/glossary/term/strength_of_function

[14] A. Fuchs, H. Birkholz, I. McDonald, and C. Bormann, "Time-Based Uni-Directional Attestation", July 10, 2022, https://datatracker.ietf.org/doc/html/draft-birkholz-rats-tuda-07

[15] "Ima and EVM Concepts", https://ima-doc.readthedocs.io/en/latest/ima-concepts.html#ima-and-evm-concepts

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

[17] "Integrity Measurement Architecture (IMA)", https://docs.trustauthority.intel.com/main/articles/tpm-ima-logs.html

[18] S. Sisinni, "Verification of Software Integrity in Distributed Systems", 2021, Master's Thesis, Politecnico di Torino, https://webthesis.biblio.polito.it/20403/

[19] M. De Benedictis and A. Lioy, "Integrity verification of Docker containers for a lightweight cloud environment", Future Generation Computer Systems, vol. 97, February 2019, pp. 1–38, DOI 10.1016/j.future.2019.02.026
[20] Internet Engineering Task Force (IETF), https://www.ietf.org/about/
[21] European Telecommunications Standards Institute (ETSI), https://www.etsi.org/about
[22] L. Ferro and A. Lioy, "Standard-Based Remote Attestation: The Veraison Project", ITASEC-2024: The Italian Conference on CyberSecurity, Salerno (Italy), April 8-12, 2024, pp. 1–13, DOI 11583/2988310
[23] N. Schear, P. T. Cable II, and T. M. Moyer, "Bootstrapping and Maintaining Trust in the Cloud", Proceedings of the 32nd Annual Conference on Computer Security Application, New York (USA), December 5-8, 2016, pp. 65–77, DOI 10.1145/2991079.2991104
[24] Cloud Native Computing Foundation (CNCF), https://www.cncf.io/about/who-we-are/
[25] Cloud Native Computing Foundation (CNCF), https://keylime.dev/
[26] Veraison Project, https://github.com/veraison
[27] Keylime Developers, "Keylime Documentation", https://keylime.readthedocs.io/en/latest/design/overview.html
[28] Viper, https://github.com/spf13/viper
[29] H. Birkholz, T. Fossati, Y. Deshpande, N. Smith, and W. Pan, "Concise Reference Integrity Manifest", October 18, 2024, https://datatracker.ietf.org/doc/draft-ietf-rats-corim/
[30] J. Schaad, "CBOR Object Signing and Encryption (COSE)", RFC-8152, July 2017, DOI 10.17487/RFC8152
[31] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)", RFC-8949, December 2020, DOI 10.17487/RFC8949
[32] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format", RFC-8259, December 2017, DOI 10.17487/RFC8259
[33] L. Lundblade, G. Mandyam, J. O'Donoghue, and C. Wallace, "The Entity Attestation Token (EAT)", September 6, 2024, https://datatracker.ietf.org/doc/draft-ietf-rats-eat/
[34] T. Fossati, E. Voit, S. Trofimov, and H. Birkholz, "EAT Attestation Results", November 19, 2024, https://datatracker.ietf.org/doc/draft-fv-rats-ear/
[35] M. B. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "CBOR Web Token (CWT)", RFC-8392, May 2018, DOI 10.17487/RFC8392
[36] M. B. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)", RFC-7519, May 2015, DOI 10.17487/RFC7519
[37] Davis, Kyzer R. and Peabody, Brad and Leach, P., "Universally Unique IDentifiers (UUIDs)", RFC-9562, May 2024, DOI 10.17487/RFC9562
[38] S. Josefsson and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC-7468, April 2015, DOI 10.17487/RFC7468
[39] Trusted Computing Group (TCG), "TCG TPM Vendor ID Registry Family 1.2 and 2.0 Version 1.07, Revision 0.02", November 20, 2024, https://trustedcomputinggroup.org/wp-content/uploads/TCG-TPM-Vendor-ID-Registry-Family-1.2-and-2.0-Version-1.07-Revision-0.02_pub.pdf
[40] H. Birkholz, M. Eckel, S. Bhandari, E. Voit, B. Sulzen, L. Xia, T. Laffey, and G. Fedorkow, "A YANG Data Model for Challenge-Response-Based Remote Attestation (CHARRA) Procedures Using Trusted Platform Modules (TPMs)", RFC-9864, December 2024, DOI 10.17487/RFC9684
[41] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC-8017, November 2016, DOI 10.17487/RFC8017
[42] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)", International Journal of Information Security, vol. 1, January 2014, pp. 36–63, DOI 10.1007/s102070100002

Appendix A

Developer's Manual

This appendix provides comprehensive documentation of the main functionalities exposed by the components of the standard-based attestation framework. These functionalities include:

- **API endpoints**: outlining the available operations for components that expose their functionalities as REST API Servers.

The content covers explanations of the system's internal processes, including the structure of incoming requests and outgoing responses. Its purpose is to serve as a practical guide for understanding how the system operates, as well as to support future modifications and enhancements.

The subsections related to the Register and the Verifier describe only the APIs concerning the Registration and the Attestation procedures, respectively. A further subsection is added (A.4), including the DBs interactions API, in which the Register has reading and writing permissions while the Verifier only has reading permissions.

A.1 Agent

When the procedure is started by the Register or the Verifier, the Agent will start REST API Servers to be properly contacted on the specified port. In the following, more details are given.

A.1.1 ID Provider

POST `/startregister`

Start the registration procedure for the Agent.

Request JSON Object

- `nonce` ([ ]byte): nonce used to guarantee the freshness of the data.

Example request:

```
{
    "nonce": "G2bDUiujDREzz5XNXNzDhBZ+8Z71bktmr2SCTjqJ6qk="
}
```

Response JSON object

- `status` (HTTP status code): outcome of the registration procedure.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:36:36 GMT
Content-Length: 0
```

POST `/shutdown`

Proper closing of the ID Provider Server port created for the Agent.

Request JSON Object

- `id` (string): Device ID used to be identified by the Verifier.

Example request:

```
{
    "id": "0b0f9e94-a641-4299-ac00-c6cb76575033"
}
```

Response JSON object

No response as the server is shut down on receiving the request.

A.1.2 Attester

POST `/startattest`

Start the attestation procedure for the Agent.

Request JSON Object

- `nonce` ([ ]byte): nonce used to guarantee the freshness of the data;
- `whitelist ID` (string): ID of the list of the reference values associated to the Agent.

Example request:

```
{
    "nonce": "R6cEQiujDREyt8XNMXhDhBZ+9H89vdwmk6SCTjqJ=qk9",
    "WhitelistID": "a0b4799f-8b38-4e49-937f-02a75d6df739"
}
```

Response JSON object

- `status` (HTTP status code): outcome of the creation of the attestation object.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:39:40 GMT
Content-Length: 0
```

POST `/endattest`

End the attestation procedure for the Agent.

Request JSON Object

- `confirmation` (string): used to receive a signal by the Verifier to write in the logs that the attestation procedure has properly been executed.

Example request:

```
{
    "confirmation": "Attestation Ended"
}
```

Response JSON object

No response is given, the log is updated and the Agent is ready to start the next attestation procedure with the proper API (`/startattest`).

A.2 Register

The Register is a REST API Server that exposes a port to be contacted. In the discussion that follows, only the APIs regarding the registration procedure are reported.

POST `/agentaddr`

Start the registration procedure for the Agent, with the received IP and port.

Request JSON Object

- `ip` (string): IP of the Agent to register;
- `port` (int): port exposed by the Agent;
- `whitelist ID` (string): ID of the list of the reference values associated to the Agent.

Example request:

```
{
    "Ip": "192.168.0.103",
    "Port": 8081,
    "WhitelistID": "a0b4799f-8b38-4e49-937f-02a75d6df739"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the whitelist ID's search in the database.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:41:20 GMT
Content-Length: 0
```

POST `/nonce`

Start the registration procedure for the Agent, with a request performed by it.

Request JSON Object

- `first request` (string): request probe sent by the Agent.

Example request:

```
{
    "Hi! I'm a new Device, I'd like to register."
}
```

Response JSON object

- `status` (HTTP status code): to indicate the creation of a new nonce;
- `nonce` ([ ] byte): nonce used to guarantee the freshness of the data.

Example response:

```
HTTP/1.1 201 Created
Date: Mon, 24 Mar 2025 15:47:38 GMT
Content-Length: 44
Content-Type: text/plain; charset=utf-8

T9cEHiujBTEOt5XMKThDhBZ+9H89vdwmk6RHBjqJ=qj9
```

POST `/firstquote`

Verifies the EK certificate chain and the first quote.

Request JSON Object

- `EK certificate` (string): certificate of the EK in PEM format;
- `Quote` (struct): data structure containing data regarding the first quote;
- `AK Public Blob` ([ ] byte): encoding of the AK Public Area;
- `Nonce`([ ] byte): nonce created and sent with the previous API;
- `whitelist ID` (string): to keep track of the whitelist ID associated with the Agent by means of the other fields.

Example request:

```
{
    "EKCertificate": "-----BEGIN CERTIFICATE-----
    MIIElTCCA32gAwIBAgIEQzo1xzANBgkqhkiG9w0BAQsFADCBgzELMAk
    GA1UEBhMC\nREUxITAfBgNVBAoMGEluZmluZW9uIFRlY2hub2xvZ2ll
    BBRzEaMBgGA1UECwwR\nT1BUSUdBKFRNKSBUUE0yLjAxNTAzBgNVBAM
    EluZmluZW9uIE9QVElHQShUTSkg\nUlNBIE1hbnVmYWN0dXJpbmcgQ0
    hUnNhTWZyQ0EwMDAzMB4XDTE2MDEwMTEzMDk1MloXDTMxMDEw\nMTEz
    [...]
```

```
    xwpMtDI+3g6l+qlWMu8AeI5GGSzbSeLcUB\nMthS7hCMrO4tT6TUVTO
    sCACMvx3XMRiRix5+BEtt2i3tDd33Z9fW1M2RYY4\n2OUyNrIcmP6x/
    O2exZD8d2Y1THbyjYE63nP6zY92RdV3+OO5YAM9BMyJHAu4D6mz\n2K
    YAM9BMyJHAu4D6mz\n2KvvPF99qjnUR06D/NaI+VZFc/eFio5TZoA==
    -----END CERTIFICATE-----
    ",
    "Quote": [
        {
            "quote": "/1RDR4AYACIAC1K+2mQSGaq95am/72ZHyjO1CCVsFGLbJ...",
            "raw_sig": "ABYACwEADsYB8CwoRRFUIxrNzKl6NlmmAv5OkIw3Yuv...",
            "pcrs": {
                "hash": 11,
                "pcrs": {
                    "0": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "1": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "2": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "3": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "4": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "5": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "6": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "7": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "8": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                    "9": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="
                }
            }
        },
        ...
    ],
    "AKPubBlob": "AAEACwAFBHIAAAAQABYACwgAAAAAAAEAl9FwiS15df12T9xQB...",
    "Nonce": "uuN7blRrOAzz7LuDxRl1Ivr34Dke/DLQsdNQWqqyC6o=",
    "WhitelistID": "c90486dc-b218-4cd0-a420-3bd484cecac4"
}
```


Response JSON object

- `status` (HTTP status code): outcome of the verification of the certificate chain and the first quote.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:41:20 GMT
Content-Length: 0
```

POST `/registration`

Receives the data required to create the challenge.

Request JSON Object

- `AK Public Blob` ([ ] byte): encoding of the AK Public Area;
- `Name Data` ([ ] byte): encoding of the hash calculated over the AK Public Blob;
- `TPM EK Public Key` ([ ] byte): public key of the Endorsement Key pair of the TPM used as Root of Trust.

Example request:


```
{
    "AKPubBlob": "ACMACwAFAHIAAAAQABgACwADABAAIOSss9JAWBTYMC/d...",
    "NameData": "ACIAC/xbpnrmR2I1PeUxypyv3R5KZOQonOye5nadIEGv3ChG",
    "EKTPMPub": "AAEACwADALIAIINxl2dEhLP4GpDMjUal1yT9UtduBlILZPKh2
     hszFGmqAAYAgABDABAIAAAAAAABALCN2PaQVTTkTj2EVkWUs+Tdmri5fkZhhZA
     uQrOl3LN6HcJP2XQLAB67wzWkliMMNW9IecVrhc9qav89IZZpV015WaoOZtSLA
     vpBjUySaLUZGITOZecydbflgv/+Bkztvkjca/TcqVhvrhSqeq5O3Ew04r0RbUp
     /7Ww4z/x0piVL+gPkZCug71gkCpUf9s2lYHxulqDIqMPCxktEAhNkGOZBjW1uC
     SXFQwuHWAwn/0Bp3WAtkajdm9GVTp9FGP7/Jyyou88mbW4TpOMLE9KMFfchG9h
     NaG0Yaj1BMP4C0bAh11/9uSA574jg1o4Z+w2iOV8X/yxf4wGRyHDye5eOV0="
}
```


Response JSON object

- `status` (HTTP status code): to indicate that the integrity verification of the AK Public Blob has been properly completed and the challenge created;
- `challenge` ([ ] byte): encoding of the challenge created with the TPM EK public.

Example response:


```
HTTP/1.1 202 Accepted
Date: Mon, 24 Mar 2025 15:43:23 GMT
Content-Length: 457

AEQAIN+nkayy/TlQq/zSBMTTgwx3e7/pmVfFxBrHOT4NtT0FtftIc755q
W8Hm/6/d2aP3BR1ocYthBu0cT6ikfl9NsboZQEAGgilBwE72E/UbdWwLx
1YuuN8ByumVC9ipmzs/sMa7vSZIQJ7DnMjKYAUM2I6pp38opV/hLwflV7
WMy3R5T02FjpqD/1PPJOyEMFqCKDpceav9Tct sAC4GcUzkLiLkkGNjh3
/7vb/7I+iDt5FzWPlsAGZc90J42IdePNLhJKWgKnvuETEOvy+OP36gfXC
IDXmJosuquL6LptVDmVgWFSCkyDvv1qAphJjAF8vfjX7tf+aXKtIze7XB
3QfV9mwME/5o8ih4mu2e3lvSJePBxIUUtmjiSN+GCid7jWTQdUijjR4EB
3E9+uqUvfBSMSYrFWyTA3mpLMupqs6ASXBA/Pg==
```

POST `/challenge`

Receives the challenge solution and verifies it.

Request JSON Object

- `Solution` ([ ] byte): challenge solution (secret decrypted using the TPM EK private);
- `whitelist ID` (string): to keep track of the whitelist ID associated with the Agent through the solution.

Example request:


```
{
    "Solution": "dv3Rk1FtrzEsCaYX1nTAOpWdQUazkYFFBJYuOCIg7Ds=",
    "WhitelistID": "f13ceb44-5de7-4e25-ac72-975253823048"
}
```


Response JSON object

- `status` (HTTP status code): indicate the result of the verification of the challenge.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:41:20 GMT
Content-Length: 0
```

POST `/updatestate`

Update the state of the Agent from "Registered" to "Attested" after performing the attestation procedure properly and verifying the attestation result.

Request JSON Object

- `Id` (string): ID of the device whose state must be updated.;
- `JSON Web Token` ([ ] byte): encoding of the registration result in JWT format;
- `JSON Web Key` (struct): data structure containing the relevant fields related to the public key used to verify the JWT.

Example request:

```
{
    "Id": "2ef61c89-04b8-4878-97f2-52b06c3b7245",
    "JWT": "ZXlKaGJHY2lPaUpGVXpJMU5pSXNJblI1YONJNklrcFhWQOo5Lm
     V5SmxZWEl1ZG1WeWFXWnBaWEl0YVdRaU9uc2lZblZwYkdRaU9pSjJaWEp
     wWm1sbGNpSXNJbVJsZG1Wc2IzQmxjaUk2SW5Sb1pYTnBjeUJQVlNKOUxD
     SmxZWFJmY0hKdlptbHNaU0k2SW5SaFp6cG5hWFJvZFdJdVkyOXRMREl3T
     WpNNmRtVnlZV2x6YjI0dlpXRnlJaXdpYVdGMElqb3hOelF5T0RJMU1Eaz
     JMQ0p6ZFdKdGIyUnpJanA3SW1Oa05EVTJNVFF5TFdJd1pXUXRORUJtTWk
     xaE16RTBMVFF4WlRVek0yRmlaV1JpWWlJNmV5SmxZWEl1YzNSaGRIVnpJ
     am9pWVdabWFYSnRhVzVuSWl3aVpXRnlMblJ5ZFhOMGQyOXlkR2hwYm1We
     mN5MTJaV04wYjNJaU9uc2lZMjl1Wm1sbmRYSmhkR2x2YmlJNk1pd2laWG
     hsWTNWMFlXSnNaWE1pT2pJc0ltWnBiR1V0YzNsemRHVnRJam93TENKb1l
     YSmtkMkZ5WlNJNk1pd2lhVzV6ZEdGdVkyVXRhV1JsYm5ScGRIa2lPaklz
     SW5KMWJuUnBiV1V0YjNCaGNYVmxJam93TENKemIzVnlZMlZrTFdSaGRHR
     WlPakFzSW5OMGIzSmhaMlV0YjNCaGNYVmxJam93ZlgxOWZRLnlMZTY0M2
     tpMVBpSUVZUUYyQ1RDX0RzVkRpR09DbzNOdjF4WGxFdkhMcjlkMTlnVlE
     wbm1PNV9SeGlXMlFMVVN3N0xCeURUTjdUem52azR5MzdJMVpn",
    "JWK": {
        "crv":"P-256",
        "kty":"EC",
        "x":"Zp0T3ZVSGkCu_maxczaEVproY-Nxar8unZfAymuTHBc",
        "y":"TUnJhIG-8IBOPNw-s-4Ozq-vEWZi7P_BYT7jr2buuxE"
    }
}
```

Response JSON object

- `status` (HTTP status code): indicate that the state has been properly updated.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:41:20 GMT
Content-Length: 0
```

POST `/registrationresult`

Give information about the registration of an Agent.

Request JSON Object

- `Id` (string): ID of the device whose registration result information are requested.

Example request:

```
{
    "Id": "dad2e04e-4396-4048-a186-9ed60d84500b"
}
```

Response JSON object

- `JSON Web Token` ([ ] byte): encoding of the registration result in JWT format;
- `JSON Web Key` (struct): data structure containing the relevant fields related to the public key used to verify the JWT.

Example response:

```
{
    "JWT": "ZXlKaGJHY2lPaUpGVXpJMU5pSXNJblI1Y0NJNklrcFhWQ0o5Lm
     V5SmxZWEl1ZG1WeWFXWnBaWEl0YVdRaU9uc2lZblZwYkdRaU9pSjJaWEp
     wWm1sbGNpSXNJbVJsZG1Wc2IzQmxjaUk2SW5Sb1pYTnBjeUJQVlNKOUxD
     SmxZWFJmY0hKdlptbHNaU0k2SW5SaFp6cG5hWFJvZFdJdVkyOXRMREl3T
     WpNNmRtVnlZV2x6YjI0dlpXRnlJaXdpYVdGMElqb3hOelF5T0RJMU1Eaz
     JMQ0p6ZFdKdGIyUnpJanA3SW1Oa05EVTJNVFF5TFdJd1pXUXROREJtTWk
     xaE16RTBMVFF4WlRVek0yRmlaV1JpWWlJNmV5SmxZWEl1YzNSaGRIVnpJ
     am9pWVdabWFYSnRhVzVuSWl3aVpXRnlMblJ5ZFhOMGQyOXlkR2hwYm1We
     mN5MTJaV04wYjNJaU9uc2lZMjl1Wm1sbmRYSmhkR2x2YmlJNk1pd2laWG
     hsWTNWMFlXSnNaWE1pT2pJc0ltWnBiR1V0YzNsemRHVnRJam93TENKb1l
     YSmtkMkZ5WlNJNk1pd2lhVzV6ZEdGdVkyVXRhV1JsYm5ScGRIa2lPaklz
     SW5KMWJuUnBiV1V0YjNCaGNYVmxJam93TENKemIzVnlZMlZrTFdSaGRHR
     WlPakFzSW5OMGIzSmhaMlV0YjNCaGNYVmxJam93ZlgxOWZRLnlMZTY0M2
     tpMVBpSUVZUUYyQ1RDX0RzVkRpR09DbzNOdjF4WGxFdkhMcjlkMTlnVlE
     wbm1PNV9SeGlXMlFMVVN3N0xCeURUTjdUem52azR5MzdJMVpn",
    "JWK": {
        "crv":"P-256",
        "kty":"EC",
        "x":"Zp0T3ZVSGkCu_maxczaEVproY-Nxar8unZfAymuTHBc",
        "y":"TUnJhIG-8IBOPNw-s-4Ozq-vEWZi7P_BYT7jr2buuxE"
    }
}
```

A.3 Verifier

POST `/deviceid`

Start the attestation procedure for the Agent, given its device ID.

Request JSON Object

- `Id` (string): ID of the device that has to be attested.

Example request:

```
{
    "Id": "dad2e04e-4396-4048-a186-9ed60d84500b"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search of the associated Agent's IP and port.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:41:20 GMT
Content-Length: 0
```

POST `/nonce`

Start the attestation procedure for the Agent, with a request performed by it.

Request JSON Object

- `Device ID` (string): ID of the device that wants to be attested.

Example request:

```
{
    "Id": "dad2e04e-4396-4048-a186-9ed60d84500b"
}
```

Response JSON object

- `status` (HTTP status code): to indicate that the nonce has been created;
- `nonce` ([ ] byte): nonce used to guarantee the freshness of the data.

Example response:

```
HTTP/1.1 201 Created
Date: Mon, 24 Mar 2025 15:47:38 GMT
Content-Length: 44
Content-Type: text/plain; charset=utf-8

T9cEHiujBTEOt5XMKThDhBZ+9H89vdwmk6RHBjqJ=qj9
```

POST `/verifyattestation`

Verifies an attestation object and creates the attestation result.

Request JSON Object

- `Device ID` (string): ID of the device that sends the attestation object;
- `Attestation` (struct): data structure containing the attestation object fields;
- `Ima` ([ ] string): contains the measurements.

Example request:


```
{
    "DeviceID": "8ad78306-f7c3-4c11-96a8-19889f0749eb",
    "Attestation": {
    "ak_pub": "ACMACwAFAHIAAAAQABgACwADABAAIHC5DVh/B5f7ibZ8hB6HR/
    U0g3Zo61ndbF7C4NMDoa2zACC3qvDMHYoWJizZcYAMFRNhCy9NSY6w8jyDNy
    9IayO/9w==",
    "quotes": [
        {
        "quote": "/1RDR4AYACIAC9voKui0+9Z+kDfUeXaaBsPs0tJ4q+G/DJU
        rS4+Fs8qtACDY9gREJ5DilNboQvnJIvCkoebvU7If4fq7nS4+PsO+tgA
        AAAAAAJOyjGNSQxsJoEEBzyKXp1/qieAAAAABAAQD/wcAACAzED/FJOg
        QWTsxS2YJpgX9uf5RO8VmzYs+zqHUv8+cqQ==",
        "raw_sig": "ABgACwAg7q1WvmquURGtx69hksDO9hqccbxOrkikC68U9
        F5/oH8AIIsizdyZwfPc0VpdcwvDoSHj/S0Z+vGkq+pHg1kLM/oG",
        "pcrs": {
            "hash": 4,
            "pcrs": {
                "0": "PcrqJdyGVU2UuUqlvI9zWkkhKvg=",
                "1": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "10": "aoVxiToW8TQdqF2txPNz6CK4zNM=",
                "2": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "3": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "4": "Q9wQ7XSc8q3vImsKhkRswmcmS2c=",
                "5": "R0QbpDEBK4YPrg9IolH6KPCH1Fc=",
                "6": "sqg7Dr8vg3Qpmlsr38MeqVWtcjY=",
                "7": "vXnshUFQBHKIhjHaKLggqMBcEfE=",
                "8": "6DcnHoQJr7wawosjtucv7yHmrKM=",
                "9": "au2zvtxG4uzylVHjIDyn6Co7FOA="
                }
            }
        },
        {
        "quote": "/1RDR4AYACIAC9voKui0+9Z+kDfUeXaaBsPs0tJ4q+G/DJUr
        S4+Fs8qtACDY9gREJ5DilNboQvnJIvCkoebvU7If4fq7nS4+PsO+tgAAA
        AAAJRljGNSQxsJoEEBzyKXp1/qieAAAAABAAsD/wcAACB2BcRVfPieqY
        yttWk65Qoamtg7GS/seaM/b1IzwyeIFA==",
        "raw_sig": "ABgACwAgnztsQx64mRfm8/AQTD0b5Wscie4DOkVfWDOlcI
        dcdyQAIPNHKYcXknyDNcxUKqK4LA6dMvV1jws9vbzdXbhZe/6m",
        "pcrs": {
            "hash": 11,
            "pcrs": {
                "0": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "1": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "10": "CB1xYi//JPJIhmCi+Fd/y+sBJQh0WW1bNdbHOByMYIw=",
                "2": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "3": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "4": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "5": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "6": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "7": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
                "8": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=",
```

```
                "9": "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA="}
            }
        }],
        "event_log": "AAAAAAgAAADEL+2tJoIAyx0V+XhBw0Tnna4zIBAAAAAe+2
        tUDB1VQKStTvS/F7g6BwAAAAEAAIAvIBEqP1U5iyCODEJoE4m0y1sYIzQAA
        ABh3+SLypPSEaoNAOCYAyuMCgAAAAAAAAAAAAAAAAAAAFMAZQBjAHUAcgBl
        AEIAbwBvAHQABwAAAAEAAICbE4cwbrt/+OeV5753VjZmu/RRbiQAAABh3+S
        ...",
        "TeeAttestation": null
    },
    "Ima": [
        "10 6bdad[...]5dcdd ima-ng sha256 7b643[...]07d61 boot_aggregate",
        "10 0b5a1[...]1e1ac ima-ng sha256 d9d17[...]b3e46 /init",
        "10 fbc98[...]28c6f ima-ng sha256 f5f25[...]36422 /usr/bin/sh",
        "10 a4c34[...]67e79 ima-ng sha256 595bc[...]72d89
         /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2",
        "10 a1c2f[...]10189 ima-ng sha256 bc375[...]967c7 /etc/ld.so.cache",
        "10 226af[...]87113 ima-ng sha256 5955d[...]15fa7
         /usr/lib/x86_64-linux-gnu/libc.so.6",
        "10 0f8b2[...]9e9a6 ima-ng sha256 91f24[...]0d9fe /conf/arch.conf",
        "10 4ca27[...]0e510 ima-ng sha256 5235c[...]592c2
         /conf/initramfs.conf",
        "10 010d8[...]6339c ima-ng sha256 1b575[...]71a94 /scripts/functions",
        "10 9debe[...]7a04f ima-ng sha256 83464[...]74942
         /scripts/init-top/ORDER",
        ...
    ]
}
```


Response JSON object

- `status` (HTTP status code): to indicate the result of the attestation object verification.

Example response:

```
HTTP/1.1 200 OK
Date: Mon, 24 Mar 2025 15:41:20 GMT
Content-Length: 0
```

POST `/attestationresult`

Give information about the attestation of an Agent.

Request JSON Object

- `Id` (string): ID of the device whose attestation result information is requested.

Example request:

```
{
    "Id": "dad2e04e-4396-4048-a186-9ed60d84500b"
}
```

Response JSON object

- `JSON Web Token` ([ ] byte): encoding of the registration result in JWT format;

- `JSON Web Key` (struct): data structure containing the relevant fields related to the public key used to verify the JWT.

Example response:


```
{
    "JWT": "ZXlKaGJHY2lPaUpGVXpJMU5pSXNJblI1YONJNklrcFhWQOo5Lm
     V5SmxZWEl1ZG1WeWFXWnBaWEl0YVdRaU9uc2lZblZwYkdRaU9pSjJaWEp
     wWm1sbGNpSXNJbVJsZG1Wc2IzQmxjaUk2SW5Sb1pYTnBjeUJQVlNKOUxD
     SmxZWFJmY0hKdlptbHNaU0k2SW5SaFp6cG5hWFJvZFdJdVkyOXRMREl3T
     WpNNmRtVnlZV2x6YjI0dlpXRnlJaXdpYVdGMElqb3hOelF5T0RJMU1Eaz
     JMQ0p6ZFdKdGIyUnpJanA3SW1Oa05EVTJNVFF5TFdJd1pXUXRORUJtTWk
     xaE16RTBMVFF4WlRVek0yRmlaV1JpWWlJNmV5SmxZWEl1YzNSaGRIVnpJ
     am9pWVdabWFYSnRhVzVuSWl3aVpXRnlMblJ5ZFhOMGQyOXlkR2hwYm1We
     mN5MTJaV04wYjNJaU9uc2lZMjl1Wm1sbmRYSmhkR2x2YmlJNk1pd2laWG
     hsWTNWMFlXSnNaWE1pT2pJc0ltWnBiR1V0YzNsemRHVnRJam93TENKb1l
     YSmtkMkZ5WlNJNk1pd2lhVzV6ZEdGdVkyVXRhV1JsYm5ScGRIa2lPaklz
     SW5KMWJuUnBiV1V0YjNCaGNYVmxJam93TENKemIzVnlZMlZrTFdSaGRHR
     WlPakFzSW5OMGIzSmhaMlV0YjNCaGNYVmxJam93ZlgxOWZRLnlMZTY0M2
     tpMVBpSUVZUUYyQ1RDX0RzVkRpR09DbzNOdjF4WGxFdkhMcjlkMTlnVlE
     wbm1PNV9SeGlXMlFMVVN3N0xCeURUTjdUem52azR5MzdJMVpn",
    "JWK": {
        "crv":"P-256",
        "kty":"EC",
        "x":"Zp0T3ZVSGkCu_maxczaEVproY-Nxar8unZfAymuTHBc",
        "y":"TUnJhIG-8IBOPNw-s-4Ozq-vEWZi7P_BYT7jr2buuxE"
    }
}
```


A.4 DBs Interactions APIs

The Register and the Verifier expose some APIs to interact with the two databases, a relational and a non-relational one. The Register accesses these with reading and writing permissions, while the Verifier only has reading permissions.

In particular, the relational database implemented with `PostgreSQL`has three tables:

1. **Agents**, to store information about the devices performing the attestation;
2. **TPM CA Certificates**, to store the trusted TPM certificates CAs;
3. **TPM Vendors**, to store information related to the TPM vendors.

While the non-relational database implemented with `MongoDB` stores the golden values of the measurements (**references** collection) against which the measurements received by the Agent are verified.

A.4.1 Agents Table

The table is composed of the following fields:

- **id** `VARCHAR PRIMARY KEY`,
- **ipaddr_port** `VARCHAR`,
- **ak** `VARCHAR`,

- **whitelist_id** `VARCHAR`,
- **state** `VARCHAR`

POST `/readAgent`

Read information about an Agent specifying its Device ID.

Request JSON Object

- `Device ID` (string): ID of the device whose information is to be retrieved.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search of the Agent's;
- `database entry` (string): entry containing information about the requested Agent.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 13:23:00 GMT
Content-Length: 344
Content-Type: text/plain; charset=utf-8

{
    "DeviceID":"eb27c762-7285-41f5-90c7-a241ca18a6d7",
    "Ip_Port":"192.168.0.103:8082",
    "AK":"-----BEGIN PUBLIC KEY-----\n
    MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEbqv0NjUV/5
    9db6quy7neoZR24KiL\nZvH6u6Mo9wtu9++Uh4+K8lCthRNo
    ZnKgTErP4/NNUoKO72EY6ED+rxdP5Q==\n
          -----END PUBLIC KEY-----\n",
    "WhitelistID":"980c169a-ad22-443a-a5c7-64a6043e5f27",
    "State":"Attested"
}
```

POST `/readIpPort`

Read the IP and port of the Agent specified by its Device ID.

Request JSON Object

- `Device ID` (string): ID of the device whose IP and port are to be retrieved.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search of the Agent's;
- `database entry field` (string): column of the entry containing IP and port of the requested Agent.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 13:38:38 GMT
Content-Length: 18
Content-Type: text/plain; charset=utf-8

192.168.0.103:8082
```

POST `/readAK`

Read the Authentication Key of the Agent specified by its Device ID.

Request JSON Object

- `Device ID` (string): ID of the device whose AK is to be retrieved.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search of the Agent's;
- `database entry field` (string): column of the entry containing the AK of the requested Agent.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 13:49:30 GMT
Content-Length: 178
Content-Type: text/plain; charset=utf-8

-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEbqv0NjUV/59db6quy7neoZR24K
iLZvH6u6Mo9wtu9++Uh4+K8lCthRNoZnKgTErP4/NNUoKO72EY6ED+rxdP5Q==
-----END PUBLIC KEY-----
```

POST `/readWhitelistId`

Read the Whitelist ID of the Agent specified by its Device ID.

Request JSON Object

- `Device ID` (string): ID of the device whose Whitelist ID is to be retrieved.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search of the Agent's;
- `database entry field` (string): column of the entry containing the Whitelist ID of the requested Agent.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 13:52:05 GMT
Content-Length: 36
Content-Type: text/plain; charset=utf-8

980c169a-ad22-443a-a5c7-64a6043e5f27
```

POST `/readState`

Read the Agent's state specified by its Device ID.

Request JSON Object

- `Device ID` (string): ID of the device whose state is to be retrieved.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search of the Agent's;
- `database entry field` (string): column of the entry containing the state of the requested Agent.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 13:56:38 GMT
Content-Length: 8
Content-Type: text/plain; charset=utf-8

Attested
```

POST `/updateIpPort`

Update the Agent's IP and port specified by its Device ID with new values.

Request JSON Object

- `Device ID` (string): ID of the device whose IP and port are updated;
- `IP` (string): new IP;
- `Port` (string): new port.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7",
    "Ip": "0.0.0.0",
    "Port": "22"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's entry update;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 14:01:47 GMT
Content-Length: 44
Content-Type: text/plain; charset=utf-8

eb27c762-7285-41f5-90c7-a241ca18a6d7 updated
```

POST `/updateWhitelistId`

Update the Agent's Whitelist ID specified by its Device ID with a new value.

Request JSON Object

- `Device ID` (string): ID of the device whose Whitelist ID is updated;
- `Whitelist ID` (string): new Whitelist ID.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7",
    "WhitelistId": "<NewWhitelistID>"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's entry update;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 14:10:34 GMT
Content-Length: 44
Content-Type: text/plain; charset=utf-8

eb27c762-7285-41f5-90c7-a241ca18a6d7 updated
```

The updates concerning the AK and the state of the Agent are not reported as these can only be changed by going through the registration or the attestation procedures.

POST `/deleteAgent`

Delete the Agent entry specified by its Device ID.

Request JSON Object

- `Device ID` (string): ID of the device to delete.

Example request:

```
{
    "Id": "eb27c762-7285-41f5-90c7-a241ca18a6d7"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's entry deletion;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 200 OK
Date: Tue, 25 Mar 2025 14:15:31 GMT
Content-Length: 44
Content-Type: text/plain; charset=utf-8

eb27c762-7285-41f5-90c7-a241ca18a6d7 deleted
```

A.4.2 TPM CA Certificates Table

The table is composed of the following fields:

- **certificateId** `SERIAL PRIMARY KEY`,
- **cn** (i.e. Common Name) `TEXT NOT NULL`,
- **PEMcertificate** `TEXT NOT NULL`.

With a specification `UNIQUE (cn, PEMcertificate)` to assure the uniqueness of this couple.

POST `/insertCertificate`

Insert a new certificate in the database.

Request JSON Object

- `Common Name` (string): common name of the certificate to insert;
- `PEM certificate` (string): string encoding of the certificate in PEM format.

Example request:

```
{
    "CommonName": "Infineon OPTIGA(TM) RSA Manufacturing CA 003",
    "PEMCertificate": "-----BEGIN CERTIFICATE-----\n
     MIIElTCCA32gAwIBAgIEQzo1xzANBgkqhkiG9w0BAQsFADCBgzELMAk
     GA1UEBhMC\nREUxITAfBgNVBAoMGEluZmluZW9uIFRlY2hub2xvZ2ll
     BBRzEaMBgGA1UECwwR\nT1BUSUdBKFRNKSBUUE0yLjAxNTAzBgNVBAM
     EluZmluZW9uIE9QVElHQShUTSkg\nUlNBIE1hbnVmYWN0dXJpbmcgQ0
     hUnNhTWZyQ0EwMDAzMB4XDTE2MDEwMTEzMDk1MloXDTMxMDEw\nMTEz
     [...]
     xwpMtDI+3g6l+qlWMu8AeI5GGSzbSeLcUB\nMthS7hCMrO4tT6TUVTO
     sCACMvx3XMRiRix5+BEtt2i3tDd33Z9fW1M2RYY4\n2OUyNrIcmP6x/
     O2exZD8d2Y1THbyjYE63nP6zY92RdV3+OO5YAM9BMyJHAu4D6mz\n2K
     YAM9BMyJHAu4D6mz\n2KvvPF99qjnUR06D/NaI+VZFc/eFio5TZoA==
     \n-----END CERTIFICATE-----"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's entry insertion;
- `confirmation message` (string): retrieves the certificateId of the new certificate.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 15:36:49 GMT
Content-Length: 47
Content-Type: text/plain; charset=utf-8

Inserted certificate with the following ID: 481
```

POST `/readCertificate`

Reads the certificate entry specified by its certificate Id.

Request JSON Object

- `Certificate Id` (string): ID of the certificate to retrieve.

Example request:

```
{
    "certificateId": "481"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `database entry` (string): entry containing information about the requested certificate.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 15:45:29 GMT
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked

{
    "CertificateID":"481",
    "CommonName":"Infineonn OPTIGA(TM) RSA Manufacturing CA 003",
    "PEMCertificate": "-----BEGIN CERTIFICATE-----\n
     MIIElTCCA32gAwIBAgIEQzo1xzANBgkqhkiG9w0BAQsFADCBgzELMAk
     GA1UEBhMC\nREUxITAfBgNVBAoMGEluZmluZW9uIFRlY2hub2xvZ2ll
     BBRzEaMBgGA1UECwwR\nT1BUSUdBKFRNKSBUUE0yLjAxNTAzBgNVBAM
     EluZmluZW9uIE9QVElHQShUTSkg\nUlNBIE1hbnVmYWN0dXJpbmcgQ0
     hUnNhTWZyQ0EwMDAzMB4XDTE2MDEwMTEzMDk1MloXDTMxMDEw\nMTEz
     [...]
     xwpMtDI+3g6l+qlWMu8AeI5GGSzbSeLcUB\nMthS7hCMrO4tT6TUVTO
     sCACMvx3XMRiRix5+BEtt2i3tDd33Z9fW1M2RYY4\n2OUyNrIcmP6x/
     O2exZD8d2Y1THbyjYE63nP6zY92RdV3+OO5YAM9BMyJHAu4D6mz\n2K
     YAM9BMyJHAu4D6mz\n2KvvPF99qjnUR06D/NaI+VZFc/eFio5TZoA==
     \n-----END CERTIFICATE-----"
}
```

POST `/readCertificateCN`

Reads the certificate's common name specified by its certificate Id.

Request JSON Object

- `Certificate Id` (string): ID of the certificate's common name to retrieve.

Example request:

```
{
    "certificateId": "481"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `database entry field` (string): column of the entry containing the certificate's common name requested.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 15:51:01 GMT
Content-Length: 45
Content-Type: text/plain; charset=utf-8

Infineon OPTIGA(TM) RSA Manufacturing CA 003
```

POST `/readCertificatePEM`

Reads the certificate specified by its certificate Id.

Request JSON Object

- `Certificate Id` (string): ID of the certificate to retrieve.

Example request:

```
{
    "certificateId": "481"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `database entry field` (string): string encoding of the column of the entry containing the certificate in PEM format requested.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 15:53:11 GMT
Content-Length: 2036
Content-Type: text/plain; charset=utf-8

-----BEGIN CERTIFICATE-----
MIIElTCCA32gAwIBAgIEQzo1xzANBgkqhkiG9w0BAQsFADCBgzELMAk
GA1UEBhMC\nREUxITAfBgNVBAoMGEluZmluZW9uIFRlY2hub2xvZ2ll
BBRzEaMBgGA1UECwwR\nT1BUSUdBKFRNKSBUUE0yLjAxNTAzBgNVBAM
EluZmluZW9uIE9QVElHQShUTSkg\nUlNBIE1hbnVmYWN0dXJpbmcgQ0
hUnNhTWZyQ0EwMDAzMB4XDTE2MDEwMTEzMDk1MloXDTMxMDEw\nMTEz
[...]
xwpMtDI+3g6l+qlWMu8AeI5GGSzbSeLcUB\nMthS7hCMrO4tT6TUVTO
sCACMvx3XMRiRix5+BEtt2i3tDd33Z9fW1M2RYY4\n2OUyNrIcmP6x/
O2exZD8d2Y1THbyjYE63nP6zY92RdV3+OO5YAM9BMyJHAu4D6mz\n2K
YAM9BMyJHAu4D6mz\n2KvvPF99qjnUR06D/NaI+VZFc/eFio5TZoA==
-----END CERTIFICATE-----
```

POST `/readCertificatePEMbyCN`

Reads the certificate specified by its Common Name.

Request JSON Object

- `Common Name` (string): common name of the certificate to retrieve.

Example request:

```
{
    "CommonName": "Infineon OPTIGA(TM) RSA Manufacturing CA 003"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `database entry field` (string): string encoding of the column of the entry containing the certificate in PEM format requested.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 15:56:10 GMT
Content-Length: 2036
Content-Type: text/plain; charset=utf-8

-----BEGIN CERTIFICATE-----
MIIElTCCA32gAwIBAgIEQzo1xzANBgkqhkiG9w0BAQsFADCBgzELMAk
GA1UEBhMC\nREUxITAfBgNVBAoMGEluZmluZW9uIFRlY2hub2xvZ2ll
BBRzEaMBgGA1UECwwR\nT1BUSUdBKFRNKSBUUE0yLjAxNTAzBgNVBAM
EluZmluZW9uIE9QVElHQShUTSkg\nUlNBIE1hbnVmYWN0dXJpbmcgQ0
hUnNhTWZyQ0EwMDAzMB4XDTE2MDEwMTEzMDk1MloXDTMxMDEw\nMTEz
[...]
xwpMtDI+3g6l+qlWMu8AeI5GGSzbSeLcUB\nMthS7hCMrO4tT6TUVTO
sCACMvx3XMRiRix5+BEtt2i3tDd33Z9fW1M2RYY4\n2OUyNrIcmP6x/
O2exZD8d2Y1THbyjYE63nP6zY92RdV3+OO5YAM9BMyJHAu4D6mz\n2K
YAM9BMyJHAu4D6mz\n2KvvPF99qjnUR06D/NaI+VZFc/eFio5TZoA==
-----END CERTIFICATE-----
```

POST `/deleteCertificate`

Reads the certificate specified by its Common Name.

Request JSON Object

- `Certificate Id` (string): Id of the certificate to delete.

Example request:

```
{
    "certificateId": "481"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's deletion;
- `confirmation message` (string): retrieves the certificateId of the deleted certificate.

Example response:

```
HTTP/1.1 200 OK
Date: Tue, 25 Mar 2025 15:59:47 GMT
Content-Length: 25
Content-Type: text/plain; charset=utf-8

CertificateId 481 deleted
```

A.4.3 TPM Vendors Table

The table is composed of the following fields:

- **vendorId** `SERIAL`,
- **name** `TEXT NOT NULL`,
- **TCGIdentifier** `TEXT NOT NULL UNIQUE`.
- **PlatformModel** `TEXT`.
- **FirmwareVersion** `TEXT`.

POST `/insertTPMvendor`

Insert a new TPM vendor in the database.

Request JSON Object

- `Name` (string): name of the vendor to insert;
- `TCG Identifier` (string): identifier defined by the TCG;
- `Platform Model` (string): optional field;
- `Firmware Version` (string): optional field.

Example request:

```
{
    "Name": "AMD",
    "TCGIdentifier": "id:414D4400",
    "PlatformModel": "",
    "FirmwareVersion": "<LatestFirmware>"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's insertion.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 16:26:25 GMT
Content-Length: 0
```

POST `/readTPMvendor`

Read a TPM vendor entry from the database with the specified TCG identifier.

Request JSON Object

- `TCG Identifier` (string): Id of the vendor to read.

Example request:

```
{
    "TCGIdentifier": "id:414D4400"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `database entry` (string): entry containing the TPM vendor to retrieve.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 16:29:12 GMT
Content-Length: 102
Content-Type: text/plain; charset=utf-8

{
    "VendorID":"3145",
    "Name":"AMD",
    "TCGIdentifier":"id:414D4400",
    "PlatformModel":"",
    "FirmwareVersion":"<LatestFirmware>"
}
```

POST `/readTPMvendorName`

Read a TPM vendor name from the database with the specified TCG identifier.

Request JSON Object

- `TCG Identifier` (string): Id of the vendor whose name is to be retrieved.

Example request:

```
{
    "TCGIdentifier": "id:414D4400"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `name` (string): column of the entry containing the TPM vendor name to retrieve.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 16:36:26 GMT
Content-Length: 3
Content-Type: text/plain; charset=utf-8

AMD
```

POST `/readTPMvendorPlatform`

Read a TPM vendor Platform Model from the database with the specified TCG identifier.

Request JSON Object

- `TCG Identifier` (string): Id of the vendor whose Platform Model is to be retrieved.

Example request:

```
{
    "TCGIdentifier": "id:414D4400"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `platform model` (string): column of the entry containing the TPM vendor platform model to retrieve.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 16:39:46 GMT
Content-Length: 0
```

POST `/readTPMvendorFirmware`

Read a TPM vendor Firmware Version from the database with the specified TCG identifier.

Request JSON Object

- `TCG Identifier` (string): Id of the vendor whose Firmware Version is to be retrieved.

Example request:

```
{
    "TCGIdentifier": "id:414D4400"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `firmware version` (string): column of the entry containing the TPM vendor firmware version to retrieve.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 16:41:33 GMT
Content-Length: 16

<LatestFirmware>
```

POST `/updateTPMvendorPlatform`

Update a TPM vendor Platform Model into the database with the specified TCG identifier.

Request JSON Object

- `TCG Identifier` (string): Id of the vendor whose Platform Model is to be updated.

Example request:

```
{
    "TCGIdentifier": "id:414D4400",
    "PlatformModel": "<NewPlatformModel>"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's update;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 16:49:11 GMT
Content-Length: 19
Content-Type: text/plain; charset=utf-8

id:414D4400 updated
```

POST `/updateTPMvendorFirmware`

Update a TPM vendor Firmware Version into the database with the specified TCG identifier.

Request JSON Object

- `TCG Identifier` (string): Id of the vendor whose Firmware Version is to be updated.

Example request:

```
{
    "TCGIdentifier": "id:414D4400",
    "FirmwareVersion": "<NewFirmwareVersion>"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's update;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 16:50:32 GMT
Content-Length: 19
Content-Type: text/plain; charset=utf-8

id:414D4400 updated
```

POST `/deleteTPMvendor`

Delete a TPM vendor from the database with the specified TCG identifier.

Request JSON Object

- `TCG Identifier` (string): Id of the vendor that is to be deleted.

Example request:

```
{
    "TCGIdentifier": "id:414D4400"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's deletion;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 200 OK
Date: Tue, 25 Mar 2025 16:56:56 GMT
Content-Length: 19
Content-Type: text/plain; charset=utf-8

id:414D4400 deleted
```

A.4.4 References Collection

The collection is composed of the following fields:

- **ID** `string`,
- **References** `CoMID`

POST `/insertreference`

Insert a new reference measurement list, creating a new ID.

Request JSON Object

- `References` (CoMID): measurement list of the references to be inserted.

Example request:

```
{
    "environment": {},
    "measurements": [
        {
         "value": {
            "digests":
            ["sha-256;e2Q2sMmPYjgIZtlDLCrw7gjOFqFxvaaVGuzZXuEwfWE="],
            "filename": "boot_aggregate"
            }
        },
        {
         "value": {
            "digests":
            ["sha-256;2dF3XWQ/b3ChpvZG3+AjBSdl9VihZ+xYlRrS8QE7PkY="],
            "filename": "/init"
            }
        },
        {
        "value": {
            "digests":
            ["sha-256;9fJdoyH5NBRiIF2P1uyRi7qJrV6l3J9th11RdNGTZCI="],
            "filename": "/usr/bin/sh"
            }
        },
        {
        "value": {
            "digests":
            ["sha-256;WVvNwwaZlxGlqceH5dAWGPlOuSxMCaF46HWjJPEHLYk="],
            "filename": "/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2"
            }
        }
        ...
    ]
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's insertion;

- `confirmation message` (string): to retrieve the Whitelist ID of the new entry.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 16:58:22 GMT
Content-Length: 94
Content-Type: text/plain; charset=utf-8

Inserted measurement list with the following WhitelistID:
cb1e0307-3fb0-450b-99f2-7b1c536d0507
```

POST `/readreference`

Read a measurement list considering the Whitelist ID.

Request JSON Object

- `Whitelist ID` (string): ID of the measurement to be retrieved.

Example request:

```
{
    "WhitelistId": "cb1e0307-3fb0-450b-99f2-7b1c536d0507"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's search;
- `measurements` (string): string encoding of the entry in the database containing the measurement list.

Example response:

```
HTTP/1.1 302 Found
Date: Tue, 25 Mar 2025 17:14:22 GMT
Content-Type: text/plain; charset=utf-8
Transfer-Encoding: chunked

{
"Id": "cb1e0307-3fb0-450b-99f2-7b1c536d0507",
"References":{
    "environment": {},
    "measurements": [
        {
         "value": {
            "digests":
            ["sha-256;e2Q2sMmPYjgIZtlDLCrw7gjOFqFxvaaVGuzZXuEwfWE="],
            "filename": "boot_aggregate"
            }
        },
        {
         "value": {
            "digests":
            ["sha-256;2dF3XWQ/b3ChpvZG3+AjBSdl9VihZ+xYlRrS8QE7PkY="],
```

```
                "filename": "/init"
                }
            },
            {
            "value": {
                "digests":
                ["sha-256;9fJdoyH5NBRiIF2P1uyRi7qJrV6l3J9th11RdNGTZCI="],
                "filename": "/usr/bin/sh"
                }
            },
            {
            "value": {
                "digests":
                ["sha-256;WVvNwwaZlxGlqceH5dAWGPlOuSxMCaF46HWjJPEHLYk="],
                "filename": "/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2"
                }
            }
            ...
        ]
    }
}
```

POST `/updatereference`

Update a reference measurement list, given its Whitelist ID.

Request JSON Object

- `Whitelist ID`: ID of the measurement list to be updated;
- `References` (CoMID): new measurement list.

Example request:

```
{
    "Id": "cb1e0307-3fb0-450b-99f2-7b1c536d0507",
    "References":{
        "environment": {},
        "measurements": [
            {
             "value": {
                "digests":
                ["sha-256;f3Z4sMmPYjgIZtlDLCrw7gjOFqFxvaaVGuzZXuEwfWE="],
                "filename": "boot_aggregate"
                }
            },
            {
             "value": {
                "digests":
                ["sha-256;3zH3XWQ/b3ChpvZG3+AjBSdl9VihZ+xYlRrS8QE7PkY="],
                "filename": "/init"
                }
            },
            {
            "value": {
                "digests":
```

```
            ["sha-256;9fJdoyH5NBRiIF2P1uyRi7qJrV6l3J9th11RdNGTZCI="],
            "filename": "/usr/bin/sh"
            }
        },
        {
        "value": {
            "digests":
            ["sha-256;WVvNwwaZlxGlqceH5dAWGPlOuSxMCaF46HWjJPEHLYk="],
            "filename": "/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2"
            }
        }
        ...
    ]
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's update;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 201 Created
Date: Tue, 25 Mar 2025 16:58:22 GMT
Content-Length: 93
Content-Type: text/plain; charset=utf-8

Updated reference values with the following WhitelistID:
cb1e0307-3fb0-450b-99f2-7b1c536d0507
```

POST `/deletereference`

Delete a reference measurement list, given its Whitelist ID.

Request JSON Object

- `Whitelist ID`: ID of the measurement list to be deleted.

Example request:

```
{
    "WhitelistId": "cb1e0307-3fb0-450b-99f2-7b1c536d0507"
}
```

Response JSON object

- `status` (HTTP status code): to indicate the result of the database's deletion;
- `confirmation message` (string): to inform about the successful outcome of the operation.

Example response:

```
HTTP/1.1 200 OK
Date: Tue, 25 Mar 2025 17:35:47 GMT
Content-Length: 44
Content-Type: text/plain; charset=utf-8

cb1e0307-3fb0-450b-99f2-7b1c536d0507 deleted
```

Appendix B

User's Manual

The following appendix defines the sequence of operations required for the user to configure and start the attestation framework, which consists of at least one Agent node, one Register node and one Verifier node.

B.1 Agent Node

The following instructions are used in order to properly run the Agent node. A detailed description of the configuration file's fields is also presented.

B.1.1 Install Docker Engine

All the components have been containerized, so in order to be executed `Docker Engine` [1] is needed. In the following are described the steps needed to install it in a system running Ubuntu operating system, the same followed in the testing phase.

At first the set up of Docker's apt repository is needed:

```
# Add Docker's official GPG key:
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl
$ sudo install -m 0755 -d /etc/apt/keyrings
$ sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
    /etc/apt/keyrings/ docker.asc
$ sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to Apt sources:
$ echo "deb [arch=$(dpkg --print-architecture)
    signed-by=/etc/apt/keyrings/ docker.asc]
    https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "${UBUNTU_CODENAME:-$VERSION_CODENAME}")
      stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
$ sudo apt-get update
```

Then, to install the latest version run:

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
    docker-buildx-plugin docker-compose-plugin
```

[1] Refer to the official Docker installation instructions: https://docs.docker.com/engine/install/

B.1.2 Run the Agent

After the successful installation of `Docker Engine`, the Agent can be run with one single command to be executed in the folder where both the `Dockerfile` and the `docker-compose.yml` files are present:

```
$ sudo docker compose up --build
```

B.1.3 Configuration File

Below is presented the Agent's configuration file with some examples values:

```
agentnet:
  ip: 192.168.0.103
  idproviderport: 8081
  attesterport: 8082

registernet:
  ip: 192.168.0.122
  port: 8080

waitregister:
  wait: false

keytemplates:
  ektemplate: ECCDefaultEKTemplate
  # ECCDefaultEKTemplate, RSADefaultEKTemplate
  aktemplate: ECCDefaultAKTemplate
  # RSASSADefaultAKTemplate, RSAPSSDefaultAKTemplate, ECCDefaultAKTemplate

whitelistid:
  id: dc56df5b-269c-442b-83a3-977cfacfa0dd

verifiernet:
  ip: 192.168.0.122
  port: 8081

waitverifier:
  wait: false

attestationperiod:
  seconds: 3

pcrs:
  numbers: 0, 1, 2, 3, 4, 5, 6,7, 8, 9,10, 11, 12
  # max 23 (ALWAYS INCLUDE 10)
```

Listing B.1: Agent's Configuration File

The purpose of each field is explained in the following:

- **agentnet**, used to define the IP and the ports on which the IDProvider and the Attester wait to be contacted when the Agent doesn't start the registration procedure;
- **register**, used to contact the register when the Agent starts the procedure;
- **waitregister**, needed to know if the Agent must start the registration procedure or not;
- **keytemplates**, to give control to the user on the templates to use for the EK and the AK;

- **whitelistid**, needed when the Agent starts the procedure to inform the Register about how to retrieve the golden values during the attestation phase;
- **verifiernet**, used to contact the Verifier when the Agent starts the attestation procedure;
- **waitverifier**, needed to know if the Agent must start the attestation procedure or not;
- **attestationperiod**, to set the period of the attestation that will be followed in the case the Agent starts this procedure;
- **pcrs**, to specify the number of PCRs included in the attestation object.

It is a crucial point to ensure that the Agent's configuration file is not altered. In particular, whitelist ID modifications could lead the Agent attestation to consider other golden values for its verification. To do so, the IMA policy needs to be modified to include the configuration file in the measurements as well.

B.2 Register Node

Also the Register has been containerized, so the steps needed to install `Docker Engine` described at B.1.1, are needed as well.

After the proper installation of the needed tool, the Register can be started with one single command:

```
$ sudo docker compose up --build
```

B.2.1 Configuration File

Below is presented the Register's configuration file with some examples values:

startattestation:
  port: 8081
  start: false

startregistration:
  start: false

createjwtkeys:
  create: true

firstquote:
  hashalgo: sha256

jwtkeypath:
  registerpath: register_ecdsa_private_key.pem

mongodb:
  mongouser: register
  mongopassword: papasswordord
  mongouserverifier: verifieruser
  mongopasswordverifier: password_ro
  mongoip: 172.20.0.5
  mongoport: 27017

postgresdatabase:
  dbname: attestationdb
  dbpassword: password

```
        dbuser: registeruser
        dbuserverifier: verifieruser
        dbpasswordverifier: password_ro

    postgreserver:
        ip: 172.20.0.4
        port: 5432

    registeridentity:
        build: register
        developer: thesis OU

    registernet:
        ip: 172.20.0.2
        port: 8080

    verifiernet:
        ip: 172.20.0.3
        port: 8081
```

Listing B.2: Register's Configuration File

The purpose of each field is explained in the following:

- **startattestation**, needed to contact the Verifier node on the specified port in case it has to start the attestation phase;
- **startregistration**, needed to start the registration phase from the Register side;
- **createjwtkeys**, needed to know if the Register must create the keys needed to sign the JWT created for the registration result;
- **firstquote**, used to specify which hash algorithm to consider during the verification of the first quote;
- **jwtkeypath**, used to state the path from which the keys are retrieved in order to sign the registration result, or to state where to store the keys in the case in which the previous flag is set to true;
- **mongodb**, contains several fields needed to access the MongoDB, it also has Verifier's username and password as the Register creates the Verifier's account and so it needs these info as well;
- **postgresdatabase**, needed for the same purposes as *mongodb* field but is used to access the PostgreSQL database;
- **postgreserver**, to specify the IP and the port on which the PostgreSQL database is available;
- **registeridentity**, needed by the Veraison EAT to be created properly;
- **registernet**, used to set the IP and port of the Register in the internal network;
- **verifiernet**, used to set the IP and port of the Verifier in the internal network.

B.3 Verifier Node

The Verifier has been containerized as well, so the steps needed to install `Docker Engine` described at B.1.1 are needed.

After the proper installation of the needed tool, the Verifier can be started with one single command:

```
$ sudo docker compose up --build
```

B.3.1 Configuration File

In the following, the Verifier's configuration file is shown together with some example values:

```
startattestation:
    port: 8082
    start: false

mongodb:
    mongouser: verifieruser
    mongopassword: password_ro
    mongoip: 172.20.0.5
    mongoport: 27017

postgresdatabase:
    dbname: attestationdb
    dbpassword: password_ro
    dbuser: verifieruser

postgreserver:
    ip: 172.20.0.4
    port: 5432

registernet:
    ip: 172.20.0.2
    port: 8080

verifiernet:
    ip: 172.20.0.3
    port: 8081

createjwtkeys:
    create: true

jwtkeypath:
    verifierpath: verify_ecdsa_private_key.pem

verifieridentity:
    build: verifier
    developer: thesis OU

attestationperiod:
    seconds: 5
```

Listing B.3: Verifier's Configuration File

The purpose of each field is explained in the following:

- **startattestation**, needed to contact the Verifier node on the specified port in case it has to start the attestation phase;
- **mongodb**, contains Verifier's username and password to access the MongoDB;
- **postgresdatabase**, needed for the same purposes as *mongodb* field but is used to access the PostgreSQL database;

- **postgreserver**, to specify the IP and the port on which the PostgreSQL database is available;
- **registernet**, used to set the IP and port of the Register in the internal network;
- **verifiernet**, used to set the IP and port of the Verifier in the internal network;
- **createjwtkeys**, needed to know if the Verifier must create the keys needed to sign the JWT created for the registration result;
- **jwtkeypath**, used to state the path from which the keys are retrieved in order to sign the attestation result, or to state where to store the keys in the case in which the previous flag is set to true;
- **verifieridentity**, needed by the Veraison EAT to be created properly;
- **attestationperiod**, to set the period of the attestation that will be followed in case the Verifier starts this procedure.