Grant Agreement No.: 952697
Call: H2020-SU-ICT-2018-2020
Topic: SU-ICT-02-2020
Type of action: RIA

ASSURED

D3.1: ASSURED ATTESTATION MODEL AND SPECIFICATION

Revision: v.1.0

Work package	WP 3
Task	Task 3.1
Due date	30/11/2021
Deliverable lead	UBITECH
Version	1.0
Authors	Thanassis Giannetsos (UBITECH), Dimitris Karras (UBITECH)
Reviewers	Richard Mitev (TUDA) Benjamin Larsen, Edlira Dushku (DTU)
Abstract	D3.1 discuss issues related to modelling and reasoning about trust requirements and assumptions in the ASSURED ecosystem. More specifically, it formally defines, by using predicates and axioms, the trust model that needs to be achieved by the designed ASSURED security (attestation) enablers and secure data sharing and management schemes towards enhancing the overall security posture of the target supply chain environment.
Keywords	Attestation, Verification, Trust, Authorization, Predicates, Axioms

Document Revision History

Version	Date	Description of change	List of contributors
v0.1	15.08.2021	ToC	Thanassis Giannetsos (UBITECH)
v0.2	10.09.2021	State-of-the-art analysis of various trust modelling languages and decision on the language to be leveraged in ASSURED (Chapter 2)	Edlira Dushku, Benjamin Larsen (DTU) Thanassis Giannetsos (UBITECH) Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA) Richard Mitev, Philip Rieger (TUDA) Liqun Chen, Nada El Kassem (SURREY)
v0.3	30.09.2021	Description of the first draft of the trust models for all of the ASSURED attestation enablers to be designed in D3.2 (Chapter 4)	Edlira Dushku, Heini Bergsson Debes, Benjamin Larsen (DTU) Richard Mitev, Philip Rieger (TUDA) Liqun Chen, Nada El Kassem (SURREY) Thanassis Giannetsos (UBITECH) Ilias Aliferis (UNIS) Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA)
v0.4	15.10.2021	Description of the first draft of the Trusted Computing Base to be adopted in ASSURED as well as initial draft of the Tracer to TPM-based Wallet communication (Chapter 3)	Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA) Richard Mitev, Philip Rieger, Marco Chilese (TUDA) Liqun Chen, Nada El Kassem (SURREY) Dimitris Papamartzivanos (UBITECH)
v0.5	29.10.2021	Description of the first draft of the trust models for the secure data sharing and management services of ASSURED (Chapter 4)	Edlira Dushku, Heini Bergsson Debes(DTU) Ilias Aliferis (UNIS) Richard Mitev, Philip Rieger, David Koisser (TUDA) Thanassis Giannetsos (UBITECH)
v0.6	12.11.2021	Update and finalization of all ASSURED trust models (Chapter 4)	Benjamin Larsen (DTU) Thanassis Giannetsos, Dimitris Karras (UBITECH) Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA) Richard Mitev, Philip Rieger, Marco Chilese, David Koisser (TUDA) Liqun Chen, Nada El Kassem (SURREY)
v0.7	10.12.2021	State-of-the-art analysis of the software- and hardware-based roots-of-trust based on the features and security models (Chapter 5)	Edlira Dushku, Heini Bergsson Debes, Benjamin Larsen (DTU) Richard Mitev, Philip Rieger, Marco Chilesse, David Koisser (TUDA) Ilias Aliferis (UNISYSTEMS)
v0.8	14.01.2022	Re-design of the protocol for establishing a secure and authentic communication channel between the Tracer and the TPM-based Wallet (Chapter 3)	Thanassis Giannetsos (UBITECH), Richard Mitev, Philip Rieger (TUDA) Meni Orenbach (MLNX/NVIDIA) Benjamin Larsen (DTU)
v0.9	24.02.2022	Review the document	Richard Mitev (TUDA) Benjamin Larsen, Edlira Dushku (DTU)
v1.0	27.02.2022	Finalisation of the document	Thanassis Giannetsos, Dimitris Karras (UBITECH)

Funded by the Horizon 2020
Framework Programme of the European Union

Editors

Thanassis Giannetsos (UBITECH), Dimitris Karras (UBITECH)

Contributors (ordered according to beneficiary numbers)

Edlira Dushku, Heini Bergsson Debes, Benjamin Larsen, Nicola Dragoni (DTU)

Richard Mitev, Philip Rieger, Jingru Wang, Marco Chilese, David Koisser (TUDA)

Liqun Chen, Nada El Kassem (SURREY)

Ahmad Atali, Meni Onrebach (MLNX)

Dimitrs Papamartzivanos, Thanassis Giannetsos, Dimitris Karras (UBITECH)

Ilias Aliferis (UNIS)

DISCLAIMER

Funded by the Horizon 2020
Framework Programme of the European Union

COPYRIGHT NOTICE

Project co-funded by the European Commission in the H2020 Programme		
Nature of the deliverable:		**R**
Dissemination Level		
PU	Public, fully open, e.g. web	✓
CL	Classified, information as referred to in Commission Decision 2001/844/EC	
CO	Confidential to ASSURED project and Commission Services	

* R: Document, report (excluding the periodic and final reports)

DEM: Demonstrator, pilot, prototype, plan designs

DEC: Websites, patents filing, press & media actions, videos, etc.

OTHER: Software, technical diagram, etc.

Funded by the Horizon 2020
Framework Programme of the European Union

Executive Summary

In the last years, academia and industry working groups have made substantial efforts towards realizing next-generation smart-connectivity "*Systems-of-Systems*". These systems have evolved from **local, standalone systems into safe and secure solutions distributed over the continuum from cyber-physical end devices, to edge servers and cloud facilities**. The core pillar in such ecosystems is the establishment of **trust-aware service graph chains**, comprising both resource-constrained devices, running at the edge, but also container-based technologies. The primary mechanism, employed in ASSURED, to establish trust is by leveraging the concept of trusted computing, which addresses the need for veriable evidence about a system and the integrity of its trusted computing base and, to this end, related specifications provide the foundational concepts such as **measured boot, sealed storage, platform authentication and remote attestation**.

An essential component in building such trusted computing services is the concrete definition of the appropriate trust models capturing all complex relationships (that need to be established between devices), assumptions and requirements that need to be met by the ASSURED offered security services. In this context, D3.1 introduces all ASSURED **trust Models by means of a modelling language capable of capturing assumptions and properties of all ASSURED functionalities**: remote attestation, dynamic real-time risk assessment and enhanced and accountable knowledge sharing of operational (threat) intelligence data flows (through the use of policy-compliant Blockchain structures). This is achieved by employing a combination of **predicate- and algebra-based** modelling languages that enables us to define strong models split into three components: (i) the predicates which are essentially the "words" of the language, (ii) the axioms which define how these predicates fit together to produce meaningful trust statements, and (iii) the assumptions which are the predicates that are of the scope of the ASSURED solution. Through these definitions, D3.1 can then formally outline the security and trust assumptions behind the remote attestation techniques, such as memory safety, type safety, control flow safety and operational correctness, as well as the confidentiality, integrity and privacy requirements towards the secure and authentic participation of a device and the secure data sharing and management.

Therefore, when we refer to providing verifiable evidence for the correctness of a device, this can be achieved by fulfilling the security predicates when the axioms we put forth for the specific attestation mechanisms hold. With these given models, the ASSURED solution should be able to *assess, monitor and verify* the trust level of a network of devices.

After having defined the trust models, capturing the complex device relationships in the envisioned application domains, that need to be achieved by the ASSURED framework using a Trusted Computing Base (TCB), we then proceed with a state-of-the-art analysis on the types of root-of-trust that we can consider based on the functional components they cover, the functionalities they offer (in terms of assurance claims throughout the entire lifecycle of a device) and any existing *security models* for their offered features. The endmost goal is to present the merits and challenges of both software- and hardware-based trusted computing technologies which led to the ASSURED decision on using the benefits of both worlds; hence, the adoption of a **TCB based on the use of a Trusted Execution Environment with the TPM as the hardware-based root-of-trust**. This design choice enables the concept of **Zero Trust security principle**, with the need of "*Never Trust, Always Verify*", for which ASSURED bootstraps vertical trust for all devices and users in the target SoS by enabling **continuous attestation, authorization and authentication** prior to be allowed to communicate and/or be granted to data or resources. This type of TCB allows the flexibility of been able to **guarantee the correctness of the tracing features**

of ASSURED (through the TEE), as the cornerstone of the entire framework correctness, and the correct **(self-) issuance of cryptographic material and verifiable credentials through the use of a TPM as the building block of the ASSURED TPM-based Wallet**.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scope and Purpose

In the context of this deliverable, the focus is to define the **trust models** that are able to capture the complex relationships between all involved entities and components of the systems operating within the ASSURED framework. These models should be able to capture the **assumptions** that should be made, as well as the **requirements** that should be fulfilled by each of the security services provided by ASSURED.

We aim to define these models in a semi-mathematical way, by employing **model languages** in order to express the system requirements in the form of **predicates** and **axioms**. The criteria for the selection of these languages will also be outlined, so that we can formally express the system requirements with the desired level of granularity. Overall, our purpose in this deliverable is twofold:

- **To model the ecosystem of the entire System-of-Systems** considered in ASSURED. This essentially creates the trust model for our entire world, since we consider that we can guarantee the required level of trustworthiness for any asset, service or operation that is located **within the boundaries of the defined model**, by performing attestation of both the **edge devices** and the **blockchain**. Conversely, we cannot guarantee the required levels of trustworthiness for anything outside the trust model.
- To provide the formal definition of the security requirements so that, when we design the schemes that will be employed by the ASSURED security services, we can then use this definition in order to perform a **formal security analysis** for how these requirements can be achieved.

Taking into consideration all the above, the scope of D3.1, includes the refinement of our research methodology needed for the design of the run-time remote attestation enablers employed by the security services provided by ASSURED, which will also be described in further detail in D3.2 [29]. To this end, in this deliverable we will focus on the description of the requirements of all **attestation methodologies and their related services**, such as **secure enrollment**, **key management**, **key revocation**, and **downloading of smart contracts**, with verifiable evidence on the integrity assurance and correctness of the device.

Also, the scope of this deliverable includes the description of the methodologies that will be used in order to protect the capabilities and the integrity of remote devices by turning them into **Trusted Platforms**, through the functionalities that the underlying Root-of-Trust can provide to the devices

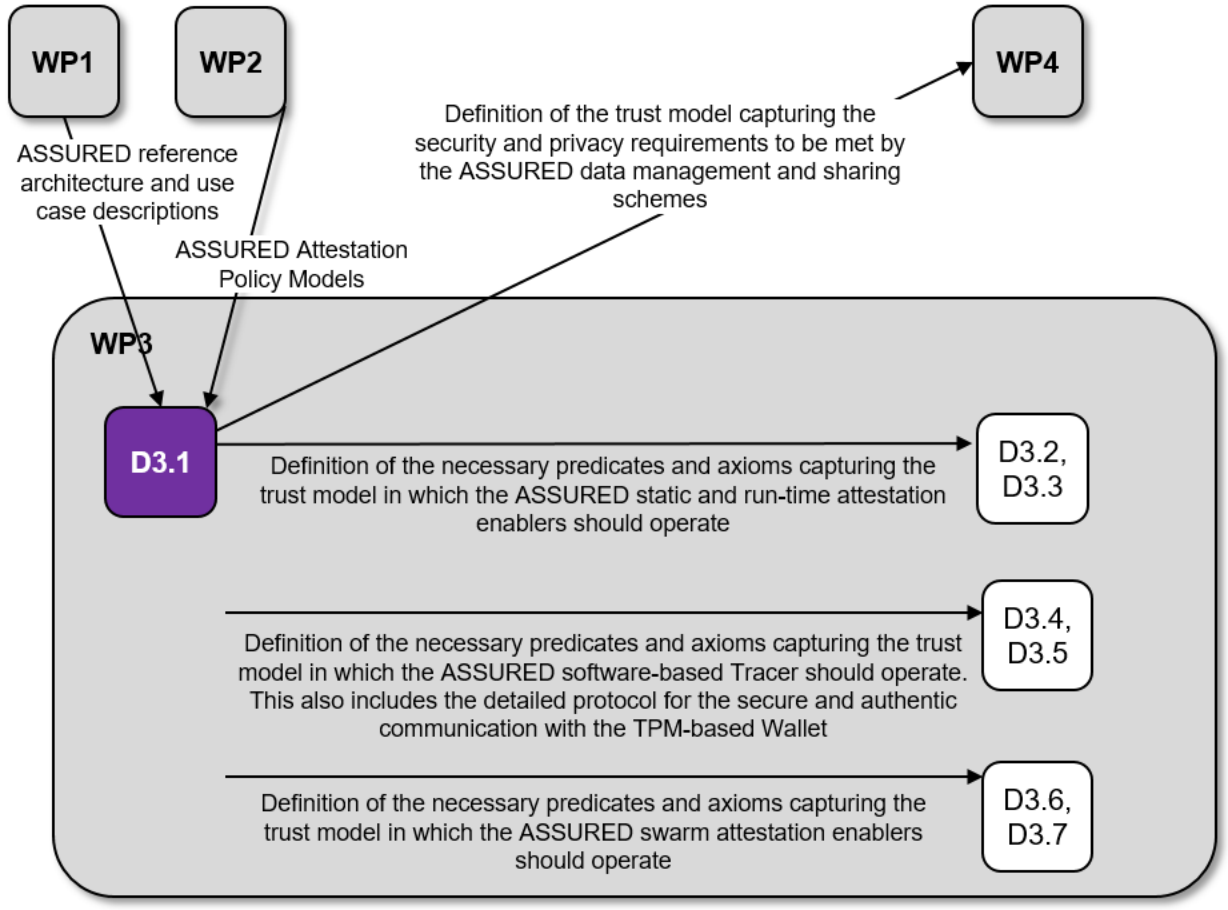

Figure 1.1: Relation of D3.4 with other WPs and Deliverables

and their OSs. Note that the instantiation in ASSURED employs **Trusted Platform Modules (TPMs)**, but the presented methodologies can be applied regardless of what trusted component is employed by each asset. **Advanced key management** will then be analyzed through predicates and axioms that apply to the different types of keys that are required for a device to be enrolled and to operate in the ASSURED ecosystem.

1.2 Relation to other WPs and Deliverables

In what follows, Figure 1.1 depicts the relationship of the deliverable with other Work Packages (WPs) as well as the other tasks in the same WP(3). As aforementioned, the main purpose of this document is to consolidate the main trust model capturing all of the assumptions and requirements that need to be met by the ASSURED offered services; starting the core attestation enablers and secure communication to the secure data management and sharing through the ASSURED Blockchain infrastructure. Therefore, D3.1 constitutes the reference point that will also drive the design of all ASSURED security schemes (Attestation Enablers (T3.2 & t3.5) and Tracer (T3.4)) and secure on- and off-chain interactions in the context of WP4. Furthermore, the well-defined trust models will also set the scene for the formal security analysis to be conducted in the context of T3.4 for all aforementioned security anchors and trust extensions. Hence, the outcome of Deliverable D3.1 is intended to support the definition of all core crypto primitives and security protocols of the project.

1.3 Deliverable Structure

This deliverable is structured as follows: **Chapter 2** describes the desired behavior of all of the security services provided by ASSURED, the design principles that should be followed and the assumptions that should be made in the design of the ASSURED platform, as well as the security properties and requirements. **Chapter 3** is focused on the description of the trust modelling framework, including the selection of the policy languages that will be used in the formal description of the model, and the contents and functionalities of the underlying TPM trusted component. **Chapter 4** describes the trust assumptions, as well as the security and safety requirements that need to be met in order to establish strong guarantees of trustworthiness in a supply chain ecosystem. We also present the formal trust models for each of the security services and attestation methodologies implemented in ASSURED. After having defined the trust models, capturing the complex device relationships in the envisioned application domains, that need to be achieved by the ASSURED framework using a Trusted Computing Base (TCB), **Chapter 5** proceeds with a state-of-the-art analysis on the types of root-of-trust that we can consider based on the functional components they cover, the functionalities they offer (in terms of assurance claims throughout the entire lifecycle of a device) and any existing security models for their offered features. The endmost goal is to present the merits and challenges of both software- and hardware-based trusted computing technologies which led to the ASSURED decision on using the benefits of both worlds; hence, the adoption of a TCB based on the use of a Trusted Execution Environment with the Trusted Platform Module (TPM) as the hardware-based root-of-trust. Then, based on the selection of the TPM as the core trusted component, **Chapter 6** discusses some challenges that we will face in the modelling the security, privacy and trustworthiness requirements that need to be met by the ASSURED framework and the need to be resolved at an early stage in the project. Finally, **Chapter 7** concludes the deliverable.

Chapter 2

ASSURED Security, Privacy and Trust Establishment Services

2.1 ASSURED Model for Security, Privacy and Trust Assessment & Establishment

One of the main goals of ASSURED, regarding the supply chain ecosystem, is to provide **robust security and trust**, while simultaneously employing strong **privacy protection**. In other words, ASSURED aims to construct a model that takes into consideration the dynamically changing security relationships and interdependencies between the assets comprising the System-of-Systems under consideration. To this end, we need to define the **security services** that need to be provided by ASSURED in order to fulfill the security, trust and privacy requirements, as well as the high-level **design principles** and **trust assumptions** for these services.

There are significant complexities and technological challenges that need to be overcome in order to achieve these objectives. These complexities arise from the fact that, in the context of ASSURED, we do not consider individual devices, but instead we need to define methodologies about how the security claims made about one device can be interpreted, when the device is part of a **service graph** representing a **complex System-of-Systems (SoS)**. In D1.3 [32] and D2.2 [33], we defined the kind of security relationships that can be identified between the assets comprising the SoS as **isolation**, **interaction** and **representation** relationships. How we can capture these relationships that need to be established between the components that run within a device or between different devices, and have interactions in the context of security services with different criticality in their safety requirements. It follows that the presence of this kind of security relationships increases the complexity of the whole system.

Therefore, the challenge ahead is to represent all these complexities in the trust models that will be defined. These complexities extend to the achievement of the requirements in the design of the ASSURED platform, such as setting up a number of network protocols that operate in tandem in order to create a composite security mechanism that complies with the underlying privacy protection requirements. To address the issue of complexity, we introduce a modelling language and define a formal model of trust. Next, based on **security predicates**, we can describe and refine **security axioms** that describe the desired system behavior.

In this context, we first need to describe the desired behaviour of devices in the ASSURED ecosystem. To this end, we will list and document the type of security and privacy-preserving services that are offered by ASSURED and will be taken into consideration when creating the

trust model in Chapter 4. The design of this model will be made by defining **security predicates**, based on which we can define and refine **security axioms** that describe the desired behavior. In this context, we need to describe the desired behavior of the devices in the ASSURED ecosystem. The envisioned core services to be protected, based on what was also described in D1.2 [30] and will be further analyzed in Chapter 4, are given in Table 2.1. Further information regarding the mechanisms employed in the context of these services is provided in Section 4.3, where the key management methodologies are analyzed.

Table 2.1: ASSURED core services to be protected

Security service	**Description**
Data confidentiality of operational data	It should be ensured that the data exchanged during communication between devices is carried out in a privacy preserving manner with verifiable guarantees.
Data integrity and verification of the attestation data	The execution of attestation processes requires the secure sharing of attestation data between the Prover and the Verifier. Information is relayed via the Security Context Broker (SCB) for certifiable recording on the ledger, and authorized and accountable data sharing.
Privacy protection of the exchanged operational and/or attestation data based on the type of sensitive information included	Depending on the operational needs, different levels of privacy configuration should be supported based on the operational needs of the application. Also, attestation should be able to be carried out in a privacy preserving manner, so that the Prover does not need to reveal information about its identity, its configuration and execution details.
Data confidentiality of accompanying system/attestation raw data	The control flow data or the configuration traces that are used in their respective attestation algorithms should not be disclosed, in the interest of retaining the privacy of the Prover.
Release of attestation-related information, to stakeholders, with different levels of access and information granularity	ASSURED implements an attribute-based encryption scheme, where it is possible to encrypt data depending on the attributes of the party that is intended to decrypt the data. Thus, it is possible to disclose data to different parties with different levels of granularity, based on the security and privacy requirements.
Secure design of a new device to be securely onboarded to the overall environment	In order to securely enroll and register a device to the system, we need to verify the correctness of its state prior to its enrollment, as well as be able to certify the correct design, meaning that the correct components are loaded, and the device ID and credentials are certified (Section 4.7).

As a continuation, a number of attestation services has been implemented in ASSURED, in order to perform privacy-preserving attestation of devices, in a manner that suits the needs and requirement of each application scenario. These attestation services are listed in Table 2.2.

Table 2.2: Attestation services in ASSURED

Security service	Description
Zero Touch Configuration Integrity Verification	Remote attestation to the correct configuration state of a device, by comparing the device state to a stored state that is already known to be trustworthy (Section 4.2.1).
Swarm Attestation	Simultaneous attestation of a group of devices where the Verifier issues a single challenge, while maintaining the privacy of the target devices, in a way that is more efficient than attesting each device separately (Section 4.2.2).
Run-Time Verification	Attestation executed throughout the operational lifecycle of the system, where the control flow of a device is checked dynamically during run-time, so that it is verified that all control flows and information transfers are permitted by the target device (Section 4.2.3).
Direct Anonymous Attestation (DAA) of Devices	A decentralized trust assurance scheme, where the responsibility for verification is offloaded to the edge devices, while simultaneously maintaining the privacy of the device to be attested (Section 4.2.4).
Jury-Based Attestation	In cases where a dispute arises regarding the correctness of a particular attestation, a second line of defense is implemented, where a group of devices, referred to as the Jury, makes a decision on whether another device is trustworthy or not (Section 4.2.5).

These services need to be described while also elaborating on examples of scenarios that will be modeled in the context of the envisioned use cases as examples. The usage of these services in the context of specific scenarios of the envisioned use cases, especially when it comes to secure data management, will be mapped and fleshed out in the context of D4.2 [36].

2.2 High Level Design Principles for the ASSURED Middleware Security

This section presents the fundamental security design principles, trust assumptions, and attestation requirements that form the basis for creating the proposed models' predicates and axioms described in the next chapter. These design choices are the foundation for the secure interactions between deployed devices and the policy-compliant Blockchain infrastructure. These security principles are orthogonal to the trust models presented in Chapter 4.

2.2.1 Design Principles and Trust Assumptions

Based on the above, we specify the design principles and trust assumptions that have to be made in regards to the ASSURED platform. These will provide the basis towards the design and implementation of the security methodologies and trust models, which will be presented in Chapter 4.

1. The ASSURED attestation protocol resides inside a Trusted Computing Base (TCB) component, consisting of a set of hardware, firmware and software components that guarantee the correct hardware isolation. Specifically, we assume that edge devices in ASSURED are equipped with a TCB which consists of a Trusted Platform Module (TPM), TrustZone, Firmware, Tracer, and TPM Wallet.
2. The TCB software must be secure against a defined adversary model.
3. TPMs shall follow Trusted Computing Group's (TCG) specification for TPM 2.0 [9].
4. In ASSURED, a runtime tracer that detects the unintended behavior of a running device is part of the TCB.
5. Device's cryptographic keys are protected by the TPM. The keys are securely stored and maintained by the TPM and can be accessed only by the TPM. The only exception is an *immutable* key in the tracer.
6. ASSURED must provide a link between the TPM attestation key and the TPM endorsement key to verify the provenance of the attestation report.
7. ASSURED must guarantee a secure and authentic communication channel between the TPM Wallet and the Privacy CA for verifying the TPM validity.
8. ASSURED must ensure Attestation Key Protection, allowing a Certification Authority (CA) to verify that a private key is protected by a TPM and the TPM is trusted by the CA.
9. ASSURED should guarantee DAA privacy through the integration of Zero-Knowledge proofs.
10. ASSURED must provide key uniqueness using a single key only for one purpose (e.g., key establishment, generation of digital signatures, etc.), as presented in more details in Section 4.4.
11. ASSURED should guarantee the correctness of the Verifier. In particular, when there is a discrepancy between the Prover's attestation response and Verifier's report to the Blockchain, ASSURED should perform Jury-based Attestation to verify the correctness of the Verifier (more details provided in Section 4.2.5).
12. ASSURED must provide a secure enrollment process of the devices when a valid attestation report of a device is generated by a genuine TPM.
13. ASSURED must apply DAA to provide a privacy-preserving enrollment process of the devices.
14. ASSURED must securely revoke the device's credentials in the case of a failed attestation event (more details presented in Section 4.5).
15. The attestation of a group of devices (i.e., swarm attestation) in ASSURED must be effective and efficient, as presented in the Section 4.2.2, Axiom 2 and Axiom 6 in Table 4.8.

2.2.2 Security Properties Requirements

In line with the state-of-the-art remote attestation protocols aimed at secure interactions, the security services provided by ASSURED, including attestation, should have the properties listed in Table 2.3. Specifically for the attestation services, the properties are listed in Table 2.4. These properties set the scene for the trust models that will be presented in Chapter 4.

Table 2.3: Generic requirements for security properties in ASSURED

Property	Description
Integrity	The attestation protocol should provide reliable evidence guaranteeing that the attestation measurement corresponds to the Prover's memory at the time of the attestation request.
Authenticity	The protocol should ensure that the attestation evidence originates from the entity the Verifier is interested in. In swarm attestation, the authenticity can be somehow 'indirect,' e.g., the Verifier can intend to attest a group of devices without checking directly the identity of each device participating in the swarm.
Correctness	The attestation should represent the actual state of the device, and the attacker should not be able to evade detection.
Completeness	All relevant parts of a system must be attested. Attesting just a portion of an entity does not yield useful information, e.g., a device cannot be trusted as a whole just because *one* application is correct.
Atomicity	The attestation procedure must fully happen or not happen at all. Thus, the attestation does not get interrupted and must not complete partially.
Freshness	The attestation should ensure that any given response to an attestation request can be reliably linked to that request.

Table 2.4: Attestation specific requirements for security properties in ASSURED

Property	**Description**
Device Configuration Correctness	We should be able to verify that the configuration of a device is as expected.
Service Graph Chain Trustworthiness	We need to be able to securely run the swarm attestation process on a group of devices.
Attestation Key Protection	To retain trust in a device, despite mutable configurations, we need to be able to create secure and certified AKs.
Immutability	The measurement process should remain unchanged over time.
Liveness and Controlled Invocation	Attestation reports need to be produced within a specific time frame.
No-replay attestation results	Attestation results should not be reproducible by third parties.
Revocation of failed to attest devices	It should be possible to revoke access of a device to the framework after a failed attestation.
Correctness of Verifier	We should be able to check that the Verifier is in a correct and expected state.
Operational Correctness	We should be able to verify that the devices are in a correct operational state.
DAA Key Protection	Keys used in the DAA attestation scheme should be protected in terms of security and privacy preservation.
Link between cryptographic keys	Endorsement Key and Blockchain keys given to the device should be linked by the ASSURED SCB. Also, Endorsement Keys should be linked to Attestation Keys.
Key uniqueness	Keys are only permitted to be used for one specific purpose and are unique.
Completeness of attestation report	Attestation reports contain sufficient information to enable the Verifier to detect malicious state changes at the Prover's side.
Privacy-preserving (zero touch) attestation report	The devices to be attested should not need to disclose details regarding their configuration and state. Attestation should be performed without the Verifier knowing the trusted reference value itself that needs to be measured by the Prover.
Soundness and correctness of Zero knowledge proofs for DAA signatures	The proofs regarding DAA signatures should be verifiably correct.
Correctness of the credential verification	The verification of credentials should be performed in a trustworthy manner.
Secure communication channels	Secure and authentic communication channels between Privacy CA/ ASSURED authority and the TPM should exist.
Correct hardware isolation	Software ouside the TCB cannot tamper with software in the TCB, e.g., with TrustZone.
Correctness of the software TCB	All software in the TCB does not contain bugs/vulnerabiltiies that can be exploited by an attacker

Chapter 3

Trust Models

In order to provide the ASSURED project's envisioned services with the appropriate levels of ***security*, *privacy* and *operational assurance***, we need to define trust models that are able to capture the complex relationships between all involved entities and components. **This model must not only capture the ASSURED ecosystem and the applications (and use cases) that rely on it, but also the cloud-based environment in which they operate, which may involve an arbitrary number of untrusted third-party entities and assets**, as described in D1.2 [30]. In this context, we employ the principles of **Zero Trust Architecture** [59]. Specifically, we consider that **no resource is inherently trusted**, i.e., there is no implicit trust granted to devices and assets based on their physical location, network positioning, or ownership. Therefore, the trustworthiness of the system is established by establishing the trust relationships between the assets. To this end, we need to define the **assumptions (security predicates)** that need to be valid as prerequisites, the **security claims** that should be made, and the **Roots of Trust** that should be contained in the assets comprising the target system, considering a decentralized architecture for trust establishment. These aspects should be captured by the trust models defined within ASSURED, in order to capture the **full stack of the trust architecture**.

In order to achieve the above requirements, we leverage several modelling languages and techniques in combination, to capture assumptions and models of the comprising fog nodes equipped with the employed Trusted Platform Modules (TPMs), as Roots-of-Trust, and their interaction with the surrounding entities. *This combination allows us to express fine-grained trust assumptions in a "top-down" manner—starting from the description of the application's trust domain, and iteratively refining it to model internal interactions between the entities involved, and the specific operations performed by each device)*. Also, note that in the context of Zero Trust Architecture, we need to consider the interactions between the **TPM-based Roots-of-Trust** and the **host devices** which are initially considered untrusted, as well as the **interactions between the devices**, in order to eventually establish a holistic trust model for the entire system.

From the above, it follows that the goal is to be able to extract and formally define the **security properties** that need to be achieved by all the **security enablers and trust extensions** (i.e., *integrity verification, direct anonymous attestation, anonymous authentication ephemeral communication channel, etc.*), to be designed in the context of ASSURED, towards fulfilling the main vision of the project: **The provision of a secure overlay mesh network for delivering the high-level functionalities related to secure (edge and mesh) device identification and integrity, data integrity and confidentiality, anonymity and resource integrity.**

In particular, the trust models we define here can be refined into security (attestation) policies to be enforced by the ASSURED SCB, ensuring that all nodes perform all expected operations at all levels. More details of this refinement process, including strategies and policies for the

enforcement and monitoring of trust in a TC-backed system, is given in D2.2 [33] and D2.3 [35]. In this context, the focus is on the **Integrity Verification** and on the **secure enrolment** of native system components. Integrity verification is the process by which a device can report in a trusted way the current status of its configuration, at any requested time. It entails the provision of integrity measurements and guarantees during both the **deployment and operation of a device**; covering the system integrity at the deployment phase by the ASSURED SCB, but also ensuring the integrity of the loaded components during their run-time execution.

Furthermore, these trust models will set the basis for the formal verification of these ASSURED security services and events interaction. The vision is to define a set of run-time verification approaches (leveraging tools like ProVerif and/or TAMARIN) for monitoring and verifying both the execution behaviour of a single system as well as the communication patterns of a set of edge devices against the set of already defined attestation services and policies. To this end, the ASSURED consortium has presented a methodology in [43], that enables the use of the aforementioned tools towards verifying complex protocols by using trusted computing, and has presented an instantiation of this approach for a TPM-based remote attestation service.

Towards this direction, a set of abstract formal models capturing these specifications of the ASSURED security enablers is presented in Chapter 4, which can then be checked against the defined formal security properties. **ASSURED security properties including Zero-Touch Configuration and Secure Enrollment, Secure Remote Asset Management, Key Management, Operational Assurance** - on top, of course, of the generic properties such as **confidentiality, authentication, integrity and availability** should be expressed, and the formal framework needs to be rich enough to express properties to the domain under consideration.

Taking into consideration all the above, we will first focus on defining and justifying the trust modelling languages that will be employed, so that the requirements of ASSURED as mentioned above are fulfilled, and the establishment of trust within the Zero-Trust Architecture is enabled. Next, we will define the Trusted Computing Base employed in ASSURED, and we will establish the trust modelling framework. Note that, in this Chapter, we focus on the definition of the trust model **within a single device**, considering the interactions between the **TPM-based Root-of-Trust (trusted world)** and the **host device (untrusted world)**. The establishment of the model of trust between multiple devices will be the focus of Chapter 4.

3.1 Trust Modelling Languages

Trust modelling languages are used to formally define the level of trust of each entity in the operational environment. By this definition, these languages can focus on different aspects of the overall system execution in an attempt to better express the properties of interest to be achieved. Based on the requirements of the use cases envisioned in the context of ASSURED, several languages have been identified that could be used to formally define the desired notion of trust in TPM equipped fog/edge devices.

- **Predicate Based Language [57]:** This language is based on a predicative system of mathematical logic. First, an informal problem statement is created, which will define the exact requirements (i.e., trust assumptions) needed to patch any known problems so that a device can be effectively identified as trusted. Then a set of specific predicates is constructed with each one of them mapping to a previously defined requirement. Based on these, a set of axioms is derived from the informal problem statement, that practically glues together the predicates in a way that best represents the "*chain of trust*" that needs to be ensured. This

way, the problem of identifying the trustworthiness of an entity is modelled and quantified based only on the properties that are of interest in the specific use cases (that, in turn, leverage specific TPM functionalities); thus, allowing for a more fine-grained expression of the trust policies to be deployed.

- **Diagram Based [23, 53]:** Diagram-based models are used to represent trust relationships between interacting entities. Usually their purpose is to investigate how trust is propagated throughout a network or a hierarchy of entities by assessing the trust levels of each component and by identifying the types of strong trust relations (federations) that need to be established among the different entities in the system. This enables the representation of a "Web of Trust", by leveraging tree-like structures, that needs to be established and continuously monitored and can serve as the basis for concluding on stronger arguments about the system's design-level trustworthiness.
- **Algebra Based [7, 44]:** Algebra-based languages are often used in conjunction with the previously described diagram-based models. Once again they focus on trust relationships, that need to be established between interacting entities (rather than the trust modelling of the execution of a deployed platform), following a more formal modelling approach with varying levels of detail. These languages aside from static trust relationship predictions, they also include protocols for dynamically updating the quantitative values used for expressing the federated trust, among entities, based on experience factors.

In ASSURED, we mainly employ a combination of **Predicate Based** and **Algebra Based** languages. Specifically, Predicate Based languages are used in order to model the assumptions and conditions that have to be valid in order to ensure that the system is in a trusted state, and the corresponding axioms that should hold in order to establish trust relationships between interacting entities are represented by using Algebra based languages. The goal is to provide a flexible representation, that can capture the level of granularity required by the heterogeneous systems considered within ASSURED.

3.2 A Trust Modelling Framework for ASSURED

In the context of ASSURED, the goal is to observe, model, and monitor not only the trust level of each edge device but also the strong trust relations that must be established among interacting entities. This requires the consideration of different aspects in each case; for instance, trusting a device first requires trusting that it operates correctly, and in particular, that sequences of software function commands are executed correctly while ensuring that the interactions between attested entities, that is required in order to maintain the trust between them, is secure.

As previously mentioned, the best approach is to use a combination of appropriate modeling languages; **namely, the Predicate-based and Algebra-based languages**. The motivation behind this design choice is that predicates can better serve the modeling of the trusted execution of ASSURED-equipped devices (at the device level), **based on both behavioral properties and low-level concrete properties about the entities' configuration and execution** (to be verified by the ASSURED attestation services), while algebra-based models can enable a formal expression, in the form of axioms, of the conditions that need to be established and maintained in order to establish system trustworthiness in fog-based ecosystems.

Overall, **the goal is to describe the chain of trusted interactions that need to take place between entities in order to ensure the trustworthiness of the assets and the entire systems, based on the necessary and sufficient identified predicates and axioms.**

The models we describe in Chapter 4 will only provide a high-level representation of the networks of trust that need to be established as part of the security models for our applications. For instance, our models contain abstract states, such as "*Device Integrity*" or "*Physical Security*", that express high-level assumptions on the attestation state of a device that must be fulfilled before making use of specific ASSURED security functionalities. In order to construct complete application-specific models, such assumptions will need to be refined, using additional predicates that can specify, for instance, the types of properties that should be attested. We will investigate these low-level predicates and express them in detail in the context of D2.3 as part of the formal modeling of the ASSURED Attestation Toolkit and its internal attestation policies.

The remainder of this deliverable follows this top-down approach. First, we identify the common security and safety requirements we aim to achieve in ASSURED with limited trust. Then, we define a semi-formal and high-level trust model in which to describe the security and safety of devices and networks, using predicates and state diagrams. Finally, we give concrete representation, in this model, of the ASSURED security and trust extensions. These are expressed as a combination of predicates and axioms.

3.3 Trusted Computing Base (TCB) of ASSURED

Each device in ASSURED consists of various modules, each one of which belongs to either the **secure world** or the **insecure world**. The modules that belong to the former category are considered to be trusted by default. One such module is the **Tracer**, that monitors the behavior of the device in order to provide the measurements required for the execution of the attestation algorithms employed in ASSURED, and is considered to be trusted by default. Also, devices are equipped with a **TPM**, which serves as the Root-of-trust.

The Trusted Computing Base (TCB) refers to a collection of hardware, firmware and software components that belong to the **secure world**. These modules should be resistant to any attacks specified in the adversarial model, and are considered to be trusted by default. The TCB is required in the context of attestation for implementing primitives that provide security guarantees, and is critical to the general security of the entire system [45]. In this section, we describe the TCB of the edge devices in ASSURED, as depicted in Figure 3.1. The TCB components are marked in gray. Next, we describe the TCB components in detail.

Trusted Platform Module (TPM): Besides standard hardware components like processors, system buses, memory, and peripherals, we assume the existence of a TPM that acts as the system's **Root-of-Trust**. The TPM is considered trusted by default, because it contains a unique, unforgeable, verifiable device identifier installed during manufacturing. Furthermore, we assume that the TPM provides trustworthy security engines that provide correct and inherently trusted output, such as a Random Number Generator, cryptographic hash-functions, encryption functions, and signature algorithms. The core services provided by ASSURED, such as attestation and Blockchain wallets, are implemented by using the TPM as a **Trust Anchor**.

TrustZone: We assume that the edge devices provide a **Trusted Execution Environment (TEE)**, that enables a hardware-assisted isolated execution environment. Specifically, we employ TrustZone [4] as the underlying TEE technology, which provides two security domains: the secure and insecure world. Each part has dedicated memory regions with a separate software

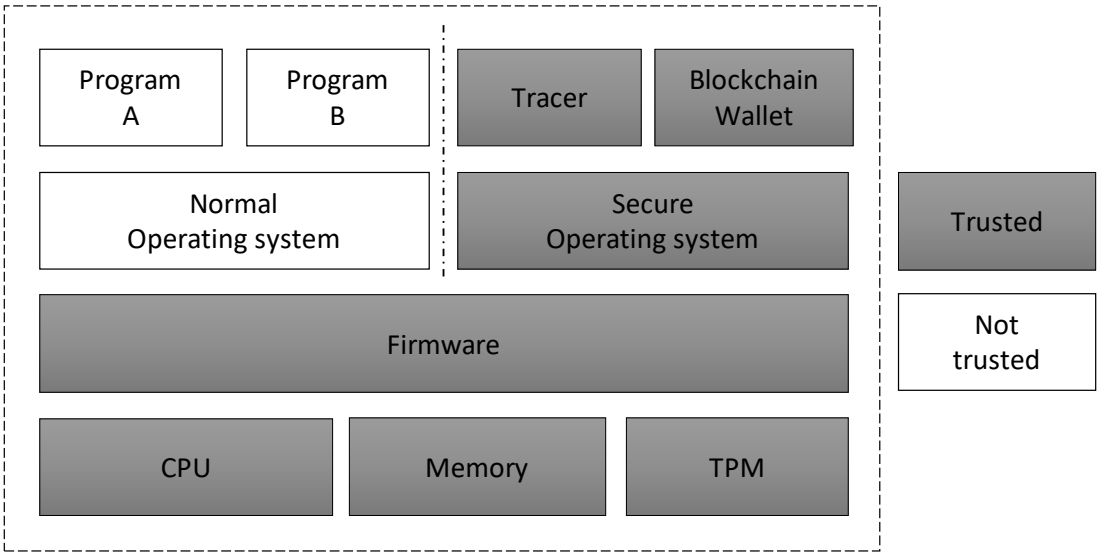

Figure 3.1: Illustration of the ASSURED TCB

stack, including a dedicated operating system (OS). The secure world OS manages its own memory and schedules the tasks that are executed in its context. Note that context switching between the secure and the insecure world can only occur through the trusted firmware, which is also referred to as the *security monitor*. It is important to note that increasing the size of the TCB also increases the complexity and computational overhead for attestation operation. To this end, we aim to minimize the use of the secure world, by assuming that the TPM Software Stack is running in the insecure world.

Firmware: The firmware is responsible for booting the edge devices, and we consider the different stages of the bootloader to be trusted and part of the TCB. Furthermore, we assume that each boot stage in the firmware measures and validates the next boot stage and extends the loaded images hashes into the TPM to form the basis for an attestation report on the system's loaded software, including the secure world OS. The firmware implements the **security monitor**, which provides a secure context switch between the secure and insecure world, and prevents leaking the internal states of the secure world to the insecure world.

Real-time Monitoring Tracer: The Tracer is responsible for continuously monitoring the processes executed in the device it belongs to, and collects information that is required in the context of the attestation methods employed in ASSURED. This information can include control flow graphs used in **Control-Flow Attestation (CFA)**, and hashes of configuration properties used in **Configuration Integrity Verification (CIV)**. This information is signed by the Tracer in the secure world, and is subsequently passed to the Attestation Agents to perform the required operations. The Tracer is executed as a user space program, and is verified before being loaded to the device by measuring and verifying the deployed binaries.

TPM-based Blockchain Wallet: The TPM-based Blockchain wallet is a core component of ASSURED, since it is responsible for the secure management of all the crpytographic keys required for the interaction between devices in the context of attestation, as well as the interaction between devices and the Blockchain. Specifically, it provides a trusted execution environment that enables secure and isolated execution of code, and enables the protection of the verifiable credentials and attributes that are required by devices for securely interacting with the Blockchain infrastructure by using hardware-based keys. This module has been defined in D4.1 [28] and extensively analyzed in D4.5 [37].

3.3.1 Communication between the TCB Building Blocks

Communication between the Tracer and the TPM needs to go through the TPM Software Stack (TSS), which specifies the software layer for using functions provided by the TPM. One instantiation of the TSS, that will be employed by ASSURED, is the IBM TPM 2.0 TSS [5]. However, recall that both the Tracer and the TPM belong to the secure world. If the TSS is running in the insecure world, then it is possible for a malicious party to manipulate and compromise the interaction. Therefore, in order to retain the trustworthiness of the TCB, we need to answer the question: *How do we establish secure and trusted communication between the Tracer and the TPM?*

In order to address this issue, we have the following options:

1. The first option is to **place the entire TSS in the secure world** of the TrustZone TEE, which was mentioned in the previous section. This would ensure that the communication between the Tracer and the TPM is trusted, therefore solving the problem, but it would have a huge impact on the system performance, because assigning additional functionality to the TrustZone creates a very large additional computational overhead.

2. The second option is to keep the TSS in the insecure world, but to implement **additional protection mechanisms, in the form of policies**, regarding who can interact with the TPM. This approach is much more computationally efficient, but the implemented policies should be constructed, so that the issue of communication through the insecure world is properly addressed.

In ASSURED, we select the second option, in order to maintain a high level of performance. This approach poses the first hurdle that we need to overcome regarding the formulation of the trust model: Since the TPM needs to sign measurements that have been provided by the Tracer, and the communication needs to go through the insecure world, *how do we make sure that the traces used in the attestation protocols originate from an authentic Tracer that resides in the device?* In the following, we outline the methods employed by ASSURED in order to address this issue.

3.3.1.1 TCB Equipped with a Pre-installed Key

In order to prove the validity of the measurements that the TPM receives from the Tracer and uses in the context of the implemented attestation protocols, we employ a **Pre-Installed Key**. This is a unique, secret cryptographic key, which is embedded in the fuses of the device during the manufacturing process, and is only accessible to software executing in the secure world domain. This key will be used to send signed traces to the Verifier, who is responsible for verifying their integrity.

Note that is is possible for this key to be used by the Tracer in order to sign the traces on behalf of the TPM. Specifically, is is possible for the Tracer to sign the measured traces by using the Pre-Installed Key, since the key is only accessible to the secure world, and thus cannot be forged by the adversary. Thus, it is possible to use this solution as a standalone application, since it is not required to send the signed traces back to the TPM. Therefore, it is possible to remove the TPM to increase the performance of the system and the computational power required by the CPU. We trust that this key is never leaked and is only readable by the TEE, and it provides guarantees that the traces signed with this key originate from an authentic Tracer residing in the device.

3.3.1.2 Locality-based Protection

A solution employed by ASSURED in order to guarantee the correct signing of a trace that has been knowingly in transit through an untrusted environment is the use of **locality-protected signing**. This method can address the aforementioned issue of the trustworthiness of the measured traces. Specifically, the purpose of this approach is twofold:

1. To ensure that **the trace wasn't compromised during transit**. To this end, the TEE should execute the sign command in the TPM with a command audit session. Thus, the TPM can attest to the TEE which commands were executed, and what parameters were used in their execution. This information proves to the TEE that the correct signing operation was executed, and can enable the TEE to detect whether the signing operation was executed with incorrect parameters.

2. To ensure that **the trace originates from a trusted source**. To guarantee that only the TEE can execute signing operations on the TPM, we employ the use of Localities, which are essentially an indicator designed to show where a command originates from. The enforcement of rules and regulations involving Localities is the responsibility of the platform hardware.

Next, we provide further information regarding the use of Localities. In Table 3.1, we provide a standard use cas of localities on a PC platform. Specifically, Locality 0 is the default setting, and indicates that the command originates from the Static OS. In general, the Locality of a command is determined by which memory area of the TPM it should write to, thus we can configure a command to write in a non-default Locality. Then, these Localities can be used in order to determine security policies. For example, in a standard chipset, the Southbridge of the platform can be configured to disallow Locality 4 commands by filtering them. Note that it is permitted to issue commands of any locality, but they will not be executed if they are disallowed by an enforced policy. Locality protection is a feature supported by the TPM, but is enforced by the platform in this example, according to the PC Client Specification [71].

Locality	Example Usage
0	Static OS
1	Dynamic OS
2	Dynamic OS Runtime
3	Auxiliary Components
4	D-RTM, CPU MC

Table 3.1: Standard use case of localities on PC Platform

The requirement for Locality based filtering to work is the presence of a platform that enforces the Locality, and can differentiate whether a command comes from a secure- or insecure world. This removes any requirement for extra signing, and since we can only use the key from one Locality, we have guaranteed that it comes from a valid origin. It does not, however, protect the integrity of the command itself. To achieve this, we must verify that the trace was correctly signed.

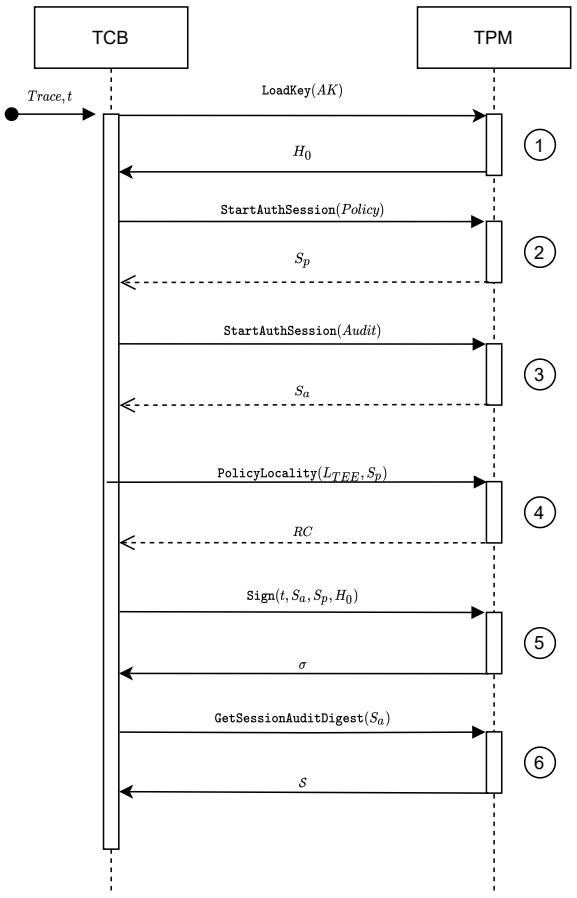

Figure 3.2: Locality Based Protection with Audit

We can use these Localities by protecting our signing key with the `TPM2_PolicyLocality` command and stating that a command needs to be provided in a Locality that represents a secure origin. An example of the protocol is shown in figure 3.2. 1) The TCB asks the TPM to load the AK, which is then decrypted by the TPM, and a handle is provided to the TCB. Then a Policy Session (2) is started, and after that, an audit session (3). Now the TCB tries to satisfy the policy of AK by proving the Locality (4). If this is successful, the return value has no error. Now the TCB can ask the TPM to sign the trace t, using the key referenced by H_0, in the policy session S_p that satisfies AK's policy. Furthermore, it passes the audit session S_a to the command (5). Next, the command and the command's parameters are extended into the audit digest behind the scenes, and if the policy digest in S_p matches that of AK's, it will return a signature to the TCB. To verify that the TPM signed the correct trace, the TCB asks for the SessionAuditDigest, which contains an extended hash of the command and command parameters executed - TCB can then compare this with a reference hash.

3.3.1.3 Traces Integrity & Authentication

This section presents a protocol securing the integrity of the reported traces, despite the presence of an adversary between the Tracer and TPM. This protocol protects against replay attacks and impersonation attacks, and ensures the integrity of the traces during correct protocol execution. A simple solution to these challenges can be to construct an Attestation Key in the TPM and use the pre-installed key in the Tracer. The Tracer would then sign the traces (with a nonce) and ask the TPM to sign them as well. Here, an adversary in the TSS cannot manipulate a trace, nor

can it present itself as a false Tracer, as he does not possess the private key of the Tracer. In a broader sense, in case an adversary gains access to the TSS, then she can use the AK to sign ad hoc digests. We need to address this issue by providing guarantees that an adversary can never gain access to the AK or manipulate the data being signed. We achieve this by constructing an Attestation Key that can only be used to sign traces that have been authenticated by the Tracer. This can be achieved thanks to the Extended (or Enhanced) Authorization feature provided by TPM 2.0. We expand on this functionalidy and provide further details in Section 4.3.2. By using this, we show that this protocol provides protection against both integrity, replay- and impersonation-attacks.

Overview: Next, we provide a step-by-step description of the trace integrity ad authentication protocol. We present this as a sequence diagram in figure 3.3.

First, the Tracer defines a policy for the Attestation Key. This policy requires a signed authorized command and parameter digest from a secondary key: the Tracer's secret key. The technical details are described in point (1) of the list of formalized operations presented below. When the policy digest is t, the Tracer sends the TPM Create Primary command to the TPM with the policy as a parameter. This command instructs the TPM to create a key-pair guarded by the policy, which then sends back a key handle, $K_{h_{TPM}}$.

The policy calls for an Authorization Digest, which includes both the command to be executed, the parameters, and *the name* of the entity concerning the command. The name of an entity is essentially a hash of its public parameters, and can be found in the returned K_{TPM} which holds the public parameters (including the policy, cryptographic public key, etc.), and the *name*.

Before signing the traces N, the Tracer must satisfy the policy. To do this, it starts a new policy session, which instantiates a new session digest inside the TPM, and returns the session identifier (s_p). The Tracer then calculates the trace-digest H_N by hashing all the traces. It can now create the Authorization Digest, H_a, representing the expected signing command and the parameters (N), as defined in item (2) of the formalized process list below. The Tracer signs this with its private key sk_T which generates the signature σ_{H_a}. Recall the policy of the Attestation Key. If this key is used, there must be a signed authorization from the Tracer for this particular action. This signed Authorization Digest provides a guarantee that it is impossible for an adversary in the TSS to manipulate the signing operation.

As the policy requires the TPM to verify the signature, the Tracer must load its public key inside the TPM. It does so using `TPM2_LoadExternal`, which makes the TPM load the key into memory, and return a key handle to it. The Tracer is now ready to satisfy the policy. It does so by executing `PolicySigned`. This command requires the key handle to the loaded Tracer key, the signature of the Authorization Digest σ_{H_a}, and the $Params$. The latter is data used to calculate the digest and needed for the TPM to verify it (2). The TPM computes a reference value to H_a by executing the same hashing operation as the host in order to calculate $aHash$ (2). It will then extend the session digest with the command code and the name of the signing key, if and only if it can verify the Authorization Digest is correct and signed by the provided key. If the *correct key* was used to sign the Authorization Digest, then the session digest should now match the policy digest of the TPM key.

As the Tracer has provided a Command Parameter Hash (CpHash), the TPM will copy this to internal memory and set a flag. This flag limits the next command execution to the command (and parameters) represented by the CpHash. If the next executed command has a different command code or the parameters to the command are different, the TPM will reject it.

Finally, the Authorization Signature, the TPM Signature, the Traces, and the TPM Public Key Data are sent to the Verifier, who then verifies a) the policy, b) the authorization signature, c) the signature over the traces.
Next, we provide a mathematical formalization of the processes that were mentioned in the flow of the authentication algorithm:

- **DefinePolicy** (1): Defines the policy to be verified by using `TPM2_PolicySigned`. Policy P is defined as $P := H(S \mid\mid CC \mid\mid pk_{T_{name}})$ where H is the chosen hashing function, S is the previous policy-state digest (zeroes), CC is the command code of `TPM2_PolicySigned` and lastly $pk_{T_{names}}$ is the name of the Tracers public key who is required to provide a signature (essentially a hash of the public parts).
- **ComputeAuthorizationDigest** (2):

  The authorization digest consists of a nested hash. The primary hash is defined below.

  $$aHash := H(Nounce_{TPM} \mid\mid Exp \mid\mid cpHashA \mid\mid PolicyRef)$$

  We denote the parameters for this Authorization Hash as $Params$.

  - $Nounce_{TPM}$: Session Nounce from the session identified by S_p
  - Exp: Expiration of authorization (none for this use case)
  - $PolicyRef$: An ad-hoc reference for special identification, none in this usecase.
  - $cpHashA$: The Command Parameters Hash for the authorized command and data. The Authorized Command Parameter hash is defined as the following.

    $$cpHashA := H(CC \mid\mid K_{T_{name}} \mid\mid H(M)))$$

    Where CC is the command code for `TPM2_Sign`. This ensures only this command can be executed upon authorization of the Tracer.
- **ComputeCpHash** (3): Does the same calculation as in (2) (the part where the CpHash is calculated), but using the input parameters given to the TPM.

Security: There are three adversarial scenarios that we analyze, where the adversaries have control over the TSS in each of these scenarios. These scenarios correspond to 1) Replay attacks, 2) Tracer Impersonation, and 3) Trace-integrity.

Replay attacks: Assuming that an adversary has captured any packages during the process, a replay attack will not be possible, because every time `PolicySigned` is executed, it requires the Session Nonce. The attempt at a replay attack would be detected when the TPM calculates the reference value of the authorization digest.

Trace-integrity: If the TSS changes the Trace values that have been transmitted, this change would be detected during the verification phase, since the signatures would not match the provided traces.

Tracer Impersonation: If the TSS acts as a Tracer, even going through the process of creating his own TPM key, impersonation would not be possible. As the adversary cannot access the Tracer's private key, it would be unable to provide a valid signature over the Authorization Digest.

To summarize the above scenarios, impersonation is prevented by the security of the Tracer's private key, data integrity is protected by the security that Tracer provides for the traced data, and replay attacks are prevented by the use of Session Nonces.

Figure 3.3: The TPM-Tracer Communication Protocol

Chapter 4

Trust Assumptions and Security & Safety Requirements

A number of requirements need to be met in order to establish and maintain strong guarantees of trust in a supply chain ecosystem, both in the context of the envisioned use cases and for trusted computing-based applications in general. Recall that one of the main objectives of ASSURED is the dynamic establishment and management of **trust-aware service graph chains**. In this context, *the communication over the continuum from edge devices to Blockchain nodes and backend cloud systems must support secure interactions between all participating entities in order to establish* ***service-specific communities of trust.***

For instance, as described in Section 3.3, a common requirement is that each device in the network must be equipped with hardware or software support for remote attestation. Attestation is one of the crucial services of a TPM (Chapter 5). It is the process by which a platform reports in a trusted way the current status of its configuration and/or execution state. The report can include as much information as required. The basis of the attestation are the measurements recorded in PCRs. They can then be read to know the current status of platform and be also signed to provide a secure report. The signed message can then be sent to the client. It is worth noticing that the TPM does not check the measurements, that is, it does not know whether a measurement is trustworthy or not. The trustworthiness of the measured value comes when an application uses some PCR value in an authorization policy, or remote clients ask for an attestation of some value, and later they evaluate its trustworthiness. Attestation enables such clients to confirm whether the platform has been compromised. Additionally, the TPM offers means of certifying and auditing the properties of keys and data that cross the TPM boundary.

This requirement serves to establish a decentralized root-of-trust, which cannot be compromised without physical access to the device, and on which the attestation process will both measure the device's configuration and communicate those results securely and privately to other devices in the system. Furthermore, we require that the device is resistant to non-invasive attacks (such as side-channel attacks, or non-invasive fault injection) while the system should be able to identify nodes that have been offline for a long time and could be victims of more invasive exploitation attempts [65] (such as micro-probing or reverse engineering).

Besides the traditional **data confidentiality, integrity and availability**, trusted supply-chain ecosystems must fulfill the following security and trust requirements:

Memory-Safety. Memory safety is a crucial and desirable property for any device loaded with various software components. Its absence may lead to software bugs but most importantly

exploitable vulnerabilities that will reduce the trust level of the device running the problematic software. In a nutshell, all accesses performed by loaded processes/services in the underlying memory map of the host device need to be "*correct*" in the sense that they respect the: (i) logical separation of program and data memory spaces, (ii) array boundaries of any data structures (thus, not allowing software-based attacks exploiting possible buffer overflows), and (iii) don't access the memory region of another running process that they should not have access to. Memory-safety vulnerabilities can be detected in design-time with static code analysis techniques [13, 19] and during run-time [60] with the well known tool Valgrind [56] that is designed to identify memory leaks of an executable binary. For instance, memory safety will prevent information from leaking in a security sensitive application that uses a TPM.

Type-Safety. Type-safety is closely related to memory safety as it also specifies a functionality that restricts how memory addresses are accessed in order to protect against common vulnerabilities that try to exploit shared data spaces (i.e., stack, heap, etc.). Type-safety is usually checked during design-time with most programming languages providing some degree of correctness (by default) paired with static code analysis tools that might catch some exceptions not covered by the language compiler (i.e., "fuzzing" tools or concolic execution engines). However, type-safety can also be checked during run-time with the possibility of identifying issues that the static method did not identify [18]

Control-Flow Safety. Besides the aforementioned static methodologies that check the code and the binaries for possible vulnerabilities and bugs, the executable binaries should be dynamically checked for their proper functionality and execution during run-time. This is done through control-flow attestation: All control transfers are envisioned by the allowed program. This translates to no arbitrary jumps in the code, no calls to random library routines, etc. This information is depicted by the allowed control-flow graphs (CFGs) that are calculated prior to the deployment of a service and are used as a baseline of the normal (trusted) sequence of execution states against which run-time control-flow footprints will be assessed [6, 49]. The basic attribute that is required here is that the binary monitoring entity (tracer), the attesting entity (prover) and the checking entity (verifier) are all operating in a trusted mode and are in a state to correctly identify bad behaviour from the monitored binary.

Operational-Correctness. This concept is an intermediate abstraction of control-flow safety. Besides integrating the control-flow mechanism that was described before, it also checks for the static state of the system and relies on the fact that a crucial part of the underlying kernel is in a trusted state. The operational-correctness aims to provide a more holistic view of the system by combining dynamic and static data collected by the ASSURED Attestation Toolkit in order to produce guarantees on the operational trust state of the system.

Cryptography. Having strong cryptographic primitives is a fundamental requirement of any security oriented system. What is needed towards this direction is a good source of entropy that will be utilized in a secure pseudo-random number generator (PRNG) so that the keys generated by the system are secure. To make good use of this source of entropy, we also must ensure that the cryptographic primitives deployed in the ASSURED-equipped platforms and related systems are fit for purpose. Although in most cases, the security of cryptographic primitives is a matter of design, the system's cryptographically secure pseudo-random generator, which is used in particular to generate keys, is often left to implementers, with po-

tentially disastrous consequences on the security of the whole system [55]. In the context of the ASSURED, we assume security against strong adversaries.

Physical Security. TPMs are discrete hardware chips that interconnect with the Low Pin Count (LPC) bus of the system through. The LPC interface can be subject to attacks from eavesdropping to injection. Many recent and old attacks [1–3, 74] have shown that through the LPC interface, an attacker can spoof PCR values and steal sensitive data (like the BitLocker disk encryption key), bypassing critical TPM trust guarantees. That is why the physical security of both the device as a whole and the actual pins that connect the TPM on the device motherboard should be carefully designed if the TPM is to be trusted. Another option that has been investigated by [11, 47], is to constantly require each device to provide a "heartbeat" attestation in a specific frequency. Because hardware attacks are time consuming and require that the device is taken offline for a considerable amount of time, the TPM will not be able to provide the attestation messages timely. In this case, the device should be considered as untrusted or partially trusted depending on the policies and the trust requirements that are in place.

4.1 Formal Trust Model

Next, we aim to present and describe the trust models that each one of the core services, offered by ASSURED, needs to adhere to if the technical security and privacy requirements, defined in D1.1 [31] are to be met. Recall that the main vision of ASSURED is to enhance the cyber-health of next-generation smart connectivity "Systems-of-Systems" towards the evolution of such safety-critical SoS from local, stand-alone systems into safe and secure solutions distributed over the continuum from cyber-physical end devices, to edge servers and cloud facilities (comprising the entire supply chain of mixed-criticality services) with the endmost goal to enable reliable, secure and privacy-preserving extraction and sharing of knowledge (originating from verified and authenticated data sources) and threat intelligence information. These requirements are grouped by type of the core functionalities and services envisioned in ASSURED in the following fields: **remote attestation, dynamic real-time risk assessment and enhanced and accountable knowledge sharing of operational (threat) intelligence data flows (through the use of policy-compliant Blockchain structures)**. Recall that the core ASSURED services to be protected are the ones depicted in Table 2.1.

Of course, the common denominator in all the services is the execution of the appropriate attestation enablers, for producing the necessary security claims on the trustworthiness level of each device, as well as the correct issuance and management of the cryptographic material and credentials towards the secure and authentic participation in the overall service graph chain. For each one of the designed static and run-time attestation mechanisms [29], we formalize the trust requirements that need to be fulfilled in order to provide verifiable evidence towards ensuring the trustworthiness of the entire system. Recall the categories of trust modelling languages that were presented in Section 3.1, where it was mentioned that ASSURED will employ a combination of **predicate-** and **algebra-based** languages in order to depict the requirements that need to be depicted in the context of the trust modelling scheme. To this end, for each of the employed schemes we will define the following:

1. **Security Predicates**, that depict the trust assumptions in a format that pairs a formally defined word with its meaning in terms of security requirements.

2. **Axioms**, that depict the trust and security claims, that should be present for ASSURED to guarantee that the attestation requirements are fulfilled.

Therefore, when we refer to providing verifiable evidence for the correctness of a device, this can be achieved by fulfilling the security predicates when the axioms we put forth for the specific attestation mechanisms hold. Based on the use of such flexible predicate-based languages, the trust models presented in the following sections are split into three components: (i) the predicates which are essentially the "words" of the language, (ii) the axioms which define how these predicates fit together to produce meaningful trust statements, and (iii) the assumptions which are the predicates that are of the scope of the ASSURED solution.

The predicates are meant to be the dictionary of the trust model that enlists each statement as a word and pairs it with its meaning. We prompt to split them into multiple categories depending on the layer that each model pertains to: either for a device, the underlying Trusted Platform Module (TPM) or a network of devices. The first one (Table 4.1) aims to give a set of predicates that represent the final (and trusted) stages of the overall system and network (trusted domain). These will be satisfied only when the system and the TPM-equipped devices are trusted; if one precondition of these predicates fails then the system will be in an untrusted state and no guarantees on the security posture can be verified. There are also other layers (per ASSURED service) that represent intermediate states of the system and the TPM-equipped devices. These intermediate states are meant to group together the requirements in separate categories based on the previously described security and safety requirements. For instance, in the context of runtime Control-flow Attestation (Section 4.2.3), the $AttestationKey_{TPM}(T, P)$ predicate translates to the fact the Attestation Key (AK) of the Prover device needs to have been created correctly during its registration and enrollment to the overall system which in turn fits under the generic $CryptoSafe_{TPM}(T)$ system predicate for verifying the presence and validity of the underlying TPM to create the AK based on the key usage policies provided to the device during registration (Section 4.7). Finally, we use separate categories to capture such low-level predicates on the configuration and execution properties of the system that need to be considered during attestation (such as firmware running, the version of its configuration file or presence of specific hardware properties, ports and network interfaces, etc. as defined in D1.3 [32]). It is separated from the other groups as it aims to cover local and remote identification of a TPM with the target of modelling the trust of specific TPM functionalities considered in the envisioned use cases.

Finally, we have to highlight that the security of the BIOS/Kernel of the system is considered as a prerequisite as part of the TCB. If the kernel is approached as a monolithic system, then it should be assumed that it is trusted in its whole since if even a single component diverges then the entire kernel is deemed untrusted. On the other hand, the kernel in the emerging edge- and cloud-computing applications where everything is considered as a service, can also be seen as a set of micro-services where only a specific set of them should be considered trusted in order for the entire system to be at a correct state. This reduction of the trusted code base of the kernel can introduce a chance for the tracing capabilities of the ASSURED [34] to monitor exactly those functionalities.

Besides these considerations, we will also assume that the TPM itself satisfies memory-safety, type-safety and control-flow safety. In other words: hardware TPMs are correct, secure and tamper-resistant. With these given assumptions, the ASSURED solution should be able to *assess, monitor and verify* the trust level of a network of devices based on the following models.

Note that these models will set the scene for the formal security analysis of all attestation and cryptographic schemes to be developed.

4.2 Static & Run-time Attestation

While each attestation scheme employed by ASSURED is characterized by its own particular set of security predicates, we first need to define a set of predicates that apply to all attestation enablers. These refer to properties that should be valid for the TCB, namely the **TPM** and the **Tracer**. Specifically, the underlying TPM should be secure from a physical standpoint, so that it is not accessible by a malicious party. Also, its memory should be inaccessible by a potential attacker, and the cryptographic primitives employed should be considered secure. It is also assumed that the TPM can execute any command correctly. Regarding the Tracer, it should be designed to act in a trustworthy manner. Also, the **Privacy Certificate Authority (CA)** responsible for verifying the validity and correctness of any TPM, as a root-of-trust, is considered inherently trustworthy, as well as the **domain of operation** of the TPM-enabled devices.

We also define some high-level axioms regarding the TPM, so that it can be considered trusted if and only if all the aforementioned predicates regarding the TPM are valid. These axioms are formally defined in Table 4.1 and are common for all of the core ASSURED services described in the following sections. The formal expression of this set of high-level predicates is given in Table 4.2. In the following, we consider that these predicates and the TPM global axiom apply to each of the considered attestation methods, in addition to the predicates and axioms specified.

Table 4.1: High-level Predicates for the ASSURED Ecosystem

Predicate	Predicate Meaning
$\mathbf{PhySecure}_{TPM}(T)$	TPM T is physically secure
$\mathbf{CryptoSafe}_{TPM}(T)$	TPM T uses secure cryptographic primitives
$\mathbf{MemorySafe}_{TPM}(T)$	TPM T has memory safety
$\mathbf{Trusted}_{Tracer}(R)$	Tracer R is acting in a trustworthy manner
$\mathbf{Trusted}_{Issuer}(I)$	The Privacy CA I is acting in a trustworthy manner.
$\mathbf{CorrectExecution}(C,T)$	TPM T will correctly execute command C
$\mathbf{TrustedDomain}(D)$	The domain D composed of a set of TPM-enabled devices is trusted

Based on these predicates we can define an axiom required for all TPM operations.

TPM Trust Axiom
(Ax1) $\forall c_i, t_i \quad \mathbf{PhySecure}_{TPM}(t_i) \wedge \mathbf{CryptoSafe}_{TPM}(t_i) \wedge \mathbf{PhySecure}_{TPM}(t_i) \wedge \mathbf{MemorySafe}_{TPM}(t_i) \wedge \mathbf{CorrectExecution}(t_i, c_i) \Leftrightarrow \mathbf{Trusted}_{TPM}(t_i)$
A TPM t_i is trusted if, and only if it will correctly execute valid commands, it is physically secure, has memory safety and crypto safety.

Table 4.2: Global Axiom for TPM

4.2.1 Zero-Touch Configuration Integrity Verification of Devices

Zero Touch Configuration Integrity Verification (CIV) focuses on the attestation of the correct configuration state of a device. Specifically, this methodology aims to verify that the list of loaded binaries on the target device is correct, when compared to a configuration state that is already known to be trustworthy. It is also considered that the list of loaded binaries does not change during runtime, unless a change is initiated by the administrator of the system through a secure software update.

In the context of Configuration Integrity Verification (CIV), we assume the following. Each host has one honest tracer and one honest TPM. We assume that each host's TPM knows the public part of an asymmetric key k, whose secret part (its inverse) is securely hidden as part of the smart contract concerning attestation policies, and that each host's tracer can communicate authentic traces (digests) to the smart contract. We assume that the inherent immutability of smart contracts guarantees the smart contract's honesty and that the smart contract knows the public part of an endorsement key that is secured on each host's TPM. The remaining host environment is untrusted, e.g., the file system. The aforementioned predicates are listed in Table 4.3.

Table 4.3: Relevant CIV predicates

Predicate	Predicate Meaning
$\mathbf{SmartContract}_{ap}$	The smart contract knows a
$\mathbf{Tracer}(h, e)$	The honest tracer on host h knows e
$\mathbf{Inv}(k)$	The inverse (i.e., secret part) of asymmetric key k
$\mathbf{Property}(a, b)$	b is an ingredient of a
$\mathbf{TPM}(h, a)$	The honest TPM on host h knows a
$\mathbf{Trusted}(e)$	e is trusted or in a trusted state
$h(e)$	h knows e
$\mathbf{Authentic}(e)$	e is authentic (e.g., a digest generated from a keyed hash function using a shared key)
$\mathbf{DeviceStateMeasured}(h)$	Host h device state has been securely measured

The CIV protocol relies on the measurements of the configuration state of a device, which are stored in the corresponding Platform Configuration Registers (PCRs) in the TPM, and the subsequent verification of the traced measurements against a known and trusted device state. Therefore, the defined axioms in the context of this attestation protocol refer to establishing trust that the corrrect PCRs have been updated in a trusted manner. Also, it should be asserted that the attestation key used to sign these measurements is trustworthy, the produced attestation keys can be managed in a trustworthy manner, and that the signing with this key can prove that the device configuration is in a trustworthy state. These axioms are formalized in Table 4.4, by using the predicates that were defined in Table 4.3.

4.2.2 Swarm Attestation of Devices

In a system that consists of multiple interconnected devices and assets, it can be inefficient in terms of computational complexity to attest each device individually. To this end, ASSURED proposes the use of a **Swarm Attestation** scheme, which aims to attest all the target devices with a single challenge in an efficient manner, while simultaneously safeguarding the location privacy of

Table 4.4: Axioms for configuration integrity verification

Axioms
(Ax1) $\forall k, ek, pol, t, T, ak, h : \mathbf{Trusted}_{TPM}(T) \wedge \mathbf{SmartContract}_{ap}(\{\mathbf{Inv}(k), ek, pol, t, \{ak\}_{\mathbf{Inv}(ek)}\}) \wedge \mathbf{TPM}(h, \{\mathbf{Inv}(ek), \mathbf{Inv}(ak)\}) \wedge \mathbf{Property}(pol, \{\mathsf{CC}_{\mathsf{PolicyAuthorize}}, k\}) \wedge \mathbf{Property}(ak, \{t, pol\}) \Leftrightarrow \mathbf{Trusted}(ak)$
An attestation key ak is trusted for host h if h can present a certificate signed using the private part of h's TPM endorsement key ek. This certificate must attest to the fact that ak was created with key template t, and correct policy pol, extracted from the smart contract and chosen by the Privacy CA. In this case, pol gives the owner of the private part of the asymmetric key k the ability to sign arbitrary policies that must be satisfied before h can use the private part of ak for TPM signing operations.
(Ax2) $\forall d, h : \mathbf{Tracer}(h, d) \Leftrightarrow \mathbf{Tracer}(h, \mathbf{Authentic}(d))$
The trusted tracer on host h can arbitrarily authenticate digests d.
(Ax3) $\forall ek, pcr, d, d_{\mathsf{audit}}, h, T : \mathbf{Trusted}_{TPM}(T) \wedge \mathbf{SmartContract}_{ap}(\{ek, pcr, d, \{d_{\mathsf{audit}}\}_{\mathbf{Inv}(ek)}\}) \wedge \mathbf{TPM}(h, \{\mathbf{Inv}(ek), d_{\mathsf{audit}}\}) \wedge \big(\mathbf{Property}(d_{\mathsf{audit}}, \{\mathsf{CC}_{\mathsf{NV\_Extend}}, pcr, \mathsf{RC}_{\mathsf{success}}, \mathbf{Authentic}(d)\}) \vee \mathbf{Property}(d_{\mathsf{audit}}, \{\mathsf{CC}_{\mathsf{PCR\_Extend}}, pcr, \mathsf{RC}_{\mathsf{success}}, \mathbf{Authentic}(d)\})\big) \Leftrightarrow \mathsf{DeviceStateMeasured}(h)$
A host h is trusted to have updated the correct PCR pcr with digest d if h can present the expected TPM-generated audit digest d_{audit} proving that its trusted TPM executed the PCR update command successfully with pcr and d as arguments, where d_{audit} is signed using the private part of h's endorsement key ek which is secured in h's trusted TPM.
(Ax4) $n, h_1, h_2, ak_{pub} : \mathbf{Trusted}(ak) \wedge h_1(\{ak, n, \{n\}_{\mathbf{Inv}(ak)}\}) \wedge \mathbf{TPM}(h_2, \mathbf{Inv}(ak)) \Leftrightarrow \mathbf{Trusted}(h_2)$
A host h_1, who knows the public part of host h_2's trusted attestation key ak_{pub}, can verify that h_2 is in a correct state by challenging h_2 to sign a fresh nonce n (chosen by h_1) using its ak_{priv}. If h_2 manages to provide a signature over n using its ak, then h_1 knows that h_2 has managed to satisfy a policy as authorized by the administrator of ak's policy, thus proving–in zero-knowledge–that h_2 is in a conformant state.

the users, by anonymizing all data coming from the deployed edge devices. Therefore, it follows that the high-level predicates, given in Table 4.6, should capture the underlying privacy requirements, by defining trusted and privacy-preserving communication between swarm members, as well as between a swarm member and the Verifier.

In addition, we define intermediate predicates in Table 4.7, which refer to the control-flow and code integrity, as well as the time required to perform attestation processes, which will be used in order to define the corresponding axioms.

In Table 4.5, we provide a set of notations which are required for the expression of the Swarm Attestation predicates and axioms. A Swarm protocol consists of a tuple ($setup$, $request$, $attest$, $report$, $verify$) where:

- $setup_{(Vrf,S)}$: algorithm executed by Vrf and all S devices in order to deploy S into the

application domain. It includes establishment of network connectivity, topology, secure provisioning of cryptographic material to all group members $m_i \in S$ and Vrf.

- $request_{(Vrf,S)}$: algorithm initiated by Vrf to request an authenticated measurement from a subset of S members. Every m_i receives an attestation challenge ch from Vrf.
- $attest_{m_i}$: integrity-ensuring function implemented and executed individually by each m_i in response to request. This function produces an attestation result h_{m_i}.
- $report_{(S,Vrf)}(h_{m_1}...h_{m_N})$: algorithm executed between S members to aggregate individual h_i results into the swarm attestation result H_S, which is then delivered to Vrf.
- $verify_{Vrf}(H_S, VS)$: algorithm executed by Vrf after completion of report. It checks whether H_S is within a set of Valid States (VS).

Table 4.5: Swarm Notation Summary

Term	Description
$\mathcal{V}$rf	Verifier
S	Set of devices involved in the Swarm protocol
N	Number of devices in S
VS	Set of valid software states of S
m_i	Member i of S
ch	Challenge used to initiate $attest$
T_{att}	Time taken to compute $attest$
T_{agg}	Time taken to aggregate attestation
h_{m_i}	Attestation result of m_i
H_S	Attestation result of the swarm

Table 4.6: High-level Predicates for Swarm attestation

Predicate	Predicate Meaning
$\mathbf{Trusted}_{Comm}(m_i, m_j)$	The communication between Swarm members m_i and m_j is trusted
$\mathbf{Trusted}_{Comm}(m_i, Vrf)$	The communication between m_i and Vrf is trusted
$\mathbf{Privacy}(m_i, m_j)$	Swarm members m_i and m_j authenticate themselves in a privacy-protecting way
$\mathbf{Privacy}(m_i, Vrf)$	The Vrf verifies member m_i while providing privacy-preserving guarantees

Next, based on the aforementioned predicates, in Table 4.8 we define the axioms that should hold for the design and application of the Swarm Attestation scheme. These axioms define the following aspects: i) The conditions that need to be valid in order to achieve the **trustworthiness** of all the devices in the swarm and their **integrity in terms of code and control-flow**, ii) the fulfillment of the **privacy requirements** for the devices and users, and iii) the **efficiency** of the swarm attestation scheme, meaning that it should be more efficient than attesting each device individually.

Table 4.7: Intermediate predicates that represent abstract states of a trusted swarm

Predicate	Predicate Meaning
$\mathbf{Integrity}_{Code}(A, m_i)$	In a swarm member m_i comprising of multiple applications, application A running on m_i has been checked for its code integrity.
$\mathbf{Integrity}_{CFA}(A, m_i)$	In a swarm member m_i comprising of multiple applications, application A running on m_i has been checked for its control-flow integrity.
$\mathbf{T}_{att}(m_i, \mathcal{V}\text{rf})$	Time taken to run attestation when $\mathcal{V}$rf attests directly m_i
$\mathbf{T}_{att}(S, \mathcal{V}\text{rf})$	Time taken to run swarm attestation when $\mathcal{V}$rf attests swarm S

Table 4.8: Axioms for designing a trusted swarm

Axioms
(Ax1) $\forall\ R[m_i \leftarrow S],\ \mathbf{Integrity}_{Code}(A, m_i) \Leftrightarrow \mathbf{Trusted}_{\mathbf{Soft}}(S)$
The software of Swarm S (comprising of all members m_i hosting TPMs T_i) is considered trusted if and only if every T_i is trusted and each application $A_i \in m_i$ has code integrity. The communication integrity among two members m_i and m_j and between members m_i and Vrf are addressed in (Ax3).
(Ax2) $\forall\ R[m_i \leftarrow S],\ \mathbf{Trusted}_{TPM}(t_i) \wedge \mathbf{Trusted}_{Comm}(m_i, m_j) \wedge \mathbf{Trusted}_{Comm}(m_i, Vrf) \Leftrightarrow \mathbf{CommSecure}(S)$
The Swarm S (comprising of all members m_i hosting TPMs T_i) guarantees the integrity of exchanged communication data among devices if and only if every m_i and T_i are trusted, the communication channel among two members m_i and m_j and between members m_i and Vrf is trusted.
(Ax3) $\forall\ R[m_i \leftarrow S],\ \mathbf{Trusted}_{TPM}(t_i) \wedge \mathbf{Integrity}_{Code}(A, m_i) \wedge \mathbf{Integrity}_{CFA}(A, m_i) \Leftrightarrow \mathbf{Trusted}_{CFA}(S)$
The Swarm S (comprising of all members m_i hosting TPMs T_i) has control flow safety if and only if the tracer checking the control flow integrity of each member m_i is secure and each application $A_i \in m_i$ has code integrity and control-flow integrity.
(Ax4) $\forall R[m_i \leftarrow S] \wedge \forall R[m_j \leftarrow S],\ \mathbf{Trusted}_{Comm}(T_i) \wedge \mathbf{Privacy}_(m_i, m_j) \wedge$ $\mathbf{Privacy}(m_i, Vrf) \Leftrightarrow \mathbf{Privacy}(S)$
The Swarm S (comprising of all members m_i hosting TPMs T_i) is considered privacy-preserving if and only if T_i associated to each member m_i is physically secure, uses secure cryptographic primitives, provides trusted communication with the TSS, and each interacting members m_i and m_j authenticate themselves in a privacy-protecting way the communication between two members m_i and m_j is trusted
(Ax5) $\sum_{n=1}^{N} T_{att}(m_i, \mathcal{V}\text{rf})$ ¿ $\mathbf{T}_{att}(S, \mathcal{V}\text{rf}) \Leftrightarrow \mathbf{Efficient}(S)$
Swarm S is efficient if and only if the time it takes to attest individually each device $m_i \in S$ is greater than the time it takes to attest the swarm S.

4.2.3 Run-time Verification of Devices

ASSURED will provide runtime verification capabilities for the devices belonging to the target system, ensuring that they have not been compromised by a malicious party. For example, the ASSURED runtime attestation process can address memory-safety vulnerabilities, which may cause security sensitive information leakage. These are closely related to type-safety vulnerabilities, which are typically addressed during design-time, but may also be detected during runtime in

the case they were not initially identified. Also, the loaded executable binaries should be checked dynamically during run-time through Control Flow Attestation (CFA), so that it is verified that all control flows and information transfers are expected and permitted by the target device.

In the context of run-time verification of the devices, the security predicates are given in Table 4.9. These aim to first capture the requirements from the side of the Prover, i.e., the device that needs to be verified. Specifically, the device should be equipped with a **Trusted Computing Base**, containing a **Runtime Tracer** and a **TPM module**, so that it is able to perform attestation of the loaded software. Also, the Prover needs to be able to create trustworthy **Attestation Keys**, as well as capture sign their traced data in a trustworthy manner.

The components required in the context of the runtime verification and attestation method are a Prover (device to be attested), Verifier (who will attest to the trustworthiness of the Prover), an attestation protocol, measurement functionality, and a root of trust, where:

- **Verifier:** The Verifier attests to the trustworthiness of the information provided by the Prover about its status. Considering the domain $\mathcal{D}$ of all possible sequences of Prover states, the verifier realizes a mapping $f : \mathcal{D} \rightarrow \{\bot, \top\}$, where $\top$ indicates the acceptance of the Verifier and $\bot$ the rejection, s.t., the attestation failed.
- **Prover:** The Prover collects measurements on its status and provides the relevant evidence to the verifier. In ASSURED, the Runtime Tracer provides reports about the device according to the provided security policy.
- **Attestation:** The attestation procedure involves the Prover supplying information about the Prover's status to a Verifier, which evaluates whether the Prover's claims of trustworthiness are valid. The decision-making process of the Verifier is based on the provided information.
- **Measurement:** Measuring a target means collecting its status data, which supports the Prover's trustworthiness claim.
- **Root of Trust:**, i.e., a Trusted Platform Module and the measurement functionality, installed on the side of the Prover, that reliably signs the evidence collected by the measurement functionality in a non-forgeable way using strong cryptography. The verification scheme relies on its known, deterministic behavior, and in particular its resilience against manipulations of attackers.
- **Attestation Protocol:** The attestation protocol is a cryptographic protocol that is used by the Verifier and Prover to provide evidence of the prover's state to other parties without sacrificing its privacy requirements.

Table 4.9: Predicates for run-time verification of devices

Predicate	Predicate Meaning
$\mathbf{Static}_{Attestation}(P, X)$	The device hosting the prover P is capable to perform static attestation of the loaded software X that is used by the dynamic attestation on the prover-side to a remote verifier
$\mathbf{TCB}(P, R, T)$	The device hosting the prover P contains a Trusted Computing Base which contains a tracer R and TPM T
$\mathbf{TracerKey}(P)$	The device hosting the prover P has an embedded key available only to software executing in the trusted execution environment
$\mathbf{AttestationKey}_{TPM}(T, P)$	The TPM T hosted by device P contains an Attestation Key
$\mathbf{KeyBound}(K, A)$	Key K is bound to the authorization of key A
$\mathbf{CommSecure}(P, V)$	The integrity and authenticity of the data exchanged between prover P and verifier V is not violated.
$\mathbf{TracerAttestable}(P, R)$	Prover P can generate a trustworthy reports using tracer R
$\mathbf{TrustworthyTraces}(P, R)$	Tracer R can sign traces in a trustworthy manner in device P
$\mathbf{TrustworthyCorrectTraces}(P, R)$	Tracer R can generate trustworthy traces in device P

Next, based on the aforementioned predicates, in Table 4.10 we provide the axioms that should hold in the context of run-time attestation. These state the conditions under which the Prover can perform measurements and collect traces in a trustworthy manner, and can sign these traces with an Attestation Key in a way that proves that they represent a trusted state. Also, the axioms express the conditions under which the Prover and the Verifier can communicate, so that the authenticity of the exchanged data is not violated.

Table 4.10: Axioms for run-time verification of devices

Axioms
(Ax1) $\mathbf{Static}_{Attestation}(P_i, R_i) \wedge \mathbf{TCB}(P_i, R_i, T_i) \Leftrightarrow \mathbf{TracerAttestable}(P_i, R_i)$
The device hosting the prover P_i can start the tracer R_i in a and provide a trustworthy attestation reports that the tracer is loaded correctly and therefore can collect valid traces, if and only if the tracer R_i resides in a prover P_i's TCB with TPM T_i and are capaple of performing static attestation.
(Ax2) $\mathbf{TracerAttestable}(P_i, R_i) \wedge \mathbf{KeyBound}(\mathbf{AttestationKey}_{TPM}(T_i, P_i), \mathbf{TracerKey}(P_i))$ $\Leftrightarrow \mathbf{TrustworthyTraces}(P_i, R_i)$
Tracer R_i can sign traces in a trustworthy manner in the device hosting prover P_i, if and only if the Attestation Key in prover P_i's TPM T_i is bound to the authorization of the Tracer Key associated with prover P_i.
(Ax3) $\mathbf{TrustworthyTraces}(P_i, R_i) \wedge \mathbf{PhysicalMemoryAccess}(R_i) \wedge \mathbf{CommSecure}(P_i, V_i)$ $\Leftrightarrow W\mathbf{TrustworthyCorrectTraces}(P_i, R_i)$
Tracer R_i generates correct traces in a trustworthy manner in the device hosting prover P_i. Further, the integrity and authenticity of the data exchanged between prover P_i and verifier V_i is not violated.

4.2.4 Direct Anonymous Attestation (DAA)

In ASSURED, we propose the use of a **Direct Anonymous Attestation (DAA)** scheme, which aims to enhance the privacy guarantees of complex supply chain ecosystems. The goal of the DAA scheme is to **shift trust assurance from the infrastructure to the edge devices**, thus creating a **decentralized approach** in the trustworthiness establishment process, in a privacy preserving manner. To this end, DAA aims to provide authentication of data in terms of their origin, meaning that a device should be able to provide verifiable evidence that it represents a **valid and enrolled user**, while simultaneously ensuring that this evidence **cannot be linked to the device's ID or location**. The DAA scheme proposed in ASSURED is a **platform authentication mechanism**, that enables the provision of privacy-preserving and accountable authentication services. This is achieved by employing **group signatures**, so that any verifying entity can verify a platform's credentials in a privacy preserving manner. Therefore, the predicates defined in Table 4.11 capture the requirement for privacy in the generation of DAA keys, while being certified by a trusted authority and remaining physically secure.

Table 4.11: Relevant DAA Predicates

Predicate	Predicate Meaning
$\mathbf{Privacy}_{TPM}(T)$	TPM's key privacy is trusted when generating DAA key
$\mathbf{TPMCredentialCert}(T_{AK}, I)$	A TPM DAA Attestation Key T_{AK} is certified by an issuer (Privacy CA) I and have a valid credential.
$\mathbf{AttestSafe}$(AK)	A TPM T safely uses its attestation AK to create DAA S is physically secure

Taking these predicates into consideration, the axioms defined in Table 4.12 specify the conditions under which a malicious party cannot create signatures that appear legitimate (**unforgeability**), the DAA mechanism does not reveal the identity of the party to be attested (**anonymity**), and no legitimate user can be accused of being illegitimate (**non-frameability**).

DAA Axioms
(Ax1) Unforgeability: $\forall\ \mathbf{Trusted}_{Issuer}(I), \mathbf{AttestSafe}(AK) \Leftrightarrow \mathbf{DAAUnforgeability}$
The DAA unforgeability is achieved if and only if each the Issuer I, the Privacy CA, is honest, The TPM is physically secure, runs CryptoSafe algorithms and safely creates DAA attestations.
(Ax2) Anonymity: $\forall\ \mathbf{Trusted}_{TPM}(P), \mathbf{AttestSafe}(AK) \Leftrightarrow \mathbf{DAAAnonymity}$
The DAA anonymity is achieved if and only if each the platform P is honest, The TPM is physically secure, runs CryptoSafe algorithms and safely creates DAA attestations.
(Ax3) Non-Frameability: $\forall\ \mathbf{Trusted}_{TPM}(P),\ \mathbf{AttestSafe}(AK),\ \mathbf{Safebinding}(EK, AK) \Leftrightarrow \mathbf{DAANon-Frameability}$
The DAA non-frameability is achieved if and only if the platform P is honest, The TPM is physically secure, it runs CryptoSafe algorithms, it safely creates DAA attestations, and the attestation key AK is bound to the correct TPM.

Table 4.12: Axioms for DAA Security Properties

4.2.5 Jury-based Attestation

Jury-based Attestation is a methodology analogous to Swarm Attestation, which was presented in Section 4.2.2, in the sense that it enables the attestation of a large group of interconnected devices, referred to as a swarm. However, the difference is that rather than requiring an attestation challenge by an external Verifier in order to attest the swarm, Jury-based attestation is a scalable approach that enables **a set of devices to attest each other, without the need for an external Verifier**. The main challenge in this approach is the absence of a centralized verifying entity, and the possibility that one or more of the verifying devices may act maliciously. The Jury-based attestation protocol proposed in ASSURED is able to address these issues, by implementing a **Jury selection and delegation process**, where each juror attests a potentially untrustworthy device, and the Jury makes a final decision on its trustworthiness based on the results of these attestations.

In ASSURED, Jury-based attestation is invoked in cases where a dispute arises regarding a particular attestation. For example, if a Verifier produces a failed attestation result but the Prover claims this verdict is erroneous, the Prover can request a Jury-based attestation in order to provide a means to resolve this dispute. In other words, Jury-based attestation acts as a **second line of defense** for the attestation schemes employed in ASSURED, and aims to address issues that may lead to failure of any of the other aforementioned attestation schemes. The Jury-based attestation process requires the following components:

- **Remote Attestation Scheme:** The basis of an agreement of the state of a device is established via a remote attestation scheme executed directly between devices. However, for the Jury-based approach, such a remote attestation scheme needs to (1) not depend on a central trusted entity, i.e., work directly between two devices, and (2) be lightweight enough to be executed on these devices, which may be limited in computational resources. The latter highly depends on the use case and the capabilities of the deployed devices. The previous sections detail the different approaches to attestation.
- **Attested Election:** One phase of the Jury-based Attestation is a distributed election among the network. For an efficient election, a TEE is required to ensure devices correctly follow the election protocol while avoiding the message overhead typically induced by distributed protocols. However, our approach for the election is agnostic regarding the specific TEE platform.
- **Byzantine Fault Tolerance:** For the final agreement on the respective attestation result, Jury-based Attestation relies on a distributed consensus protocol, namely Byzantine Fault Tolerance. These protocols provide two properties when finding an agreement among the consensus group. *Safety* ensures that no inconsistent decisions are accepted, while *liveness* ensures that all decisions are eventually made.

Next, in Table 4.13, we define the predicates for jury-based attestation. These aim to capture the requirements for the Jurors, namely, that each Juror should be equipped with a TCB, is elected under safe circumstances, and is able to perform operations in a trustworthy manner. We also define predicates regarding the attestations performed by the jurors, in order to capture the requirements of trustworthiness of the results of these attestations, as well as attestations by the Jury as a whole. Finally, we define a predicate for the timely execution of an attestation by a Juror, since a prolonged attestation time can indicate a compromise of integrity.

Table 4.13: Predicates for Jury-based Attestation

Predicate	Predicate Meaning
$\mathbf{TCB}_J$	A juror J contains a Trusted Computing Base.
$\mathbf{CorrectExecution}(J, O)$	A juror J can perform an operation O in a trustworthy manner.
$\mathbf{JuryElected}(\mathcal{J})$	A jury $\mathcal{J}$ has been elected correctly under safe circumstances.
$\mathbf{ResultReceived}(J, r)$	An attestation result r has been received from the juror J
$\mathbf{TrustedAttestationResult}(J, R)$	The result R from a juror J is trusted to be correct.
$\mathbf{StaticAttestaion}(E)$	The result of the static attestation E can either be true or false.
$\mathbf{OutputSafe}(\mathcal{J})$	The attestation output from the jury $\mathcal{J}$ is considered to be safe and trustworthy.
$\mathbf{OutputFresh}(\mathcal{J})$	The attestation output from the jury $\mathcal{J}$ is considered to be fresh.
$\mathbf{TimelyCorrectReceived}(R, J)$	An attestation result R from juror J has been received within a timely manner without any integrity compromises.

Based on the defined predicates, we next provide the axioms that should hold for Jury-based attestation in Table 4.14. These axioms define the conditions for the correct construction of the Jury, by selecting the Jurors out of a set of potential candidates so that each Juror is equipped with a TPM and can perform operations in a trustworthy manner. We also define conditions for a Jury-based attestation to reach a verdict, which involve all the Jurors being able to produce an attestation result in a trustworthy manner. Also, a Jury verdict can be considered safe if the majority of Jurors can produce non-faulty attestation results in a timely manner.

Jury-based Attestation Axioms

(Ax1) $\forall\ j \in \mathcal{J}_{cand}$, $\mathbf{TCB}_j \wedge \mathbf{CorrectExecution}(j, ElectionProtocol) \Leftrightarrow \mathbf{JuryElected}(\mathcal{J} \subseteq \mathcal{J}_{cand})$
A jury $\mathcal{J}$ is correctly constructed as a subset of all candidates $\mathcal{J}_{cand}$ if and only if each juror j contains a Trusted Computing Base (TCB) and will correctly enforce the correct execution of the election protocol.
(Ax2) $\forall\ j \in \mathcal{J}$, $\mathbf{JuryElected}(\mathcal{J}) \wedge \mathbf{ResultsReceived}(R, j) \wedge \mathbf{CorrectExecution}(j, AttesttionProtocol) \Leftrightarrow \mathbf{TrustedAttestationResult}(j, R)$
An Attestation Result R can be generated and trusted if and only if each juror $j \in \mathcal{J}$ executed the attestation protocol correctly, and all results for each juror has been received.
(Ax3) $\exists j \in \mathcal{J} : \lvert j\rvert > \frac{\lvert\mathcal{J}\rvert}{2} + 1$, $\mathbf{TrustedAttestationResult}(j, R) \wedge \mathbf{StaticAttestation}(R \vee \neg R) \Leftrightarrow \mathbf{OutputSafe}(\mathcal{J})$
A jury output is safe if and only if the majority of the jury members are not faulty and produce non-faulty attestation results, which they should contain the same binary result (true or false).
(Ax4) $\exists j \in \mathcal{J} : \lvert j\rvert > \frac{\lvert\mathcal{J}\rvert}{2} + 1$, $\mathbf{TrustedAttestationResult}(j, R) \wedge \mathbf{TimeleyCorrectReceived}(R, j) \Leftrightarrow \mathbf{OutputFresh}(\mathcal{J})$
A jury output is fresh if and only if the majority of the jury members have created the attestation report in a timely manner and those received are safe.

Table 4.14: Axioms for Jury-based Attestation

4.3 Secure Key Management

One fundamental aspect of any security framework, such as the one implemented in ASSURED, are the **cryptographic keys**. In the context of ASSURED, several security services are envisioned, each one of which requires the generation and use of several different kinds of keys, as it was shown in the description of the Trust Models presented in Chapter 3. For example, some of the services of ASSURED that employ cryptographic keys are the following:

- The attestation services implemented in ASSURED require the use of **Attestation Keys**, in order to perform the operations required in various phases of the attestation process.
- A cryptographic key is required in order to achieve **secure communication between the TPM and the Runtime Tracer**.
- **Symmetric keys** need to be employed for the secure communication **between different devices of the system**.
- **Public-private key pairs** are required in order to provide access to the Blockchain.
- Cryptographic keys are required in the context of **Attribute-based encryption**, which is a purely decentralized approach to encryption implemented in ASSURED.

From all the above, it follows that, due to the use of several different cryptographic keys that are used in different ways by each of the multitude of security services offered by ASSURED, the need arises for the definition of methodologies for **secure key management**. We aim to employ the underlying TPM-based Roots of Trust, as well as their offered functionalities and features regarding hierarchy, in order to define various key usage and key protection policies, since each key has different protection requirements.

Consider, for example, the aforementioned **Attribute based encryption** scheme implemented in the ASSURED framework. This scheme is a novel methodology that enables the encryption of a dataset multiple times, granting different levels of granularity of data to different parties, depending on properties they may have. This is instantiated by using the underlying TPM, which can only create a decryption key to obtain access to data that has been encrypted with this method, only if it is characterized by the appropriate properties. Also, it is important to note that this encryption approach is **decentralized**, meaning that these keys can be created by any device equipped with a TPM, and a centralized entity is not required. It is evident that this encryption scheme creates the need for a key management scheme that can handle the storage and usage of the various keys generated for different levels of attribute-based access.

From all the above, it follows that the ASSURED architecture must provide methods in order to handle secure key generation, key distribution, key attributes, and key destruction. In the following sections, we will discuss the above TPM key functionalities as explained in [10] and describe, in Table 4.15, some relevant TPM key commands needed for the TPM functionality in the ASSURED framework.

Next, we provide the features that the underlying trusted component of each device should have, as part of a system considered by ASSURED. Note that, in the following, we consider that TPMs are used as the trusted components in the formulation of the trust models, since they were selected for the implementation of the ASSURED framework. However, any trusted component that can support the defined trust model is appropriate for consideration in ASSURED. Therefore, the features that the underlying trusted component should have are as follows:

Key Generation With generating an ASSURED user key, the most important thing the user has to consider is that the key is generated randomly. If a poor random number generator is chosen, the picked key won't be secure. TPM's keys are organized in key hierarchies. Each TPM key can be created as a primary key retrieved from a secret seed or a key created randomly. Keys in a TPM hierarchy have a parent-child relationship. Each hierarchy has a primary (root) parent keys and trees of child keys. A parent is an encryption key, and a parent key wraps (encrypts) child keys before leaving the TPM to ensure a secure boundary. A TPM has some non-volatile memory to store long-term keys. Both the Endorsement Key (EK) and the Storage Root Key (SRK), which is created by the TPM2_CreatePrimary command and forms the basis of a key hierarchy that manages secure storage, are stored in the TPM non-volatile memory as shown in Figure 4.1.

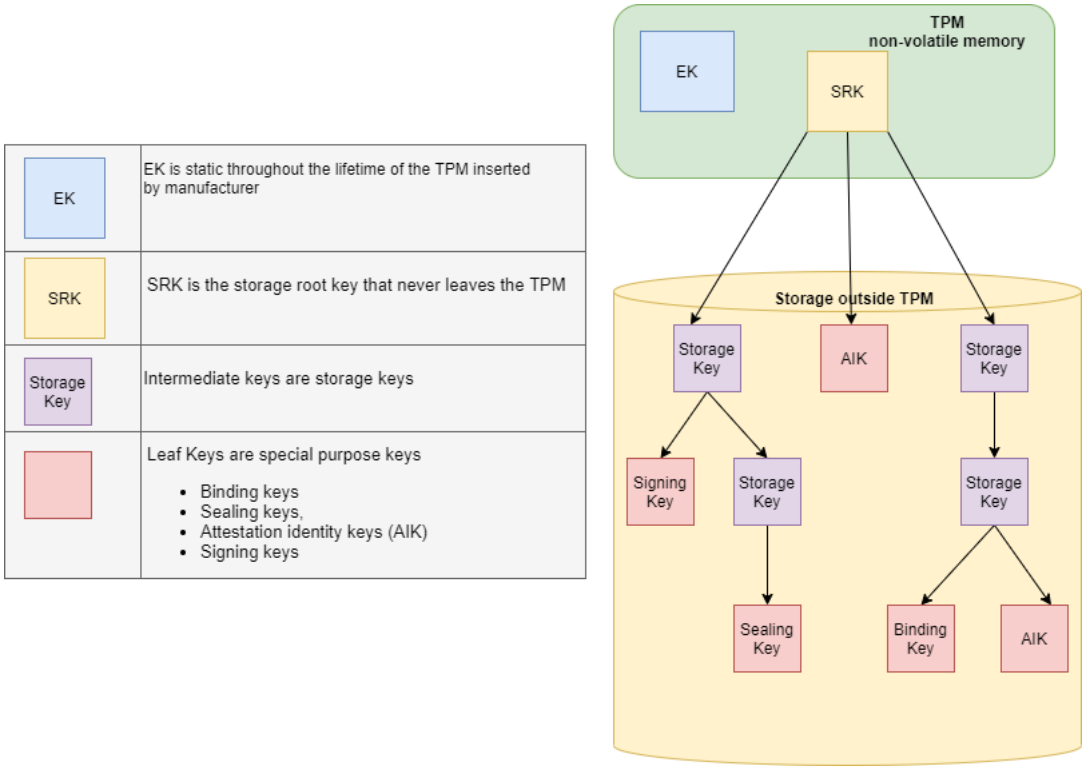

Figure 4.1: TPM Key Hierarchy

Keys can have use restrictions as well. They can be specified as only signing or only decryption keys, and they can be restricted to only signing or decrypting specific data. The TPM endorsement key EK is a primary encryption/decryption key, while the TPM attestation key AK is a random key used for signing purposes. The TPM endorsement key EK represents the parent of AK. In general, key attributes determine how the key can be used. The attribute value is established when the key is created and cannot be changed. An example of a key attribute is the "Restricted" attribute which means that the key can only be used on structures that have a known format, e.g., when a signing key is restricted, the key can only sign the data created by the TPM itself. Another attribute key designation is migratable or non-migratable. This key attribute determines whether a key may be transferred from one TPM to another (migratable). Migratable keys are cryptographic keys that are not bound to a specific TPM. They can be generated either inside or outside of a TPM and with appropriate authorization. Keys can be certified by a trusted authority that can validate the TPM public key, the TPM key's attributes, and its access policy. The TPM can also be used as a certificate authority itself; it can certify its own generated keys by signing them. The command TPM2_Certify asserts that an object with a Name is loaded on the TPM

then signs this object. Because the name cryptographically represents the object's public area, a relying party can be assured that the object has an associated private part. The name may also include the key's attributes, including whether it's restricted, fixed to a parent, or fixed to a TPM, and the authorization policy.

Symmetric and Asymmetric Keys: TPM 2.0 supports a variety of asymmetric algorithms as well as symmetric algorithms. A symmetric signing key can be used in TPM HMAC commands. TPM 2.0 can do symmetric signing or symmetric encryption such as AES. In TPM 2.0, child keys are wrapped with the symmetric key of the parent, even if the parent itself is an asymmetric key. All storage keys have a symmetric secret.

Key Duplication: Keys can be duplicated (wrapped with a different parent), and all children are duplicated when the parent is duplicated. Duplication is subject to restrictions. In order to do duplication of keys, these keys must be created to be duplicable, and they must have a policy created for them that has TPM2_Policy_Command code with TPM2_Duplicate selected. Some keys are fixed to the TPM, i.e., they can't be duplicated. Some are fixed to their parent and so can only be duplicated when the parent is duplicated.

Key Distribution: In the context of the ASSURED framework, sometimes keys need to be distributed among ASSURED components. For instance, to access blockchain ledger via user's attributes, some attribute-based keys should be created by a trusted authority such as the security context broker and then securely distributed to the ASSURED users that possess such attributes. The TPM design adopted in the ASSURED framework makes this easy. When each system is set up, a non-duplicable storage key is generated on the system, and a trusted authority keeps a record associating this key with the system id. At some later point, if the trusted authority wants to distribute the key to a group of users, the following takes place:

1. The trusted authority creates the key using TPM2_GetRandom.
2. The trusted authority encrypts the key with the public portion of the target user's storage key.
3. The trusted authority signs the encrypted key with its private signing key.
4. The encrypted key is sent to the user along with a signature that proves it came from the trusted authority.
5. The user verifies the signature on the encrypted key by loading the trusted authority public key. (This can be done with the TPM using TPM2_Load and then using TPM2_VerifySignature.)
6. The user imports the verified, encrypted key into its system using TPM2_Import, getting out a loadable, encrypted blob containing the key.
7. The user loads the key when the user wishes to use it, using TPM2_Load and uses it as normal.

Key Split: A key split is a cryptographic construct where two sets of entropy are used to produce a key. Neither one alone is able to provide even a single bit of the final key's entropy. For instance, one split of the key may be the hierarchy's seed inside the TPM. The other split can be stored securely when not in use (for example, in a smart card) is held outside the TPM in the template. The aim of the key split is to prevent an attacker who knows the TPM's seed from being able to determine the secrets of the primary key.

Key Destruction: Once an ASSURED key has been created, it is sometimes important to be able to destroy it as well. For instance, the authorization has been corrupted, or the machine is being re-purposed. Keys that are stored in software can never be destroyed because they may have been copied almost anywhere. In the ASSURED framework, TPMs provide this facility in several easy ways. If the key used is a primary key, the easiest way to destroy it is to ask the TPM to change its copy of the seed of the hierarchy on which it was created (usually the storage hierarchy). TPM2_Clear does this for the storage hierarchy. Clearing the TPM destroys all non-duplicable keys that are associated with the hierarchy, evicts all keys in the hierarchy from the TPM, and changes the seed, preventing any primary keys previously associated with that hierarchy from being re-generated. Duplicable keys can no longer be loaded into the system, although if they have been duplicated to a different system, they may not be destroyed. It's also possible to destroy keys that are generated outside the TPM. If the copies outside the TPM are destroyed, then evicting the key from persistent memory also destroys the key.

4.3.1 Key Attributes

The next aspect that needs to be defined in the context of ASSURED are the **Key Attributes**. These are required in order to determine the context that each key can be used in. Specifically, when a key is generated, the reason that it will be used needs to be determined. For example, we need to determine if a key can only be used for signing data, or if it can be used for encryption operations as well. The key attributes, which are set at the time of their generation, include the following: (i) **Use attribute:** signing or encryption, (ii) **Type attribute:** symmetric or asymmetric, (iii) **Restrictions on duplication attribute**, and (iv) **Restrictions on use attribute**.
Next, we expand on each of these key attributes in the context of ASSURED.

Use Attribute: signing or encryption This attribute specifies the purpose that the TPM key was created. It mainly specifies if the TPM key is a signing or a decryption key. The sign attribute is used to allow the key to perform signing operations, e.g. this key can be used for the TPM2 Sign() command. The decrypt attribute is used to allow the key to perform decryption operations, e.g. this key can be used for the TPM2 ECDH ZGen() command.

Type Attribute: symmetric or asymmetric A symmetric signing key can be used in TPM HMAC commands. TPM 2.0 can do symmetric signing (a MAC) with a key that is never in the clear outside the TPM. The TPM library specification includes symmetric encryption keys that can be used for general-purpose encryption, such as AES. Asymmetric keys are used by asymmetric algorithms for digital identities and key management. An asymmetric key pair consists of a private key known only to one party and a public key known to everyone. Calculating the public key from the private key is relatively easy computationally, but calculating the private key from the public key should be computationally infeasible.

Restrictions on duplication Attribute TPM keys have two attributes that control duplication. A key may be locked to a single parent on a single TPM and never duplicated. On the other hand, a key may be freely duplicated to another parent on the same or another TPM. The controlling key attributes are defined as follows:

- fixedTPM: A key with this attribute set to true can't be duplicated. Although the name seems to permit duplicating a key from one location in a hierarchy to another within a TPM, this isn't the case.

- fixedParent: A key with this attribute set to true can't be duplicated to (rewrapped to) a different parent. It's locked to always to have the same parent.

Restrictions on Use Attribute We introduce two types of restrictions attributes as follows:

- Restricted Signing Keys

  A variation on the key attribute sign (a signing key) is the restricted attribute. The use case for a restricted key is signing TPM attestation structures. These structures include Platform Configuration Register (PCR) quotes, a TPM object being certified, a signature over the TPM's time to create a TPM time stamp or a signature over an audit digest. The signature over a digest restriction is essential when creating PCR quotes for ASSURED attestation services. For instance, a user could generate a digest of any PCR values and use a nonrestricted key to sign it. The user could then claim that the signature was a quote. However, the relying party would observe that the key was not restricted and thus not trust the claim. A restricted key provides assurance that the signature was over a TPM-generated digest.

- Restricted Decryption Keys

  A restricted decryption key is, in fact, a storage key. This key only decrypts data that has a specific format, including an integrity value over the rest of the structure. Only such keys can be used by parents to create or load child objects or to activate a credential.

TPM Command	Description
TPM2_Create and TPM2_ CreatePrimary	Creates all key types from templates
TPM2_Load	Loads wrapped private keys onto the TPM
TPM2_LoadExternal	Makes a loaded key persistent
TPM2_EvictControl	Loads public keys and possibly plaintext private keys onto the TPM
TPM2_FlushContext	Removes a key from the TPM
TPM2_VerifySignature	Verifies a digital signature
TPM2_Certify	Certifies that a TPM key sign another loaded key inside the TPM
TPM2_Import	returns a normal TPM-encrypted blob
TPM2_Clear	Clears the TPM's storage hierarchy

Table 4.15: Relevant TPM2.0 Commands

4.3.2 Policy-based Key Usage

Policy-based Key Usage constitutes the main feature that the trusted component needs to be able to offer in the context of ASSURED. Essentially, as we have already mentioned in regards to the communication between the Tracer and the TPM, in ASSURED we need to protect the usage of the keys, depending on the state and the attributes of the device. This is achieved by leveraging one of the strong features of the TPM, which is the **use of policies**. Next, we describe how the TPM can offer policy-based key usage, which is one of the most important features it provides, and we leverage for the majority of the keys employed across the security services provided by ASSURED. These features need to be **provided by the underlying trusted component**, in order to be able to capture the required trust model, and protect the keys from malicious usage. It should be mentioned that, in the following, the descriptions given in the following consider a TPM as the trusted module. However, it is possible to support different kinds of trusted components as well.

Trusted Platform Modules protect the use of keys through two distinct methods: **password-based** and **policy-based**. In password-based protection, a key cannot be used unless the correct password is provided. This can solve basic authentication requirements but is very limited, as it requires the usage of a password in order to control the use of the key. In policy-based protection, also called **Extended (or Enhanced) Authorization (EA)**, we can define a policy or a set of policies that must be satisfied before the key can be used. These policies can provide multiple factors to the authentication process.

In ASSURED, we define a particular authentication challenge. We need to send Traces from the TCB (secure world) through the TSS (insecure world) to the TPM (secure world) for signing. To this end, we must guarantee that our signing key cannot be misused in any way by the insecure world. We achieve this protection by using policies, which may dictate that the signing key can only be used if the Tracer authorizes it.

In this section, we will introduce the **technical details** of these key usage policies. We will show in detail how policies are created, and how they are being used to govern entities such as keys. As we have used a particular policy, `PolicySigned`, we will use this as an example throughout the section.

`PolicySigned` is a simple, yet powerful, policy. It states that to satisfy the policy (hence be able to use the key), a signature must be provided by an authorizing key. For example, in a workplace environment, to access a service, the supervisor must provide manual approval for its usage. However, this approval can also be granted by methods such as a fingerprint reader which afterwards creates the signature, attesting that a correct fingerprint has been provided. Such approval is protected against replay attacks by using a nonce from the TPM, and it can even include an expiry time, or more importantly **an authorized action**.

Next, we provide details regarding the construction of these key usage policies. A policy is represented as a hash digest and is part of the public TPM key structure. However, an integrity check is made when the key is loaded, so it cannot be changed. That means any entity holding the public key can verify the policy of use. Depending on the policy, the digest is constructed in different ways. In `PolicySigned`, it's constructed as follows:

$$Policy = H(00_{16} \,||\, TPM\_CC\_PolicySigned \,||\, authObject \rightarrow Name \,||\, policyRef)$$

The first part is used in other kinds of policies except for key usage, and is outside the scope of this deliverable. The second part, `TPM_CC_PolicySigned`, is the command code of the policy.

This is a static value, in this case, `0x00000149`. The third part is the *name* of the authorizing key. The name is essentially a unique hash representing a particular key. The last part is an optional addon and can be omitted. An example where this is useful is in the case of a fingerprint reader. A policy might dictate that the fingerprint reader needs to sign off to get access to the key, but only some people can have access. In this case, the field would represent *whose* fingerprint should be provided.

Next, we expand on the process of signing data by using the authorizing key. The authorizing key defines the restrictions of its authorization by deciding what it is permitted sign. The digest to be signed is referred to as the Authorization Digest, and it can contain the following:

- **nonce**: To protect against replay protection, a session nonce from the TPM can be included.
- **Expiration**: A time limit for how long the authorization is valid.
- **CpHashA**: What command and parameters are authorized to be executed by the key.
- **PolicyRef**: Additional value (for example the userid)

The digest is then constructed as follows:

$$aHash = H(Nonce \mid\mid Expiration \mid\mid CpHashA \mid\mid PolicyRef)$$

If there are no restrictions, the hash to sign would be:

$$aHash = H(00 \mid\mid 00 \mid\mid 00 \mid\mid 00_{16})$$

The xcpHashA is constructed in different ways, depending on the command to be authorized. An example could be a signing operation. In this case, the cpHashA would look like this:

$$cpHashA = H(TPM2\_Sign \mid\mid Key \rightarrow name \mid\mid Digest)$$

That is the command code for the authorized command (*TPM2_Sign*), the name of the key doing the signing, and which digest is authorized to sign.

Let us take a look at how the workflow is. Say we have a key K which is protected by policy P, which is `PolicySigned` without any policy reference, and requires a signature from authorizing key A.

The key policy is then $K_p = H(TPM\_CC\_PolicySigned \mid\mid A \rightarrow Name \mid\mid 00_{16})$. For the sake of simplicity, consider that the policy calculates to $K_p = 0x0A0C$.
The Authorizing Key determines it does not require any restrictions, and constructs an empty $H_a = H(00 \mid\mid 00 \mid\mid 00 \mid\mid 00_{16})$, and signs it: $\sigma_A = Sign(H_a)_A$.

The user of the TPM must *prove* to the TPM that it possesses a valid signature from the authorizing key A. The way to do this is through a session. The user instructs the TPM to start a (policy) session, using `TPM2_StartAuthSession`. When this command is executed, the TPM will start an internal session with an empty session digest and provide the user with a Session Handle (S_0). It is now the job of the user to get this session digest to *match* the policy digest of the key, and the only way to do so is through policy commands, such as `TPM_PolicySigned`. These particular commands take the following arguments: (i) **AuthObject** Key handle to the public part of the

authorizing key, (ii) **Session** Session Handle, (iii) **nonce** (Same as described earlier), (iv) **Expiration** (Same as described earlier), (v) **CpHashA** (Same as described earlier), (vi) **PolicyRef** (Same as described earlier), and (vii) **Auth** The signed digest.

Consider that the user already loaded the public key into the TPM and has a Key Handle KH_{auth}. The user then calls the policy command with KH_{auth} as AuthObject, S as Session Handle, nothing for the nonce, Expiration, cpHashA, or policyRef, and lastly σ_A. The TPM will now *internally* reconstruct the digest H_a based on the inputs (nonce, Expiration, etc.) and verify that σ_A is signed by the key loaded as AuthObject. If and only if this is verified, the TPM will then extend the session digest in S with the *command code* of the policy and the *name* of the authorizing key (and PolicyRef if present).

$$S = H(S \mid\mid TPM\_CC\_PolicySigned \mid\mid authObject \rightarrow Name \mid\mid (policyRef)) = 0x0A0C$$

Now the user can execute an operation, for example `TPM2_Sign`, using key K and session S. The TPM will recognize that there is a policy bound to the key, and it then compares the current session digest with the keys policy digest. If and only if these match, the operation is permitted. It follows that if the authorizing key was used to signed a digest with restrictions, then the user must provide these exact restrictions to the TPM in the policy call. Otherwise, the reference hash calculated internally in the TPM would not match, and the signature could not be verified.

The TPM handles the restrictions in different ways. For example, if a cpHash was set, then the TPM would set an internal flag and only execute a command if that command and its parameters match the cpHash. If any different command was executed, the TPM would reject it despite being able to verify the signature.

As shown in this section, policies are very versatile and secure, as all verifications happen within the TPM itself. It is the responsibility of the users to *prove* to the TPM that they complies with the policy.

4.4 Advanced Key Management for ASSURED

In this section, we present the generic predicates and axioms for achieving secure key management in ASSURED, which describe the requirements that should be fulfilled for the secure usage of the keys.

Table 4.16: Predicates for Key Management

Predicate	Predicate Meaning
$\mathbf{CorrectLoadProperties}(K, T)$	All properties of key K is correctly loaded into the TPM T
$\mathbf{Certify}_{CA}(K)$	Key K is certified by Privacy CA
$\mathbf{SecurelyLoadExternalKey}(K, T)$	An external key K is loaded securely into tpm T with the correct properties of the key.
$\mathbf{SecurelyLoadKey}(K, T)$	A key k generated on t is loaded correctly on t with the expected properties
$\mathbf{Safebinding}(EK, AK)$	Each TPM AK binds to the correct TPM with the corresponding endorsement key EK
$\mathbf{TPMCredentialCert}(K, I, T)$	Issuer (Privacy CA) certifies the properties and ownership of key K to TPM T.
$\mathbf{SignSafe}(K)$	The key K can generate safe and correct signatures.

TPM Key Axioms

Ax1	$k, t, \mathbf{Trusted}_{TPM}(t) \wedge \mathbf{Certify}_{CA}(k) \Leftrightarrow \mathbf{SecurelyLoadKey}(k, t)$
	A key k generated on t can be assumed to be securely created and loaded if and only if the Privacy CA certifies it.
Ax2	$t, k_{ak}, k_{tk}, k_{ek}, \quad \mathbf{SecurelyLoadKey}(k_a k, t) \quad \wedge \quad \mathbf{TPMCredentialCert}(k_{ak}, I, t) \quad \wedge \mathbf{Safebinding}(k_{ek}, k_{ak}) \wedge \mathbf{KeyBound}(k_{ak}, k_{tk}) \Leftrightarrow \mathbf{SignSafe}(k_{ak})$
	The Attestation Key k_{ak} can generate secure attestations if and only if the TPM attestation key is securely loaded, has a valid credential, signed by the Privacy CA I, binds it to the correct TPM t by validating the Endorsement Key k_{ek}, and that k_{ak} can only sign when authorized by Tracer Key k_{tk}
Ax3	$k, t,$ $\mathbf{Trusted}_{TPM}(t) \wedge \quad \mathbf{CorrectlyLoadProperties}(k, t) \quad \wedge \quad \mathbf{Certify}_{CA}(k) \quad \Leftrightarrow \mathbf{SecurelyLoadExternalKey}(k, t)$
	A TPM t can only load an external key k securely if the key is certified by the Privacy CA and the properties are loaded correctly.

Table 4.17: Axioms for TPM Keys

4.5 Revocation

Key revocation refers to the functionality that provides the capability to render a key inoperable, in case it is suspected to have been compromised, or if it should be renewed for freshness purposes. In ASSURED, revocation goes beyond traditional key revocation but also entails the revocation of all credentials of a devices that have deemed as suspicious. This include both the revocation of any cryptographic material (i.e., keys) but also any short-term anonymous credentials that might have been generated for the privacy-preservation of the devices while exchanged sensitive information. As will be described in D3.2 [29], ASSURED through the use of the DAA protocol, enables a device to securely self-issue such short-terma credentials (e.g., pseudonyms) so that it can achieve privacy-preserving properties including anonymity, untraceability, unobservability, and unlinkability. In order to implement such advanced revocation in ASSURED [50], we introduce a non-volatile index inside the TPM: the **revocation index**. Each short-term anonymous credential, and its signing key (based on the use of group signatures; i.e., pseudonym), is assigned two bits in this index, which are referred to as **revocation bits**. One bit is shared between all anonymously linked pseudonyms (**hard revocation bit**), and one is unique per pseudonym (**soft revocation bit**).

We bind the pseudonym to the state of both bits. If any of them are set, then the pseudonym is rendered inoperable; otherwise, it's functional. To protect against a rogue user, only the **Security Context Broker** has the authority to set a bit on the index, and therefore to revoke a pseudonym. We ensure this by safeguarding the operations on the revocation index with policies. For reasons described in D3.2 [29], we allow any policy that is signed by the Authorization Key (AK) to act as a valid policy for the revocation index. The Final Policy for the index states that for revocation to

be allowed, the Revocation Authority must sign a **pre-shared command-hash** that determines which bits to set. Only then is a revocation operation allowed. The Authorization Key must also allow an initial write to the index, which activates it.

Table 4.18: Description of Key Revocation Policies

Policy	Policy Meaning
$\mathbf{PolicySigned}(K)$	To satisfy policy, Key K must provide a signature over nonce Can also include what command is authorized.
$\mathbf{PolicyNV}(I, B)$	The index I is compared with B, and is satisfied if comparison succeeds.
$\mathbf{PolicySecret}(I)$	Policy is satisfied if one can authorize to the index I successfully. The name secret stems from an index often being protected by a password or similar, hence one must know the secret.
$\mathbf{PolicyAuthorized}(K)$	Any policy that is signed by K will satisfy this policy.

Table 4.19: Key Revocation Predicates

Predicate	Predicate Meaning
$\mathbf{KeyDestroyed}(K)$	Key K is destroyed and not recreatable.
$\mathbf{IndexImmutable}(I)$	The index I cannot be written to or be recreated.
$\mathbf{WriteIndex}(V, I)$	Write value V to index I
$\mathbf{IndexPolicy}(I, P)$	Index I is protected by Policy P
$\mathbf{KeyPolicy}(K, P)$	Key K is protected by policy P
$\mathbf{IndexActivated}(I, K_{Auth})$	Index I have received an initial write by K_{Auth} authorizing (signing command parameters) a zero-write.
$\mathbf{AuthorizationsLeft}(I, N)$	There are N authorizations left for index I to be used in PolicySecret
$\mathbf{AdditionalIndexesNeeded}$	There is a need for more revocation indexes
$\mathbf{PseudonymRevocable}(K_p, RA)$	Pseudonym K_p is revocable by Revocation Authority RA
$\mathbf{FinalPolicyTrusted}(\mathcal{P}, H)$	The final Policy $\mathcal{P}$ is created correctly and generates revocation hashes H for all pseudonyms.
$\mathbf{RevocationHashesShared}(H, RA, K_p)$	Revocation hashes $\{H_s, H_h\} \in H$ for pseudonym K_p are shared with Revocation Authority RA
$\mathbf{TrustedChannel}(RA, T)$	There is a trusted channel between RA and TPM T
$\mathbf{FinalPolicyContains}(\mathcal{P}, P)$	Final (compound) Policy $\mathcal{P}$ contains policy P
$\mathbf{PseudonymsAnonymouslyLinked}(\mathcal{K})$	All pseudonyms $K_p \in \mathcal{K}$ are linked anonymously
$\mathbf{SharesHRB}(\mathcal{K}, I)$	All pseudonyms $K_p \in \mathcal{K}$ shares a revocation bit in index I
$\mathbf{Signed}(S, D)$	Entity S signed digest D.
$\mathbf{ExecutionSuccess}(C, T)$	TPM T executes command C without error.
$\mathbf{PsudonymSoftRevoked}(K_p)$	Pseudonym K_p can no longer be used but other anonymously linked pseudonyms are functional.
$\mathbf{PsudonymHardRevoked}(K_p, \mathcal{K})$	Pseudonym K_p and all pseudonyms in $\mathcal{K}$ can no longer be used.

To ensure the AK can only authorize these two policies, we bind it to another non-volatile index inside the TPM: The **Authorization Counter Index (ACI)**. This index holds the number of authorizations we can use the AK for. When this number of authorizations has been performed, the AK is rendered inoperable. Again, we must make sure the user cannot reset the ACI, so we introduce a key that guards this index: the **Ephemeral Key**. When the ACI is created, this key is destroyed, and without this key, it's impossible to write to the ACI, hence guaranteeing immutability throughout the system.

The predicates defined in order to capture the requirements for key revocation are given in Table 4.19, the employed key revocation policies are described in more detail in Table 4.18, and the overall axioms in Table 4.20.

4.6 Secure Device Enrollment & Registration

During the execution of a remote attestation process, a verifier needs to be able to verify that the given attestation report was created by a genuine TPM, even if it is unidentified. This essentially translates that all keys and cryptographic material were created correctly by the prover device. To meet this requirement, ASURED will define a secure device enrollment and registration protocol [36] where an Attestation Certificate Authority (CA) (also called Privacy CA) is involved in authenticating that the Attestation Key (AK) holder is a genuine TPM in order to issue a credential for this specific AK. TPM AK credential issuing scheme involves three entities: a set of TPMs, a set of hosts, and a credential provider (Privacy CA). The Privacy CA has a public and secret key pair (cpk; csk), which is used for a signature scheme. Each TPM has a public and secret Endorsement Key (EK) pair (epk; esk), which is used for an asymmetric encryption scheme. The EK is usually certified by the TPM manufacturer. The credential provider has access to an authentic copy of the public endorsement key and its certificate. The TPM also has a public and secret AK pair, which is used for a signature scheme (either a conventional signature scheme or a Direct Anonymous Attestation (DAA) signature scheme).

For the TPM to use its attestation key to create a signature, it should have a valid credential from a trusted CA. The TPM with EK(epk, esk) generates an Attestation Key AK(apk, ask), which should be certified by a Certification Authority CA. The TPM signs its epk using its attestation key ask and sends its signature with the public part of the attestation key apk together with the public part of the endorsement key to the to the Privacy CA through the host as described in Figure 4.2. The Certification authority then verifies the TPM's signature and generates a random secret key k that is used to protect the credential. The CA then encrypts apk concatenated with the secret k using some asymmetric encryption algorithm aENC, sign apk using its signing key csk to generate the credential CRED. Finally, the CA wraps the credential to the TPM through the host.

The TPM unwraps the credential and returns the symmetric decryption key k to the host to decrypt the credential. This TPM operation is called "activate credential." The host decrypts the credential using the decryption key provided by the TPM, then verifies the credential, i.e., verifying that the signature provided on the TPM public attestation key under the CA public key is valid. Finally, the host stores the credential and loads it whenever the platform (TPM + Host) needs to generate signatures. To enhance the key management in TPM 2.0, the TCG has made two major changes: one is requiring an Attestation Key AK to have the attribute "Restricted," meaning that such a key can only be used to sign a message created by the TPM,

The CA credential (Cred) is needed whenever a new edge device wants to get access to a Blockchain ledger and executes a smart contract. Any administrator who can be the security

TPM		Host		CA
ask, esk		epk, apk, cpk		epk, csk
load $AK = (apk, ask)$ If ak is unknown reject else				
$SCER \leftarrow \mathsf{SIG}_{ask}(epk)$	$\xrightarrow{SCER}$		$\xrightarrow{apk,epk,SCER}$	If epk is invalid, return $\perp$ If $SCER/apk/epk$ is invalid return $\perp$ else create k $c \leftarrow \mathsf{aENC}_{epk}(apk\|k)$ $CRED \leftarrow \mathsf{SIG}_{csk}(apk)$
$apk\|k \leftarrow \mathsf{aDEC}_{esk}(c)$ If apk is unknown reject	$\xleftarrow{c}$		$\xleftarrow{c,\ d}$	$d \leftarrow \mathsf{sENC}_k(CRED)$
else	$\xrightarrow{k}$	$CRED \leftarrow \mathsf{sDEC}_k(d)$ Verify $CRED$		Record $(epk, CRED/apk)$

Figure 4.2: The enhanced privacy-CA solution (ePCAS) in [25, 27]

context broker verifies that the TPM has a valid CA credential on the TPM public key. If this verification is successful, then any Blockchain user will be able to authenticate and attest to the new edge device. The attestation report is then forwarded to the security context broker, who verifies the reported attestation result. Upon successful verification, the broker provides the new edge device with Blockchain keys that allow the device to access the ledger, securely download and execute the smart contract and upload its attestation results on the Blockchain ledger.

Table 4.21: Predicates

Predicate	**Predicate Meaning**
$\mathbf{SeucreEnrollment}(D)$	A Device D is able to securely enroll in the system.
$\mathbf{ValidTPM}(T, D)$	A TPM T on device D is valid and not compromised.
$\mathbf{Certifed}_{CA}(T, D)$	A TPM T on device D is certified by a Privacy CA.
$\mathbf{CorrectlyEnrolled}(D)$	A Device D has correctly enrolled in the system.
$\mathbf{CorrectlyCreated}(AK, T, P)$	An Attestation Key AK has been created correctly in TPM T with policy P.
$\mathbf{ConfigurationCorrect}(D)$	A Device has a correct, exptected, configuration.

Ax1	$\forall\ \{d_i, t_i\} \in \mathcal{P},\ \mathbf{Trusted}_{TPM}(t_i) \wedge \mathbf{ValidTPM}(T_i, D_i) \wedge \mathbf{Certifed}_{CA}(T_i, D_i) \Leftrightarrow \mathbf{SecureEnrollment}(d)$
	A Platform $p \in \mathcal{P}$ represented by a device d and TPM t is able to securely enroll if and only if its TPM t is valid and certified by the Privacy CA.
Ax2	$\forall\ \{d_i, t_i\} \in \mathcal{P},\ P_i\ \mathbf{SecureEnrollment}(d_i) \wedge \mathbf{ConfigurationCorrect}(d_i) \wedge \mathbf{CorrectlyCreated}(K_{ak}, T_i, P_i) \Leftrightarrow \mathbf{CorrectlyEnrolled}(d_i)$
	A Platform $p \in \mathcal{P}$ represented by a device d and TPM t is correctly enrolled if it's able to securely enroll, have the expected (correct) configuration, and have created its attestation key K_{ak} using the correct policy P_i defined by the Privacy CA.

Table 4.22: Axioms for Secure Enrollment

4.7 Secure Download & Execution of Smart Contracts

Smart contracts in ASSURED, as described in D1.1 [31], are deployed to the ASSURED DLT network via the Security Context Broker, for deploying and enforcing the attestation policies (as outputted by the Policy Recommendation Engine [33]) to the deployed edge device. To achieve this, an edge device has to request access to the ledger's contents and once the DLT network provides this access, the device is able to read and download the different policies and functions which are stored on the ledger and are described by smart contracts as the necessary input that needs to be provided to them so they can be executed.

The whole flow starts with an an edge device that is about to perform an attestation function (triggered by a smart contract) and then needs to provide the results back to that smart contract as explained in [28]. The edge device makes use of its embedded trusted TPM Wallet which is the one that will eventually retrieve the required functions and policies from the contract and will performs the attestation functions as required. In this line of events, the TPM wallet first executes the necessary function to get the list of attributes that need to be exhibited by any device trying to access this specific policy. Via this method, a request is sent to the Security Context Broker to access the designated attestation reports. The SCB retrieves a X.509 certificate and the request is evaluated by the Membership Service Provider which checks for the correctness of the attributes. If the attributes are correct, then the SCB is notified that the requesting device (to whom the TPM Wallet belongs) can indeed access the requested reports. At that point, the SCB retrieves the pointer to the block where the attestation report is located and is passing this to the TMP Wallet, which van then query the Private Ledger and knowing the pointer can download the report. For the aforementioned process, the necessary predicates and axioms are depicted in Tables 4.24 and 4.23, respectively.

Attestation Policy Download, Execution and Reporting axioms

(Ax1) $\forall$ Contract C, TPM t_i, $\mathbf{Trusted}_{TPM}(t_i) \wedge \mathbf{Certified}_{TPM}(T) \wedge \mathbf{CorrectAttributes}_{Host}(H, \mathcal{A}) \Leftrightarrow \mathbf{CorrectExec}(C)$
Any contract C is executed correctly if and only the Host H possesses the correct attributes $\mathcal{A}$ and TPM T is trusted, valid and certified.
(Ax2) $\forall$ Report R, $\mathbf{ContractContains}(C, N) \wedge \mathbf{CorrectExec}(C) \wedge \mathbf{CorrectSign}(R, K) \wedge \mathbf{ReportContains}(R, N) \Leftrightarrow \mathbf{FreshCorrectReport}(R)$
Any report R is considered fresh and correct if and only if contract C contains nonce N, Report R contains the same nonce N and is signed by K. Finally, contract C has to be executed correctly.
(Ax3) $\forall$ Contract C, $\mathbf{Certified}_{TPM}(T) \wedge \mathbf{ContractContains}(C, N) \wedge \mathbf{CorrectAttributes}_{Host}(H, \mathcal{A}) \Leftrightarrow \mathbf{CorrectlyPushed}(C)$
Any contract C is correctly pushed if and only if the pusher of the contract, H, has the correct attributes $\mathcal{A}$ and the contract contains a fresh nonce N.
(Ax4) $\forall$ Report R, $\mathbf{Certified}_{TPM}(T) \wedge \mathbf{ReportContains}(R, N) \wedge \mathbf{ReaderKnows}(H, N, C, R) \wedge \mathbf{CorrectAttributes}_{Host}(H, \mathcal{A}) \Leftrightarrow \mathbf{CorrectlyPushed}(C)$
Any report R is correctly read if and only if the reader H, has the correct attributes $\mathcal{A}$ and the report contains a nonce N, and the reader H knows this nonce N from contract C from which the report R originates from.

Table 4.23: Axioms for Downloading and Execution of Smart Contracts

Table 4.24: Smart Contract Download & Execution Predicates

Predicate	Predicate Meaning
$\mathbf{Certified}_{TPM}(T)$	TPM T is certified to be a valid TPM.
$\mathbf{CorrectAttributes}_{Host}(H, \mathcal{A})$	The host H has received the correct attributes $\mathcal{A}$ by the broker.
$\mathbf{CorrectExec}(C)$	A contract C has been decrypted executed correctly according to its policies.
$\mathbf{CorrectSign}(R, K)$	A report R has been correctly signed by K.
$\mathbf{ContractContains}(C, N)$	A contract C contains a valid fresh nonce N.
$\mathbf{ReportContains}(R, N)$	A report R contains a valid nonce N.
$\mathbf{FreshCorrectReport}(R)$	A report R is fresh and correctly signed.
$\mathbf{CorrectlyPushed}(C)$	A contract C is correctly pushed to the ledger.
$\mathbf{ReaderKnows}(H, N, C, R)$	A reader H of report R knows the nonce N from the contract C

Table 4.20: Axioms for Revocation

Axioms
(Ax1) $\mathbf{Trusted}_{TPM}(t) \wedge \mathbf{FinalPolicyTrusted}(\mathcal{P}, H) \wedge \mathbf{KeyPolicy}(K_p, PolicyNV(b_s \wedge b_h == 0), I_r) \wedge$ $\mathbf{RevocationHashesShared}(H, RA, K_p) \Leftrightarrow \mathbf{PseudonymRevocable}(K_p)$
A pseudonym K_p is revocable if the policy of the revocation index I_r is trusted ($\mathcal{P}$), the pseudonym is bound to a soft- and hard revocation bit (b_s, b_h) in revocation index I_r and the revocation hashes from the final policy has been shared with the RA.
(Ax2) $\forall K_p \in \mathcal{K} : \mathbf{PseudonymRevocable}(K_p) \wedge \mathbf{SharesHRB}(K_p, I_r) \Leftrightarrow \mathbf{PseudonymsAnonymouslyLinked}(\mathcal{K})$
A set of pseudonyms $\mathcal{K}$ is anonymously linked together if they are all revocable and share a hard revocation bit in the same index.
(Ax3) $\mathbf{Trusted}_{TPM}(t) \wedge \mathbf{KeyDestroyed}(K_{eph}) \wedge \mathbf{IndexPolicy}(I_a, PolicySigned(K_{eph})) \Leftrightarrow \mathbf{IndexImmutable}(I_a)$
Authorization Counter Index, I_a, is immutable if and only if the index is protected with $PolicySigned$ using ephemeral key K_{eph}, and that this key no longer exists or can be recreated.
(Ax4) $\mathbf{Trusted}_{TPM}(t) \wedge \mathbf{Sign}(K_{auth}, WriteIndex(0, I_r)) \wedge \mathbf{WriteIndex}(0, I_r) \Leftrightarrow \mathbf{IndexActivated}(I_r)$
Index I_r is activated if and only if the Authorization Key K_auth authorized a zero-write and that command is executed.
(Ax5) $\mathbf{Trusted}_{TPM}(t) \wedge \mathbf{IndexImmutable}(I_a) \wedge \mathbf{KeyPolicy}(K_{auth}, PolicySecret(I_a)) \wedge$ $\mathbf{IndexPolicy}(I_r, PolicyAuthorized(K_{auth})) \wedge \mathbf{IndexActivated}(I_r) \wedge \mathbf{Signed}(K_{auth}, \mathcal{P}) \wedge$ $\mathbf{AuthorizationsLeft}(I_a, 0) \Leftrightarrow \mathbf{IndexImmutable}(I_r)$
Revocation index I_r is immutable if the Authorization Counter Index I_a is immutable, and if the use of the Authorization Key $K_{auth}$is protected by $PolicySecret$ that depends on the authorizations left in I_a. Furthermore, I_r must be protected by $PolicyAuthorize$ with K_{auth}, meaning only signed policies by said key allows access to the index. Finally, the index must be activated, and by the time K_{auth} have signed the final policy $\mathcal{P}$ there must not be any authorizations left in I_a,
(Ax6) $\mathbf{Trusted}_{TPM}(t) \wedge \mathbf{IndexImmutable}(I_r) \wedge \mathbf{PseudonymRevocable}(K_p) \wedge \mathbf{Signed}(RA_{priv}, K_p(H_s)) \wedge$ $\mathbf{TrustedChannel}(RA, t) \wedge \mathbf{ExecutionSuccess}(SetBits(b_s, I_r)) \Leftrightarrow \mathbf{PsudonymSoftRevoked}(K_p)$
Pseudonym K_p is revoked if the revocation index I_r is immutable and the pseudonym is revocable. To be able to gain access to write to the index, the RA must sign the soft revocation hash H_s, and the write command must be broadcasted over a trusted channel to the TPM t. Only if the command executes correctly, the pseudonym is assumed revoked.
(Ax7) $\mathbf{Trusted}_{TPM}(t) \wedge \mathbf{IndexImmutable}(I_r) \wedge \mathbf{Signed}(RA_{priv}, K_p(H_h)) \wedge \mathbf{TrustedChannel}(RA, t) \wedge$ $\mathbf{ExecutionSuccess}(SetBits(b_s, b_h, I_r)) \wedge \mathbf{PseudonymsAnonymouslyLinked}(\mathcal{K})$ $\Leftrightarrow \mathbf{PsudonymHardRevoked}(\forall K_p \in \mathcal{K})$
All anonymously linked pseudonyms linked with K_p will be revoked if the revocation index I_r is immutable and the hard revocation hash H_h for K_p is signed by the RA and the execution of the two bits b_s and b_h are executed correctly.

Chapter 5

Cryptography for the ASSURED TCB

After having defined the trust models, capturing the complex device relationships in the envisioned application domains, that need to be achieved by the ASSURED framework using a Trusted Computing Base (TCB), as described in Section 3.3, in this chapter we proceed with a state-of-the-art analysis on the types of root-of-trust that we can consider based on the functional components they cover, the functionalities they offer (in terms of assurance claims throughout the entire lifecycle of a device) and any existing *security models* for their offered features. The endmost goal is to present the merits and challenges of both software- and hardware-based trusted computing technologies which led to the ASSURED decision on using the benefits of both worlds; hence, the adoption of a **TCB based on the use of a Trusted Execution Environment with the TPM as the hardware-based root-of-trust**. This design choice enables the concept of **Zero Trust security principle**, with the need of "*Never Trust, Always Verify*", for which ASSURED bootstraps vertical trust for all devices and users in the target SoS by enabling **continuous attestation, authorization and authentication** prior to be allowed to communicate and/or be granted to data or resources. This type of TCB allows the flexibility of been able to **guarantee the correctness of the tracing features of ASSURED (through the TEE)**, as the cornerstone of the entire framework correctness [29], and the correct **(self-) issuance of cryptographic material and verifiable credentials through the use of a TPM as the building block of the ASSURED TPM-based Wallet** [37].

We have to highlight that while ASSURED implementation will be instantiated based on the use of OP-TEE, as the underlying trusted execution environment [34], and TPM, all of the attestation models and secure data management schemes to be designed (as part of the WP3 and WP4 research activities) should be agnostic on the type of TCB to be considered.

5.1 Towards Mechanisms for Bootstrapping Trust with SW & HW-based Trust Anchors

Bootstrapping trust in an edge device (Prover) is achieved by convincing a remote device (Verifier) that a particular measurement chain represents a correct software and configuration state of the device. The Verifier's trust initiates from a hardware root-of-trust. Specifically, the Verifier has knowledge of the Prover's hardware and has access to a public key, which corresponds to the Prover's hardware root-of-trust. Given this public key, the Verifier is able to validate the measurement chain the Prover sends representing a valid state of the device's software and hardware, which in turn lets the Verifier trust the Prover. Next, we provide a state-of-the-art analysis of the mechanisms for a hardware Root-of-trust.

TPM. The TPM is a dedicated crypto-processor used to secure the hardware through integrated cryptographic keys and seeds. The Prover uses a TPM to generate an attestation measurement chain. The chain is then signed by the TPM with a unique Attestation Identity Key pair (AIK), which is an asymmetric key pair whose public component is known to the Verifier, but its private component is only accessible to the TPM. Since devices boot in stages, each loaded software is measured by the earlier loaded software component and extended in the TPM's Platform Configuration Registers (PCRs). To bootstrap trust in the device, the Verifier supplies the Prover with a nonce to ensure freshness, thereby preventing a replay of old measurements. The Prover asks the TPM to generate an attestation measurement, which covers the nonce and the PCRs. The Verifier can then validate the measurement is as expected, which means the Prover's software loaded correctly.

General-purpose cryptographic coprocessors. Coprocessors are a high-security, tamper-resistant, and programmable PCI board. Coprocessors use specialized cryptographic algorithms, and random number generators implemented in hardware and have dedicated memory, thereby providing a tamper-resistant environment for a highly secure subsystem. Coprocessors enable performing both data processing and cryptography operations. Unlike TPMs, coprocessors allow controlling the strength of the root-of-trust, for example, storing information about measurements made on the previous boot of the device, including software that may have been executed in an older version on the device.

Device Identifier Composition Engine (DICE). Many devices contain non-volatile memory, e.g., fuse banks, used to store cryptographic keys used as a basis for a unique device secret (UDS). This UDS is read by software code to be used as the device's hardware Root-of-trust for signing the measurement chains. State-of-the-art mechanisms are used to secure the UDS. First, power-on-latch allows only early boot software to access the UDS, which minimizes the TCB. Second, one-way cryptographic functions and key derivation algorithms are used to transform the UDS, so that if any code is compromised after the derivation of the keys, only the derived keys are compromised, making re-keying possible.

5.2 Trust Frameworks for Run-time Level of Assurance

In this section, we provide an overview of the state-of-the-art mechanisms that have been proposed in the literature towards achieving **run-time trust assurance** based on the use of TCBs comprising both TEEs and TPMs.

Due to increasing software TCB sizes, run-time trust assurance is of critical importance towards the verification of device security. Specifically, while it can be proven that the device contains specific hardware and is configured correctly, the loaded software may contain vulnerabilities, which may be exploited by an attacker to compromise the confidentiality or integrity of programs running in the device. The ever-increasing TCB sizes boost the probability of vulnerabilities existing in the loaded software on the devices. To that end, prior work has proposed mechanisms to achieve a level of trust towards software running on a device during runtime, such that a Verifier can continuously validate that a Prover's device is in a trustworthy state, even after the boot process has concluded. The two main techniques that have been proposed in the literature for runtime assurance are as follows:

5.2.1 TPM: Dynamic Root-of-Trust Measurement (D-RTM)

Trust is the expectation that a device will behave in a particular manner for a specific purpose. In order to establish trust in a property, we should be able to measure it and compare it against values that are known to be trustworthy. This level of trust is achieved in ASSURED by the Trusted Platform Module (TPM) that is intended to provide three roots of trust: a **root-of-trust for Measurement (RTM)**, a **root-of-trust for Storage (RTS)**, and a **root-of-trust for Reporting (RTR)**.

Static RTM (S-RTM) is a trusted implementation of a hash algorithm that is responsible for an initial measurement on a platform. This measurement can be done at a boot time or at a later time in order to establish trust in a platform. In the static root-of-trust method, all trust starts with a fixed piece of trusted code in the BIOS. This trusted piece of code measures the next piece of code to be executed, and extends a Platform Configuration Register (PCR) in the TPM based on this measurement, before control is transferred to the next program. If each new program, in turn, measures the next one before transferring control, a chain of trust is established, and the resulting PCR values will reflect the measurement of all files used. This **measurement before execution** model, therefore, leads to a chain of trust that is observable by a remote party, that aims to assess the trustworthiness of a system. Hence, S-RTM enables trust on the entire boot chain, including the master boot record, boot loader, kernel, drivers, and all files referenced or executed during boot.

Dynamic RTM (D-RTM), where trust for an extended period of time needs to be measured while some running code may change due to updates. D-RTM may occur while the hardware platform is running and without a hardware platform restart. In contrast, the Static root-of-trust for Measurement (S-RTM) requires a platform shutdown or restart [58]. The D-RTM implementations introduce a new CPU instruction family, which creates a controlled and attested execution environment. During D-RTM, the platform can jump directly into a trustworthy state, a ”measured launched environment.” That explains the ”dynamic” part of the D-RTM since the launching of a protected environment can happen at any time without the need for rebooting. The term **dynamic Root-of-trust** refers to approaches for providing evidence for a trustworthy platform state at more arbitrary points in time, in addition to the device start-up. This mechanism will be adopted by ASSURED to bring runtime trust assurance for software executing on devices, so that a Verifier can continuously validate that a Prover’s device is in a trusted state, even after the boot process has concluded.

5.2.2 Trusted Execution Environments (TEEs): Isolated Execution

Trusted execution environments (TEEs) are designed to guarantee the authenticity of the executed code, the integrity of the runtime states (e.g., CPU registers, memory, and sensitive I/O), and the confidentiality of its code, data and runtime states stored on persistent memory. In addition, it shall be able to provide a remote attestation that proves its trustworthiness for third parties. The content of TEE is not static, as it can be securely updated. The TEE resists all software attacks, as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible [61].

Combined with a hardware Root-of-trust to bootstrap trust in the initial state of the device (Section 5.1), It can be proven to a Verifier that a device has TEE-enabled hardware, was configured as expected, and that software was loaded to be executed inside a TEE. This provides a runtime level of assurance, since the processor provides strong isolation of code executing inside the TEE

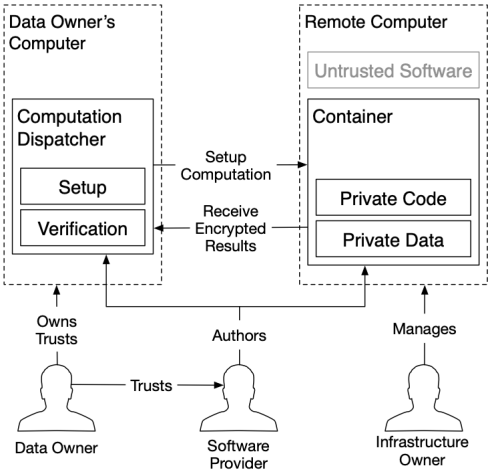

Figure 5.1: Secure remote computation. A user uses a remote computer for performing computations on their data. Confidentiality and integrity has to be guaranteed. Credit to [38].

from the rest of the device such that external software cannot compromise the code executing inside TEEs.

It is possible to distinguish between two different families of TEEs:

1. **Enclave TEEs**: sensitive data are inside of a secure area, separated from the untrusted operating system. These can belong to the following categories:
   - x86 Systems:
     - Intel Security Guard eXtension (SGX);
     - AMD Secure Encrypted Virtualization (SEV);
   - RISC-V Systems:
     - Sanctum;
     - Keystone;
     - CURE;
     - Penglai.
2. **Non-enclave TEEs**: sensitive data is not stored inside a particular area of the system, so its security and integrity has to be guaranteed in a different way. These can belong to the following categories:
   - TPM (Trusted Platform Module);
   - SecureZone.

5.2.2.1 Intel Security Guard eXtension (SGX)

Originally introduced by [8, 46, 54], SGX is an extension to the Intel architecture that aims to provide integrity and confidentiality guarantees to security-sensitive computation performed on a computer where all the privileged software (kernel, hypervisor, etc.) is potentially malicious [38], in particular in a context of remote execution.

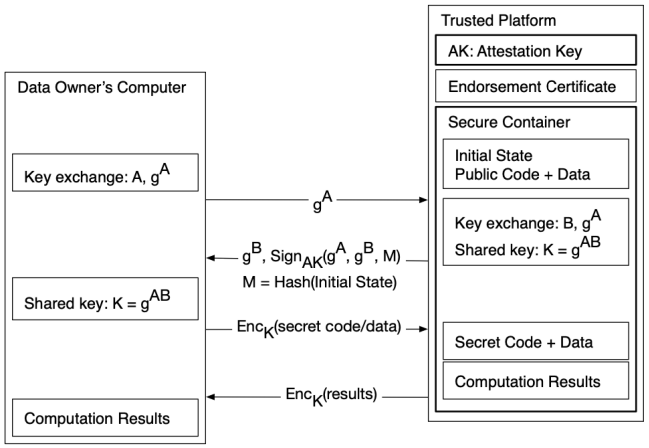

Figure 5.2: Attestation schema followed in SGX. Credit to [38].

SGX is process-oriented and relies on software attestation. Attestation proves to a user that it is communicating with some software running in a secure container hosted by the trusted hardware. The proof is a cryptographic signature that certifies the hash of the secure container's contents (Figures 5.1 and 5.2). In case this hash does not match, the user will refuse to send data.

Intel SGX can use different remote attestation schemes: EPID (*Intel Enhanced Privacy Identifier*) and DCAP (*Intel Data Attestation Primitives*) [62]. Intel ECDSA Attestation (*Elliptic Curve Digital Signature Algorithm*) enables third-party attestation via DCAP. It supports Intel Xeon Scalable processor (at least of 3rd generation) and some selected Intel Xeon E3 Processors. This method provides flexible provisioning based on ECDSA certificates, allow on-premise attestation services, requires flexible launch control in supported Intel platforms, and it is available an open-source licensing model.

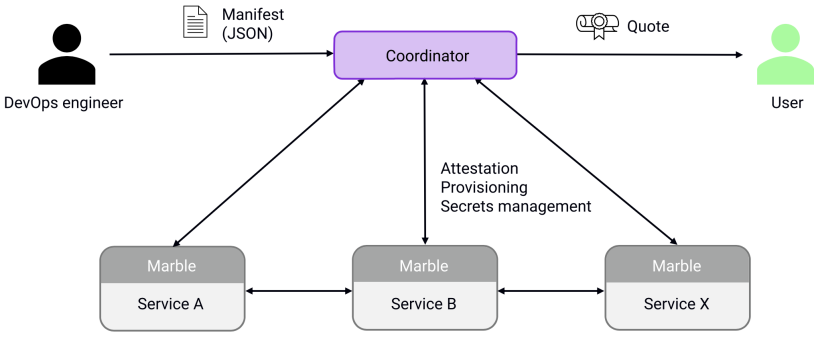

Figure 5.3: MarbleRun attestation schema. Credit to [70].

In the context of SGX, the following schemes can also be adopred:

- **MarbleRun** (Gramine Attestation Service Mesh): A library for Linux multi-process applications, with Intel SGX support [69]. It is a service mesh for confidential computing from Edgeless Systems [68]. MarbleRun operations are divided in two planes [70] (See Figure 5.3):
  - Control Plane, the Coordinator: it is the centralized attestation and who offer the secret provisioning service, deployed in the cluster;
  - Data Plane, the Marbles: separates Gramine applications, integrated with each service.

- **OPERA** (*Open Remote Attestation for Intel's Secure Enclaves*) [24]: This scheme addresses the limitations of the Intel-centric attestation model. It is designed to achieve openness, security and performances. Figure 5.4 depicts a high-level overview of the attestation process.

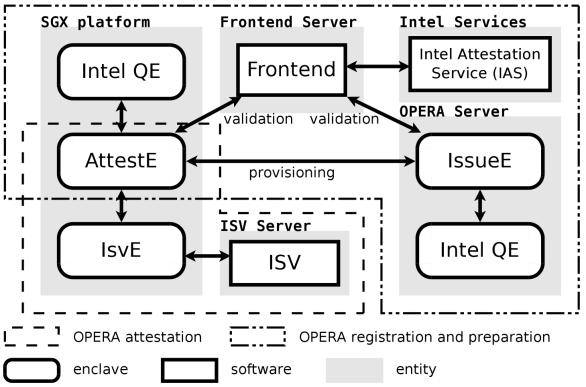

Figure 5.4: Opera attestation schema. Credit to [24]

5.2.2.2 AMD Secure Encrypted Virtualization (SEV)

Instead of being process oriented, as Intel SGX, AMD SEV [17] is virtual machine oriented, specially designed for AMD EPYC processors [48].

SEV encrypts the main memory of virtual machines with VM-specific keys, thereby denying the higher-privileged hypervisor access to a guest's memory. To enable the cloud customer to verify the correct deployment of his virtual machine, SEV additionally introduces a remote attestation protocol: it is a crucial component of the SEV technology that can prove that SEV protection is in place and that the virtual machine was not subject to manipulation.

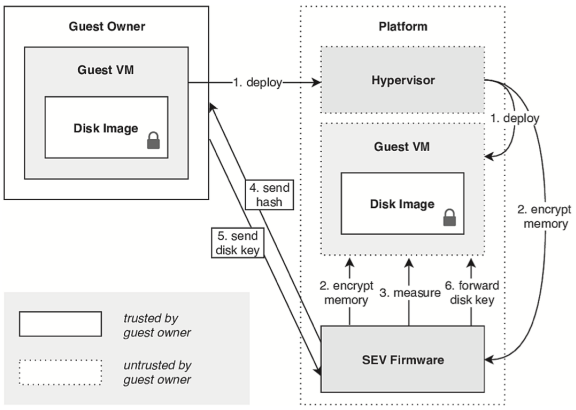

Figure 5.5: Deployment of a guest VM in SEV scenario. Credit to [17].

5.2.2.3 Keystone: An Open-Source Secure Enclave for RISC-V

Keystone [40, 42, 51, 52] was born from the necessity of having a non-closed-source commercial enclave. Intel SGX, AMD SEV, and ARM TrustZone require commercial hardware.

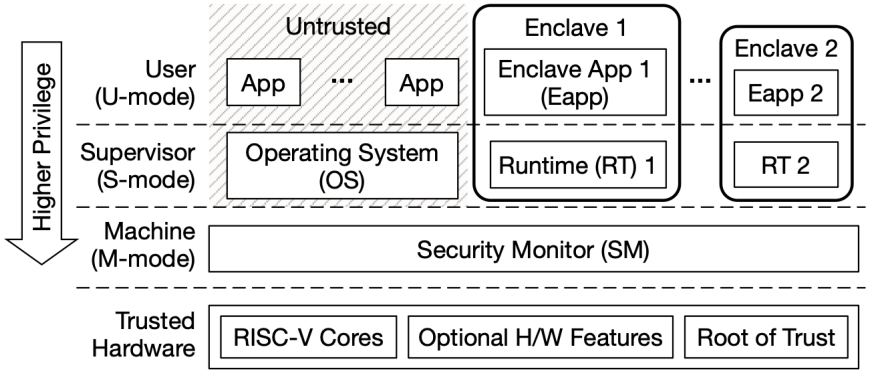

Figure 5.6: Keystone system with host processes, untrusted OS, security monitor, and multiple enclaves. Credit to [51]

Keystone is a full-stack open-source enclave (depicted in Figure 5.6) with minimal requirements, which can achieve memory isolation only with standard RISC-V primitives (RISC-V privileged ISA U-, S- and M-mode support) using PMP (Physical Memory Protection). It is built in a modular and portable manner for easy extension.

Keystone started from the experience of Sanctum [39], the first enclave design in RISC-V ISA. These two projects share many good practices, and the main goal of Keystone is to have an open end-to-end framework. Furthermore, Keystone uses Sanctum's root-of-trust, which uses ECDSA and SHA-3.

5.3 Security Modelling for TPM (and TEE)

We finally review existing (partial) security models for the TPM and other TEEs. We organise this review by the functional components they cover, then by the threat and security model they consider. We focus our review on literature that presents precise *security models* for the functionalities they analyze.

5.4 Cryptography, Storage and Key Management

At the core of the TPM lies its collection of cryptographic primitives, which both rely on and support its key hierarchies–including its storage hierarchy. This section reviews models and analyses of these two components.

Shao, Feng and Qin [63] develop a type-based framework to analyse the security of the TPM's Protected storage API in the same scenario where authorization is not used. They use it to show that (part of) the TPM API cannot be misused to extract from the TPM the value of keys whose FixedTPM attribute is set. The framework is defined to provide symbolic security guarantees, and is limited to a very small subset of the TPM's commands.

Zhang et al. [76] use Tamarin to model a large subset of the TPM's key management API, including key duplication and import, and search for proofs of symbolic secrecy and integrity for duplicated and imported blobs. They find simple attacks. In particular, they note that the raw TPM mechanisms cannot provide identification of the destination on duplication, or of the origin of a blob import, which allows for breaks in authentication and secrecy. These attacks, rather than

identifying a break in the TPM specification, illustrate the need to consider fine-grained security policies in the security models of TPM components.

Wang et al. [73] describe a formal game-based model of the TPM's cryptographic support commands in CryptoVerif, and a CryptoVerif-based proof of the fact that honestly-generated keys can be used to securely encrypt or authenticate messages, even if the adversary can otherwise interact with the TPM, including to create his own keys and to request encryptions under user keys (thereby modelling a worst-case scenario). Their model does not allow duplication, and considers a single TPM (and therefore does not consider problems related to the migration of keys).

5.5 Sessions and (Enhanced) Authorization

If the articles discussed above focus on modelling the core properties of the TPM's hierarchies, they abstract away the TPM's authorization mechanisms, which enforce usage control on TPM-protected objects. Instead, the articles above place themselves in a pessimistic scenario where all keys can be used by the adversary. If this is a good abstraction to analyse the security of the protected storage mechanism in isolation, it does very little to support the verification of *applications* that make use of that protected storage. In particular, such an abstraction lacks the flexibility needed to allow controlled usage of cryptographic functionalities in larger protocols. We now review existing models of the TPM's authorization mechanisms, for TPM1.2 and 2.0.

Chen and Ryan [26], in the first attempt at defining security for (part of) the TPM, model TPM1.2's authorization mechanisms and find that a naive treatment of the authentication data (authdata) used to access objects can lead to complete usurpation of the TPM's secure storage in multi-tenant scenarios, even when encrypted sessions are used. They also prove using ProVerif symbolic security properties (authentication and secrecy) of a modified version of the protocol, which is used as part of TPM2.0's Enhanced Authorization mechanism. Delaune et al. [41] propose more general notions of authentication for TPM1.2's session mechanism as properties of its API, and show that they are sufficient to capture known attacks and verify fixed APIs.

Wang, Qin and Feng [72] formalize TPM2.0's HMAC-based authorization sessions in CryptoVerif and prove that they provide the expected authenticity properties: that the TPM only executes protected commands when called by a user who possesses the appropriate secret, and that callers engaged in a protected sessions with a TPM can trust that execution of the commands indeed occurred on the TPM upon receiving results. The contributions are limited to only HMAC-based authorization sessions, and abstract away much of the complexity due to the TPM's session mechanisms being used for various purposes.

Shao et al. [64] produce a SAPIC model for a larger subset of TPM2.0's Enhanced Authorization mechanism, which includes a large subset of the policies (including PCR-based authorization, policies based on counters stored in NVRAM, signature-based authorization), but abstracts some of the object management away. They use Tamarin to obtain proofs of symbolic authentication for most modes of authorization, and identify some cases where misuse is possible and needs to be managed by careful usage of the TPM's API.

5.6 Direct Anonymous Attestation

Direct Anonymous Attestation is the only cryptographic functionality that was developed specifically for the TPM, which otherwise reuses standard cryptography wherever possible–although

sometimes in novel and somewhat untested ways. As such, DAA–taken in isolation–has been the focus of much more attention than the rest of the TPM, leading to the development and refinement of complex models for its security. We describe this evolution below.

Brickell, Camenish and Chen [15] propose a security model of DAA, as well as its RSA-based realization for TPM1.2. The proposed model is game-based, and is therefore difficult to use in security proofs for larger systems that may involve parallel DAA instances, or compositions. A ProVerif-based analysis by Backes, Maffei and Unruh [12], which considers the protocol's security properties (and in particular the authentication guarantees it is meant to provide) finds possible attacks on the join protocol, whereby anonymous credentials can be delivered to the wrong TPM.

A proposal for ECC-based DAA [16]–a version of which was to be included in the TPM2.0 specification–was formalized in ProVerif and its privacy properties analyzed in the symbolic model under an additional symbolic assumption by Smyth, Ryan and Chen [66, 67]. This symbolic analysis was among the main drivers for the development of equivalence-based notions of symbolic security–needed to express strong secrecy properties (equivalent to IND-CPA) and privacy properties such as anonymity and unlinkability. This analysis, which focuses on privacy properties for which the host is trusted, is performed on a model of the cryptography that does not consider the fact that DAA operations in the real TPM are split between the Host and the TPM itself. Xi and Feng [75] formalize the TPM2.0 DAA-related APIs in ProVerif and verify that all expected properties of the API are indeed met. They also propose a new notion of privacy–*forward anonymity*–and show that, although it is not met by the API as specified–a small modification to the API enhances the protocol to meet it. In particular, this failure of forward anonymity is in fact related to a weakness that prevents reductions to the more standard assumptions [20].

However, these positive symbolic security results–by the very nature of the models they are obtained in–fail to capture some important realistic attacks. Indeed, Bernard et al. [14] show that existing models of security for DAA–including new simulation-based models introduced to analyze the ECC-based version of Brickell, Chen and Li [16]–can be used to deem secure protocols that are fully insecure. Bernard et al. [14] propose game-based security models for pre-DAA–usable in a setting where the Host is trusted. A more flexible and realistic trust model is recovered by Camenisch, Drijvers and Lehmann [21], who propose a UC-based security model for Direct Anonymous Attestation. This shift to UC-based security models further enabled refinements of the trust model, such as the consideration of the effect of compromised TPMs on the security of uncompromised TPMs [22]. Camenisch et al. [20] propose modifications to the specification of DAA in TPM2.0 that improves the overall security of TPM2.0 DAA (by weakening the assumption on which the proof relies), and prove their security–including forward anonymity–in a UC-based setting.

Chapter 6

Challenges in TPM Modelling

In this chapter, we consider and discuss the challenges we will face in modelling the security, privacy and trustworthiness for the ASSURED framework as a whole based on the design of the TPM-based Wallet [37], as well as those we will face in modelling the security of protocols that involve multiple TPMs.

6.1 Trust Assumptions

Generally speaking, the TCG works under the assumption that TPMs should be trusted to withstand any software attack. From a security modelling point of view, this means that no TPM could ever be compromised by a software-based adversary (for example, by revealing its primary seed). In practice, however, TPMs will be deployed in situations where an adversary can mount physical attacks–including side-channel and fault attacks–and locally compromise one or several TPMs. It is important to ensure that such an adversary is not able to break the security or privacy properties of uncompromised TPMs. It is therefore important for us to consider various models of trust, especially when analysing the security of functionalities that involve multiple TPMs. In particular, we will need to consider the following models:

- The TPM is completely trusted;
- The TPM is partially trusted;
- The TPM is subverted.

Such models have recently been considered in formal analyses of the ECC-DAA protocols standardized in TPM 2.0.

6.2 Split Operations and Untrusted Hosts

The TPM is a low-cost and relatively slow chip, and has limited resources. Therefore, some operations, including cryptographic operations, are split between the TPM and the software of its host platform (referred to as a Host). Further, the TPM relies entirely on the Host to communicate with remote partners (for example, the privacy-CA or a remote attestation partner). This has a further effect on security models, and requires careful consideration of the models to enable fine-grained modelling of trust both for the TPM (as justified above) and the Host (which may be a compromised platform). For example, the DAA signing operation is split between the TPM and

the Host, with the TPM being trusted for both security and privacy, but the host being trusted only to preserve its own privacy.

Defining security models for TPM functionalities will therefore require careful consideration of the roles played by different parts of the TPM (including its hardware component and various components of the TSS), and careful consideration of the appropriate trust assumptions on each of them.

6.3 State, Command and Key Sharing

High-level TPM functionalities often require multiple calls to TPM commands. Due to its limited resources, the TPM itself rarely maintains state between calls to the commands. This has in the past been a source of vulnerabilities–that are made harder to find and analyze due to hidden invariants. Modelling such split functionalities may require a thorough understanding of these hidden invariants and of all expected uses of the TPM commands being captured by the model.

Since a single command could appear in multiple high-level functionalities–including sharing not only code, but also cryptographic material–analyzing the security of the TPM *as a whole* is critical. In particular, local models for specific functionalities do retain value but need to be carefully reworked to allow their use in more complex scenarios where part of the computation and the limited state may be shared with other functionalities that are being accessed concurrently.

6.4 Flexible and Secure Usage Policies

The TPM serves as hardware support for secure software systems, and is therefore meant to be used by external parties. If some of its security properties *must* hold independently of the usage that is made of it (in particular, properties of the Roots of Trust), properties of user objects stored on the TPM, or of interactions between users and the TPM, can only be guaranteed under certain conditions, including assumptions of trust in the user (for a particular scenario), and assumptions that a secure policy of interaction with the TPM was defined and followed. At both extremes, modelling security of the TPM is straightforward: a TPM that forbids any external interaction is easy to model and most certainly secure, but prevents any rigorous analysis of the security user applications; at the other end of the spectrum, a TPM that allows all interactions is also relatively easy to model, but provides only very limited security guarantees. Ensuring that our security models are both *realizable* in practice and *usable* to support security analysis for applications that use the TPM will require the careful definition of families of models parameterized by usage policies.

6.5 Multi-Tenant Security

In practice, TPMs–especially hardware TPMs used to back security in virtualized environments–will be multi-tenanted, and used by multiple users concurrently. Each of these users may have a different trust and authorization relationship with the TPM, including various authorization policies and external authentication factors. Capturing the security requirements of multi-tenanted scenarios into cryptographic security models is a complex challenge that currently remains open.

Chapter 7

Conclusions

In this deliverable, we provided a formal description of the trust models to be achieved in ASSURED, which consists of a set of security predicates that are used in order to formulate a set of security axioms. **These provide the basis for the formulation of the trust assurance methodologies, so that a set of security and privacy requirements is fulfilled**. The presented models capture the full range of relationships between assets and devices, including representation, isolation and interaction relationships, in an efficient and verifiable manner.

We also defined the hardware required in order to formulate these models, as well as the functionality that needs to be provided in terms of cryptographic functions by the underlying trusted component. We then defined the implemented functionalities in terms of encryption, in order to fulfill our requirements of secure and privacy-preserving communication between all involved parties of the system, as well as functionalities related to key management, key distribution, attribute-based access, and key revocation.

From all the above, it follows that in this deliverable, we have provided a **comprehensive model that encompasses all the employed attestation and security services**, which in turn fulfill all the security and privacy requirements for all the devices that comprise a complex System-of-Systems, while utilizing the functionalities provided by the underlying trusted component available in each of the considered devices.

List of Abbreviations

Abbreviation	Translation
AE	Authenticated Encryption
ABE	Attribute-based Encryption
AK	Attestation Key
CA	Certification Authority
CFA	Control-flow Attestation
CIV	Configuration Integrity Verification
CSR	Certificate Signing Request
DAA	Direct Anonymous Attestation
DLT	Distributed Ledger technology
EA	Enhanced Authorization
EK	Endorsement Key
GSS	Ground Station Server
MSPL	Medium-level Security Policy Language (MSPL)
NMS	Network Management System
Privacy CA	Privacy Certification Authority
Prv	Prover
PCR	Platform Configuration Register
PLC	Program Logic Controller
RA	Risk Assessment
RAT	Remote Attestation
SCB	Security Context Broker
SoS	Systems of Systems
SSR	Secure Server Router
S-ZTP	Secure Zero Touch provisioning
TC	Trusted Component
TLS	Transport Layer Security
TPM	Trusted Platform Module
Vf	Virtual Function
VM	Virtual Machine
Vrf	Verifier
WP	Work Package
ZTP	Zero Touch Provisioning

References

[1] ://rdist.root.org/2007/07/17/ tpm-hardware-attacks-part-2/.

[2] Lawson nate, tpm hardware attacks. https://rdist.root.org/2007/07/16/tpm-hardware-attacks/, 2007. Accessed: 2022-02-24.

[3] Denis andzakovic. extracting bitlocker keys from a tpm. https://pulsesecurity.co.nz/articles/TPM-sniffing, 2019. Accessed: 2022-02-24.

[4] Arm trustzone. https://www.arm.com/technologies/trustzone-for-cortex-m, 2022. Accessed: 2022-02-24.

[5] Ibm's tpm 2.0 tss. https://sourceforge.net/projects/ibmtpm20tss/, 2022. Accessed: 2022-02-24.

[6] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-flat: Control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 743–754, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Alessandro Aldini. A formal framework for modeling trust and reputation in collective adaptive systems. *arXiv preprint arXiv:1607.02232*, 2016.

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7. Citeseer, 2013.

[9] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, USA, 1st edition, 2015.

[10] Will Arthur, David Challener, and Kenneth Goldman. *A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security*. Springer Nature, 2015.

[11] N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. Seda: Scalable embedded device attestation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 964–975, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Michael Backes, Matteo Maffei, and Dominique Unruh. Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 202–215, 2008.

[13] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 178–183, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[14] David Bernhard, Georg Fuchsbauer, Essam Ghadafi, Nigel P. Smart, and Bogdan Warinschi. Anonymous attestation with user-controlled linkability. *Int. J. Inf. Sec.*, 12(3):219–249, 2013.

[15] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*, pages 132–145, 2004.

[16] Ernie Brickell, Liqun Chen, and Jiangtao Li. Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *Int. J. Inf. Sec.*, 8(5):315–330, 2009.

[17] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. Insecure until proven updated: Analyzing amd sev's remote attestation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1087–1099, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, page 90–105, Berlin, Heidelberg, 2003. Springer-Verlag.

[19] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. volume 6617, pages 459–465, 04 2011.

[20] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. One TPM to bind them all: Fixing TPM 2.0 for provably secure anonymous attestation. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 901–920, 2017.

[21] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, pages 234–264, 2016.

[22] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Anonymous attestation with subverted tpms. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, pages 427–461, 2017.

[23] M. Carbone, M. Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *First International Conference onSoftware Engineering and Formal Methods, 2003.Proceedings.*, pages 54–61, Sep. 2003.

[24] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. Opera: Open remote attestation for intel's secure enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2317–2331, 2019.

[25] Liqun Chen, Ming-Feng Lee, and Bogdan Warinschi. Security of the enhanced tcg privacy-ca solution. In Roberto Bruni and Vladimiro Sassone, editors, *Trustworthy Global Computing*, pages 121–141, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[26] Liqun Chen and Mark Ryan. Attack, solution and verification for shared authorisation data in TCG TPM. In *International Workshop on Formal Aspects in Security and Trust*, volume 5983 of *LNCS*, pages 201–216, Eindhoven, The Netherlands, November 2009. Springer.

[27] Liqun Chen and Bogdan Warinschi. Security of the tcg privacy-ca solution, 12 2010.

[28] The ASSURED Consortium. Assured blockchain architecture. Deliverable D4.1, November 2021.

[29] The ASSURED Consortium. Assured layered attestation and runtime verification enablers design implementation - version 1. Deliverable D3.2, November 2021.

[30] The ASSURED Consortium. Assured reference architecture. Deliverable D1.2, August 2021.

[31] The ASSURED Consortium. Assured use cases & security requirements. Deliverable D1.1, February 2021.

[32] The ASSURED Consortium. Operational sos process mddels specification of properties. Deliverable D1.3, November 2021.

[33] The ASSURED Consortium. Policy modelling cybersecurity, privacy and trust policy constraints. Deliverable D2.2, November 2021.

[34] The ASSURED Consortium. Assured real-time monitoring and tracing functionalities. Deliverable D3.4, February 2022.

[35] The ASSURED Consortium. Assured runtime risk assessment framework - version 1. Deliverable D2.3, February 2022.

[36] The ASSURED Consortium. Assured secure distributed ledger maintenance data management. Deliverable D4.2, February 2022.

[37] The ASSURED Consortium. Assured tc-based functionalities. Deliverable D4.5, February 2022.

[38] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[39] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016.

[40] Kevin Cheang Cameron Rasmussen Kevin Laeufer Ian Fang Akash Khosla Chia-Che Tsai Sanjit Seshia Dawn Song Dayeol Lee, David Kohlbrenner and Krste Asanovic. Keystone enclave: An open-source secure enclave for risc-v. Keystone Enclave presentation at RISC-V Summit https://keystone-enclave.org/files/keystone-risc-v-summit.pdf.

[41] Stéphanie Delaune, Steve Kremer, Mark D Ryan, and Graham Steel. A formal analysis of authentication in the TPM. In *International Workshop on Formal Aspects in Security and Trust*, volume 6561 of *LNCS*, pages 111–125, Pisa, Italy, September 2010. Springer.

[42] Keystone Enclave. Keystone enclave. Keystone Enclave website https://keystone-enclave.org.

[43] Georgios Fotiadis, José Moreira, Thanassis Giannetsos, Liqun Chen, Peter B. Rønne, Mark D. Ryan, and Peter Y. A. Ryan. Root-of-trust abstractions for symbolic analysis: Application to attestation protocols. In Rodrigo Roman and Jianying Zhou, editors, *Security and Trust Management*, pages 163–184, Cham, 2021. Springer International Publishing.

[44] Munirul M Haque and Sheikh I Ahamed. An omnipresent formal trust model (FTM) for pervasive computing environment. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 49–56. IEEE, 2007.

[45] James Hendricks and Leendert van Doorn. Secure bootstrap is not enough: Shoring up the trusted computing base. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, page 11–es, New York, NY, USA, 2004. Association for Computing Machinery.

[46] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370, 2013.

[47] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. Darpa: Device attestation resilient to physical attacks. In *Proceedings of the 9th ACM Conference on Security amp; Privacy in Wireless and Mobile Networks*, WiSec '16, page 171–182, New York, NY, USA, 2016. Association for Computing Machinery.

[48] AMD Inc. Amd secure encrypted virtualization (sev). AMD Inc. Developer Documentation, retrieved from https://developer.amd.com/sev/.

[49] Nikos Koutroumpouchos, Christoforos Ntantogian, Sofia-Anna Menesidou, Kaitai Liang, Panagiotis Gouvas, Christos Xenakis, and Thanassis Giannetsos. Secure edge computing with lightweight control-flow property-based attestation. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 84–92, 2019.

[50] Benjamin Larsen, Thanassis Giannetsos, Ioannis Krontiris, and Kenneth Goldman. Direct anonymous attestation on the road: Efficient and privacy-preserving revocation in c-its. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, New York, NY, USA, 2021.

[51] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[52] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: An open framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[53] Michael E Locasto, Steven J Greenwald, and Sergey Bratus. Trust distribution diagrams: Theory and applications. In *Proceedings of the 4th Layered Assurance Workshop (LAW 2010), Austin, TX*, 2010.

[54] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[55] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used rsa moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1631–1648, New York, NY, USA, 2017. Association for Computing Machinery.

[56] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. volume 42, pages 89–100, 06 2007.

[57] Bryan Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, pages 9:1–9:6, Berkeley, CA, USA, 2008. USENIX Association.

[58] Sandeep Romana, Himanshu Pareek, and PR Lakshmi Eswari. Dynamic root of trust and challenges. *Int. J. Secur. Privacy Trust Manag.*, 5:01–06, 2016.

[59] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. NIST Special Publication 800-207, Zero Trust Architecture. 2020. Available online at https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf.

[60] Grigore Roşu, Wolfram Schulte, and Traian Florin ŞerbănuŢă. Runtime verification of c memory safety. In Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification*, pages 132–151, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[61] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015.

[62] Beaney Zmijewski Scarlata, Johnson. Supporting third party attestation for intel sgx with intel data center attestation primitives. Intel Corportation Document, retrieved from https://www.intel.com/content/dam/develop/external/us/en/documents/intel-sgx-support-for-third-party-attestation-801017.pdf.

[63] Jianxiong Shao, Dengguo Feng, and Yu Qin. Type-based analysis of protected storage in the TPM. In *International Conference on Information and Communications Security*, volume 8233 of *LNCS*, pages 135–150, Beijing, China, November 2013. Springer.

[64] Jianxiong Shao, Yu Qin, Dengguo Feng, and Weijin Wang. Formal analysis of enhanced authorization in the TPM 2.0. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 273–284, Singapore, April 2015. ACM New York.

[65] Sergei Skorobogatov. *Physical Attacks and Tamper Resistance*, pages 143–173. Springer New York, 2012.

[66] Ben Smyth, Mark Ryan, and Liqun Chen. Formal analysis of anonymity in ECC-based Direct Anonymous Attestation schemes. In *International Workshop on Formal Aspects in Security and Trust*, volume 7140 of *LNCS*, pages 245–262, Leuven, Belgium, September 2011. Springer.

[67] Ben Smyth, Mark D Ryan, and Liqun Chen. Formal analysis of privacy in Direct Anonymous Attestation schemes. *Science of Computer Programming*, 111:300–317, 2015.

[68] Edgeless Systems. Edgeless systems. Edgeless Systems website https://www.edgeless.systems.

[69] Edgeless Systems. Gramine library os with intel sgx support. Gramine Project repository, retrieved from https://github.com/gramineproject/gramine.

[70] Edgeless Systems. Marblerun developer docs. MarbleRun Developer Documentation, retrieved from https://docs.edgeless.systems/marblerun/.

[71] TCG. *TCG PC Client Platform TPM Profile Specification for TPM 2.0*. Trusted Computing Group, 2020.

[72] Weijin Wang, Yu Qin, and Dengguo Feng. Automated proof for authorization protocols of TPM 2.0 in computational model. In *International Conference on Information Security Practice and Experience*, volume 8434 of *LNCS*, pages 144–158, Fuzhou, China, May 2014. Springer.

[73] Weijin Wang, Yu Qin, Bo Yang, Yingjun Zhang, and Dengguo Feng. Automated security proof of cryptographic support commands in TPM 2.0. In *International Conference on Information and Communications Security*, volume 9977 of *LNCS*, pages 431–441, Singapore, November 2016. Springer.

[74] Johannes Winter. Trusted computing and local hardware attacks. *Master's thesis, Graz University of Technology*, 2014.

[75] Li Xi and Dengguo Feng. Formal analysis of DAA-related APIs in TPM 2.0. In *International Conference on Network and System Security*, volume 8792 of *LNCS*, pages 421–434, Xi'an, China, October 2014. Springer.

[76] Qianying Zhang, Shijun Zhao, Yu Qin, and Dengguo Feng. Formal analysis of TPM2.0 key management APIs. *Chinese Science Bulletin*, 59(32):4210–4224, August 2014.