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

ASSURED

# D3.2: ASSURED LAYERED ATTESTATION AND RUNTIME VERIFICATION ENABLERS DESIGN & IMPLEMENTATION

Revision: v.1.0

| | |
|---|---|
| **Work package** | WP 3 |
| **Task** | Task 3.2 |
| **Due date** | 30/11/2021 |
| **Deliverable lead** | TUDA |
| **Version** | 1.0 |
| **Authors** | Richard Mitev (TUDA), Philip Rieger (TUDA) |
| **Reviewers** | Liqun Chen, Nada El Kassem (SURREY)<br>Meni Orenbach (MLNX) |
| **Abstract** | Deliverable D3.2 design and develops novel runtime attestation and verification schemes that covers a device's whole lifecycle, from boot-time integrity measurement to runtime control-flow and information-flow attestation of those safety-critical components, and extend the single-device attestation to establish chains of trust across multiple devices |
| **Keywords** | Trusted Computing, Configuration Integrity Verification, Control-Flow Attestation, Swarm Attestation, Direct Anonymous Attestation, Jury-based Attestation |

## Document Revision History

| Version | Date | Description of change | List of contributors |
|---|---|---|---|
| v0.1 | 15.08.2021 | ToC | Jingru Wang (TUDA) |
| v0.2 | 10.09.2021 | Description of the overall flow of attestation actions in ASSURED based on the created architecture (Chapter 2) | Edlira Dushku, Benjamin Larsen (DTU)<br>Thanassis Giannetsos (UBITECH)<br>Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA)<br>Richard Mitev, Philip Rieger (TUDA)<br>Liqun Chen, Nada El Kassem (SURREY) |
| v0.3 | 30.09.2021 | Description of the high-level conceptual overview of the attestation schemes (Chapter 3) | Edlira Dushku, Heini Bergsson Debes, Benjamin Larsen (DTU)<br>Richard Mitev, Philip Rieger (TUDA)<br>Liqun Chen, Nada El Kassem (SURREY)<br>Thanassis Giannetsos (UBITECH)<br>Ioannis Avramidis (INTRA)<br>Ilias Aliferis (UNIS)<br>Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA) |
| v0.4 | 15.10.2021 | Description of the ASSURED tracing mechanism plus the Control-flow Attestation and Direct Anonymous Attestation (Chapter 3) | Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA)<br>Richard Mitev, Philip Rieger, Marco Chilese (TUDA)<br>Liqun Chen, Nada El Kassem (SURREY)<br>Dimitris Papamartzivanos (UBITECH) |
| v0.5 | 29.10.2021 | Description of the Swarm Attestation, Configuration Integrity Verification and Jury-based Attestation (Chapter 3) | Edlira Dushku, Heini Bergsson Debes(DTU)<br>Ioannis Avramidis (INTRA)<br>Ilias Aliferis (UNIS)<br>Richard Mitev, Philip Rieger, David Koisser (TUDA)<br>Thanassis Giannetsos (UBITECH) |
| v0.6 | 12.11.2021 | Description of the Revocation protocol (Chapter 6) plus finalization of the system (Chapter 2) and adersarial models (Chapter 4) | Benjamin Larsen (DTU)<br>Thanassis Giannetsos (UBITECH)<br>Ahmad Atamli, Meni Orenbach (MLNX/NVIDIA)<br>Richard Mitev, Philip Rieger, Marco Chilese, David Koisser (TUDA)<br>Liqun Chen, Nada El Kassem (SURREY) |
| v0.7 | 29.11.2021 | Finalization of the security analsysis of all attestation schemes (Chapter 7) | Edlira Dushku, Heini Bergsson Debes, Benjamin Larsen (DTU)<br>Richard Mitev, Philip Rieger, Marco Chilesse, David Koisser (TUDA)<br>Liqun Chen, Nada El Kassem (SURREY)<br>Thanassis Giannetsos (UBITECH) |
| v0.8 | 10.12.2021 | Refinements on the positioning and use of the attestation schemes in relation to the overall ASSURED Architecture | Thanassis Giannetsos (UBITECH), Richard Mitev, Philip Rieger (TUDA) |
| v0.9 | 13.12.2021 | Review the document | Liqun Chen, Nada El Kassem (SURREY)<br>Meni Onrebach (MLNX) |
| v1.0 | 20.12.2021 | Finalisation of the document | Richard Mitev, Philip Rieger (TUDA) |



Funded by the Horizon 2020
Framework Programme of the European Union

**Editors**

Richard Mitev (TUDA), Philip Rieger (TUDA)

**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)

Riccardo Orizio (UTRCI)

Ilias Aliferis (UNIS)

Ioannis Avramidis (INTRA)

## 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.

# Executive Summary

Deliverable D3.2 provides the ASSURED security, privacy and trust extensions enhanced with **secure remote attestation capabilities for verifying the configurational attestation policies and properties** as well as Direct Anonymous Attestation for privacy-preserving and accountable services and Jury-based Attestation for acting as a second line of defense in case of a misbehavior detected during the execution of the the attestation process per se.. More specifically, it presents a new set of attestation protocols for supporting trust aware service graph chains with verifiable evidence on the integrity assurance and correctness of the comprised platforms. It is the core step towards the provision of a **secure supply chain ecosystem for delivering the high-level functionalities related to secure device identification and integrity, data integrity and confidentiality, anonymity and resource integrity** as described in the overall framework reference architecture (see Deliverable D1.2 [24]).

In order to support enhanced **system and network trust assurance**, ASSURED has defined the security protocols that are necessary for providing a range of **secure attestation services in order to support verifiable evidence on the correct configuration state and/or execution of a remote platform** targeting the software layer and covering all phases of a device's execution. To do so, ASSURED designs a **layered attestation toolkit** leveraging purely sw-based tracing and introspection capabilities, enhanced remote attestation mechanisms and advanced cryptographic primitives for guaranteeing the required privacy requirements. For the former, ASSURED provides execution stream monitoring and introspection capabilities necessary for tracing the control- and information-flow execution paths, needed by the run-time attestation, overcoming the current limitations of state-of-the-art mechanisms as it pertains to instrumenting the traced binary which results to affecting the normal operation of the target application. When it comes to remote attestation, ASSURED orchestrates the integration of techniques for not only protecting the **secure enrollment of devices** and their **run-time operation** but also acting as a **second line of defense** when inconsistencies are identified between the results reports by proving and verifying devices and further investigation is needed for detecting who is lying.

Furthermore, we also proceed with a formal analysis of the offered security and assurance guarantees based on the trust model and properties defined in D3.1 [21]. This constitutes the first step towards the definition of formal models capable of capturing the security properties for the **execution assurance of a single system** (remote attestation), and how these can be transferred to statements on the **security properties of compositions of systems** ("*Systems-of-Systems*" through swarm attestation) but also the **communication assurance between interacting systems**. For this purpose, we have introduced the notion of an *idealized functionality*. This is a model of a TPM (as the underlying trusted component) that captures the actions of the trusted platform module when the command is executed in such a way that it excludes the cryptographic operations carried out internally (e.g., hash functions, signature creation, encryption), as perfectly secure, and focuses on the non-cryptographic operations related to key management, key usage and/or the internal configuration registers used for depicting the state of the host device. We essentially developed a **trusted abstract platform model consisting of a specific set of formally-specified primitives sufficient to implement the core attestation functionalities beyond the core crypto operations**. Such an abstraction modelling will enable the formal security reasoning (to be documented in the next version of this deliverable) under various adversarial models and different security guarantees, excluding any possible implications from the leveraged cryptographic primitives.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Towards Decentralized Roots of Trust in Supply Chain Ecosystems

Seeking to design successful supply chain service management and various IoT applications comprising millions of autonomous cyber-physical systems, one has to cater to the **security, trust and privacy requirements** of all involved actors (i.e., smart connected edge and cloud devices). As described in D1.2 [24], a key challenge that ASSURED tries to resolve is to establish and manage trust between entities, starting from bi-lateral interactions between two single system components and continuing as such systems get connected to ever larger entities. *But how can we make sound statements on the security properties of single systems and transfer this to statements on the security properties of hierarchical compositions of systems ("Systems-of-Systems" (SoS))?*

This pressing need for establishing **federated trust** between services and devices, in a complex supply chain ecosystem, cannot be solely secured with common centralized solutions like Public Key Infrastructures (PKIs). Recent research [3, 36] has demonstrated the need to move towards decentralized federated safety critical systems that aim to establish roots-of-trust in intelligent edge devices by leveraging software- or hardware-based trusted computing mechanisms. Particularly with respect to **safety and operational assurance**, software components must be enabled to prove statements about their state so that other components can align their actions appropriately and an overall system state can be assessed and verified.

Towards this direction, the main goal of this deliverable is to present the first release of the ASSURED Collective Attestation enablers towards fulfilling the main vision of the project for the establishment and management of **trust- and privacy-aware service graph chains**. In this context, *the communication over the continuum from edge devices to gateways (e.g., fog nodes) and back end cloud systems must support secure interactions between all participating entities in order to establish* ***service-specific "node communities of trust".*** This is considered as one of the main goals towards **"security and privacy by design"** solutions, including all methods, techniques, and tools that aim at enforcing security and privacy at software and system level from the conception and guaranteeing the validity of these properties.

ASSURED will achieve **high security, privacy and trustworthiness guarantees** - throughout the entire lifecycle of a device - by employing advanced **remote attestation mechanisms**, as a central building block for the **trusted exchange of data as well as for secure device management**. In a nutshell, remote attestation mechanisms operate on a network that comprises thousands of low-end collaborating edge devices that work together to support a safety-critical

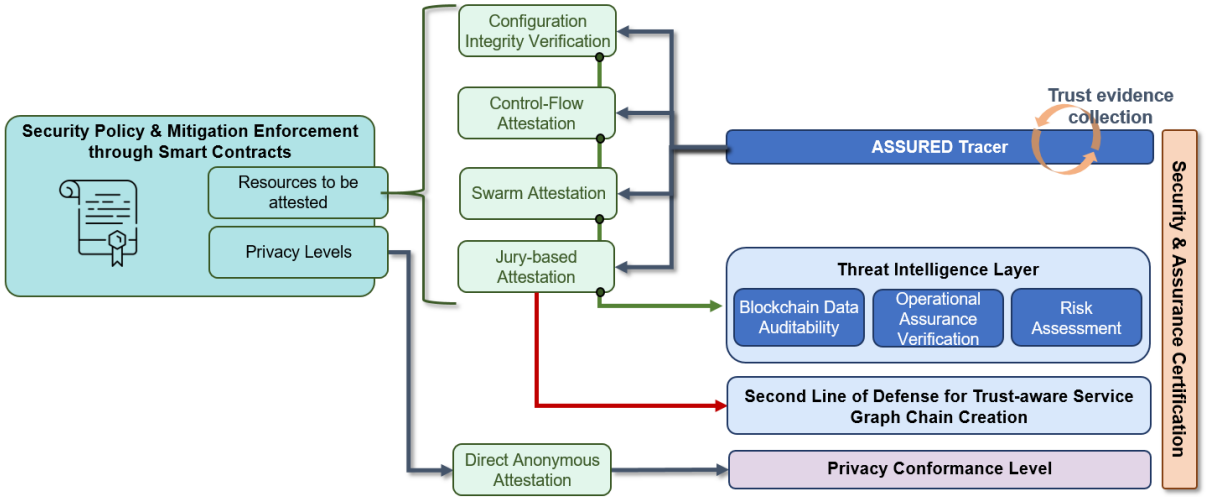



Figure 1.1: ASSURED Security Process Toolchain

decision process based on measurements received from many deployed actuators (i.e., edge devices) [24]. In this context, the underlying protocols should not only be able to handle all the messages originating from these devices but also actuators need to verify that all platforms from which they receive data are uncompromised (integrity) while also having the minimum possible performance impact.

By designing novel, **highly efficient attestation schemes**, we aim to convert edge devices to trust anchors capable of providing verifiable evidence for their correct configuration and execution covering both (privacy-preserving, if needed) **device authentication purposes** (secure enrollment of a device wanting to join an SoS) and all phases of a device's execution; from the **trusted boot and integrity measurement** of a CPS, enabling the generation of static, boot-time or load-time evidence of the system's components correct configuration, to the **run-time behavioral attestation** of those safety-critical components of a system providing strong guarantees on the correctness of the control- and information-flow properties (as modelled in D1.3 [26]), thus, enhancing the performance and scalability when composing secure systems from potentially insecure components. *The goal is to prove to any remote party that an edge device, its configuration and running services are intact and trustworthy* - the latter is achieved through the secure recording, sharing and auditing of all attestation reports (allowing also for certification purposes) leveraging ASSURED's policy-compliant Blockchain infrastructure [22].

To do so, ASSURED designs a **layered attestation toolkit** leveraging purely software-based tracing and introspection capabilities (Section 3.1), enhanced remote attestation mechanisms and advanced cryptographic primitives for guaranteeing the required privacy requirements. For the former, ASSURED provides execution stream monitoring and introspection capabilities necessary for tracing the control- and information-flow execution paths, needed by the run-time attestation, overcoming the current limitations of state-of-the-art mechanisms as it pertains to instrumenting the traced binary which results to affecting the normal operation of the target application. When it comes to remote attestation, ASSURED orchestrates the integration of techniques for not only protecting the **secure enrollment of devices** and their **run-time operation** but also acting as a **second line of defense** when inconsistencies are identified between the results reports by proving and verifying devices and further investigation is needed for detecting who is lying.

Figure 1.1 depicts the flow of actions of all ASSURED Attestation Schemes as part of the overall

| Attestation Scheme | Functionality |
|---|---|
| Configuration Integrity Verification (CIV) (Section 5.3) | Mechanism for verifying the correct configuration (i.e., list of loaded operational binaries) of an edge device. This is usually invoked during the **secure enrollment** of a device into a network or periodically during run-time when an **update** takes place in the device configuration (e.g., *software update*). |
| Control-Flow Attestation (CFA) (Section 5.1) | Mechanism for verifying the execution behavior of an edge device based on the attestation policies deployed. ASSURED enables **control-flow-based attestation**, focusing on the execution verification of only those safety-critical software functions, leveraging Deep Neural Networks. |
| Swarm Attestation (Section 5.4) | As an alternative to the attestation of a single device, ASSURED also considers the parallel attestation of a swarm of devices featuring the same processes to be verified. |
| Direct Anonymous Attestation (DAA) (Section 5.2) | Besides operational assurance, one has to cater for a number of properties like anonymity, pseudonymity, unlinkability, and unobservability and the strict trust requirements of a wide variety of multi vendor devices and platforms. ASSURED leverages DAA for the provision of **privacy-preserving and accountable authentication services** leveraging group signatures. |
| Jury-based Attestation (Section 5.5) | ASSURED Jury-based Attestation scheme (Section 5.5) acts as a second line of defense towards detecting possible misbehaviors during the execution of a (run-time) remote attestation process; either Configuration Integrity Verification or Control-flow Attestation. Essentially detect any outliers during the execution of an attestation task. |

Table 1.1: ASSURED Proposed Designed Attestation Schemes

security process: ASSURED leverages security- and privacy-based Risk Assessment methodology [28] for identifying the most severe attacks and vulnerabilities per asset in the overall ecosystem that can affect the level of trustworthiness of the entire system. Based on such a risk dependency graph, the ASSURED Policy Recommendation engine then extracts the list of optimized attestation policies [27], depicting the set and order of execution of attestation tasks per asset so as to achieve the required level of security and safety, which are then enforced as smart contracts through the ASSURED Blockchain infrastructure [22] (**Security Policy & Mitigation Enforcement throgh Smart Contracts**). Based on such "*attestation contracts*", the respective devices then execute the correct set of attestation tasks leveraging the input from the underlying **ASSURED Tracer**. Once the attestation process finishes, the (binary) result is recorded on the Distributed Ledger Technologies (DLTs) in order to enable secure sharing as well as auditing and certification of the entire attestation mechanism (**Threat Intelligence Layer**). On top of that, based on the privacy requirements of a safety-critical task, the executing device might choose to protect the necessary features through the adoption of the DAA-based anonymous credentials.

Furthermore, the employed attestation strategy should enable the network to operate undisturbed, for long periods of time, and automatically recover from failures (as a result of potential attacks) without manual intervention. This implies that an attestation protocol should be robust and sustain its security service in case of a device disruption especially since these devices are usually deployed in wide and uncontrolled areas where physical and/or remote compromise is

much easier than typical computer systems.

Addressing these challenges lies in the heart of ASSURED towards the provision of a **secure overlay mesh network for delivering the high-level functionalities related to secure (edge) device identification and integrity, data integrity and confidentiality, anonymity and resource integrity** as described in the overall framework reference architecture (see Deliverable D1.2 [24]).

## 1.2 Scope and Purpose

The main purpose of this deliverable is to **present the first release of the ASSURED Attestation Toolkit enhanced with integrity and execution verification and DAA trust extensions for attesting configurational and behavioural properties of a deployed edge node**. More specifically, it documents the functionalities and underlying crypto operations of all designed attestation schemes (Table 1.1) focusing on ensuring not only on the trust level of each device (against the conceptual model of trustworthiness defined in D2.2 [27]) but also the strong trust relations that must be established among interacting entities. This includes all the interactions with the host TPM-based Wallet, as the root-of-trust for providing authenticity and integrity of the traces used throughout the attestation process as well as the interaction with the Security Context Broker (SCB) and the DLTs for downloading the "*attestation contracts*" to be executed. Each attestation functionality is coupled with detailed sequence diagrams per operational phase depicting also the commands that the underlying TPM-based Wallet needs to offer.

Furthermore, we also proceed with a formal analysis of their offered security and assurance guarantees based on the trust model and properties defined in D3.1 [21]. This constitutes the first step towards the definition of formal models capable for capturing the security properties for the **execution assurance of a single system** (remote attestation), and how these can be transferred to statements on the **security properties of compositions of systems** ("*Systems-of-Systems*" through swarm attestation) but also the **communication assurance between interacting systems**. For this purpose, we have introduced in [43] the notion of an *idealized functionality*.

## 1.3 Relation to other WPs and Deliverables

In what follows, Figure 1.2 depicts the relationship of this 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, formally define and evaluate the ASSURED secure remote attestation toolkit and operational stack. Thus, within WP3, this deliverable takes input from D3.1 [21] as it pertains to the overall trust model and detail axioms defined for depicting the security, privacy and trust requirements that each attestation scheme needs to achieve. Based on these axioms, we proceed with the security analysis put forth in Chapter 7. In addition, based on the design of all ASSURED attestation mechanisms, this provides the baseline for the description of the tracing capabilities towards introspecting and tracing the necessary system properties (D3.4 and D3.5) as well as the integration of such remote attestation schemes in the context of swarm attestation for verifying the operational and configuration correctness of a swarm of devices simultaneously (D3.6 and D3.7).

The outcome of Deliverable D3.2 is intended to support the definition of later activities in the project. In relation with the rest of the WPs of the project, D3.2 serves as a point of reference

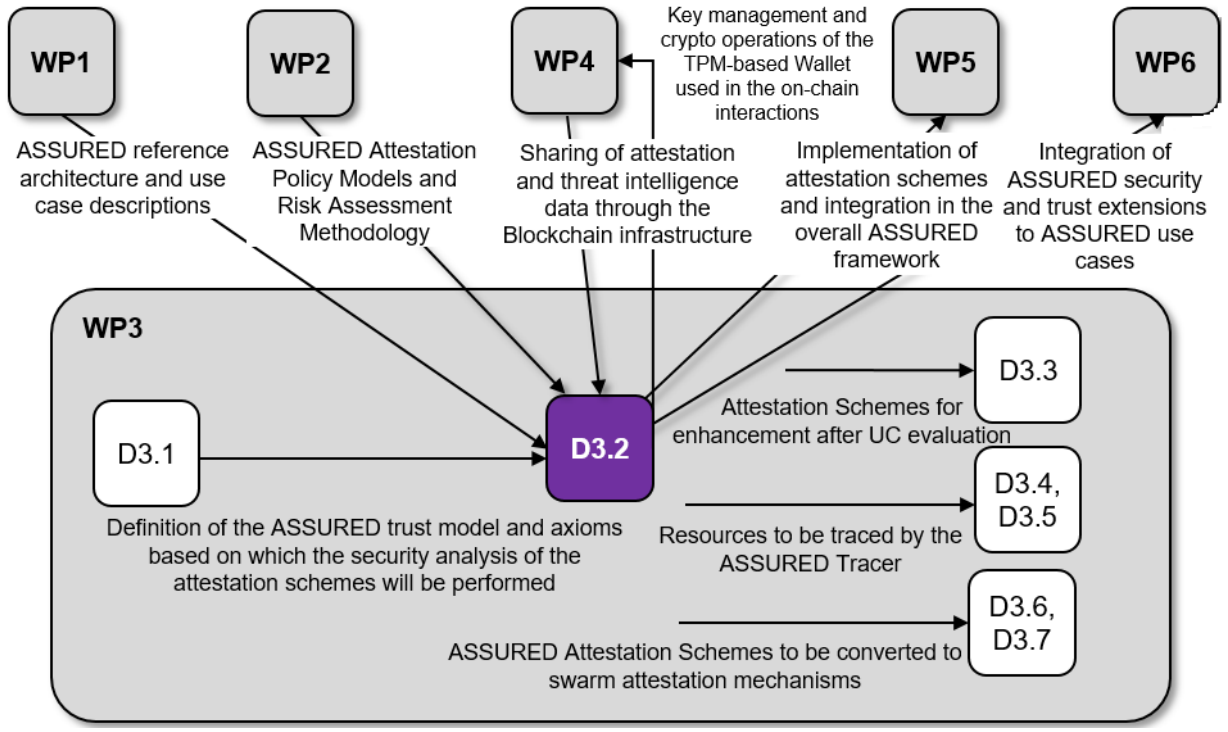



Figure 1.2: Relation of D3.2 with other WPs and Deliverables

for the technical developments of the project as it offers a set of directions to each WP. More specifically, D3.2 provides the description of the operation of each attestation scheme, the outcome of which needs to be recorded, audited and shared through the policy-compliant Blockcain infrastructure designed in WP4. Last but not least, WP5 that undertakes the development of the integrated framework and WP6 that aims to its validation in the context of the pilots, inherit the baseline of the overall systems and the high-level interactions that need to be validated.

## 1.4 Deliverable Structure

This deliverable is structured as follows: In **Chapter 2** we describe the ASSURED System Model comprising the building blocks and hardware and software components, including discussion about the choice of the architecture. In **Chapter 3.2** we provide a high-level description of each attestation mechanism and their positioning in the overall ASSURED ecosystem. Then, in **Chapter 4**, we summarize the adversarial model considered in ASSURED (based on the detailed description provided in D1.3 [26] and D2.1 [28]) before we proceed to the detailed description of the models, workflow of actions and crypto operations per attestation mechanism in Chapter 5. To complete the operational landscape of ASSURED, in Chapter 6 we describe a revocation mechanism that enables us to revoke the credentials of a misbehaving entity/device in a supply chain ecosystem. In Chapter 7, we continue with a formal analysis of the security guarantees every attestation scheme can provide to the ASSURED framework and which attacks can be defended by single mechanisms of a combination of more than one. Finally, Chapter 8 concludes the deliverable.

# Chapter 2

# System Model

## 2.1 ASSURED Attestation Building blocks

Based on the aforementioned vision, ASSURED relies on two core pillars: **remote attestation of specific properties** and **enforcement of dynamically adaptable policies** depicting the optimized set of attestation tasks that need to be executed in each edge device towards achieving the required level of overall security and safety. With this, we claim that an SoS can withstand even a prolonged siege by a pre-determined attacker with known or unknown capabilities (Chapter 4) as the system can dynamically adapt to its security and safety state. This is substantially more flexible than traditional security mechanisms that often try to maintain and enforce a pre-defined set of policies using static attestation mechanisms (**zero-trust paradigm**).

Policy-based security management is an administrative approach for simplifying access control and security management of networks, services, etc. by establishing policies. Policies, in ASSURED, are sets of attestation rules, usually in the form "*on event, if condition, then action*", that reflect the resource owner's intention of adequately protecting valuable resources and are a means for **enforcing different levels of trustworthiness** based on the overall conceptual model of trustworthiness, for complex ecosystems, as defined in D2.2 [27]. Once the policies have been extracted by the ASSURED Policy Recommendation engine, these are then pushed to the respective devices through the Blockchain infrastructure; i.e., they are converted to smart contracts so as to be able to record and audit the correct execution of each attestation tasks executed by the target device [22].

It is imperative that we are able to express policies that: (*i*) when enforced, mitigate the risks of the safety and security critical systems we wish to compose, (*ii*) regard properties that can be attested by the resource constrained embedded components, (*iii*) safeguard the privacy of attesting devices by specifying the general principles for attestation data protection, and (*iv*) specify the type of evidence to be collected from a system, in case it fails to attest some of its properties, so as to perform a more in-depth investigation of the system's behaviour towards detecting if any type of malware is resident (D1.3 [26]). Defined policies must be expressive, deployable, and enforceable and may be dynamically updated if the attack graph is amended with new types of vulnerabilities.

After the correct definition of such policies, the system can proceed to periodically (or on-demand) attest the modelled configuration and execution properties that mainly include either **configuration digests** (i.e., list of correct loaded binaries) or **execution paths** to specific memory regions, as a result of the invocation of the functions of interest and their control-flow. All correct execution paths (i.e., Control Flow raphs) need to have been identified and securely deployed to the

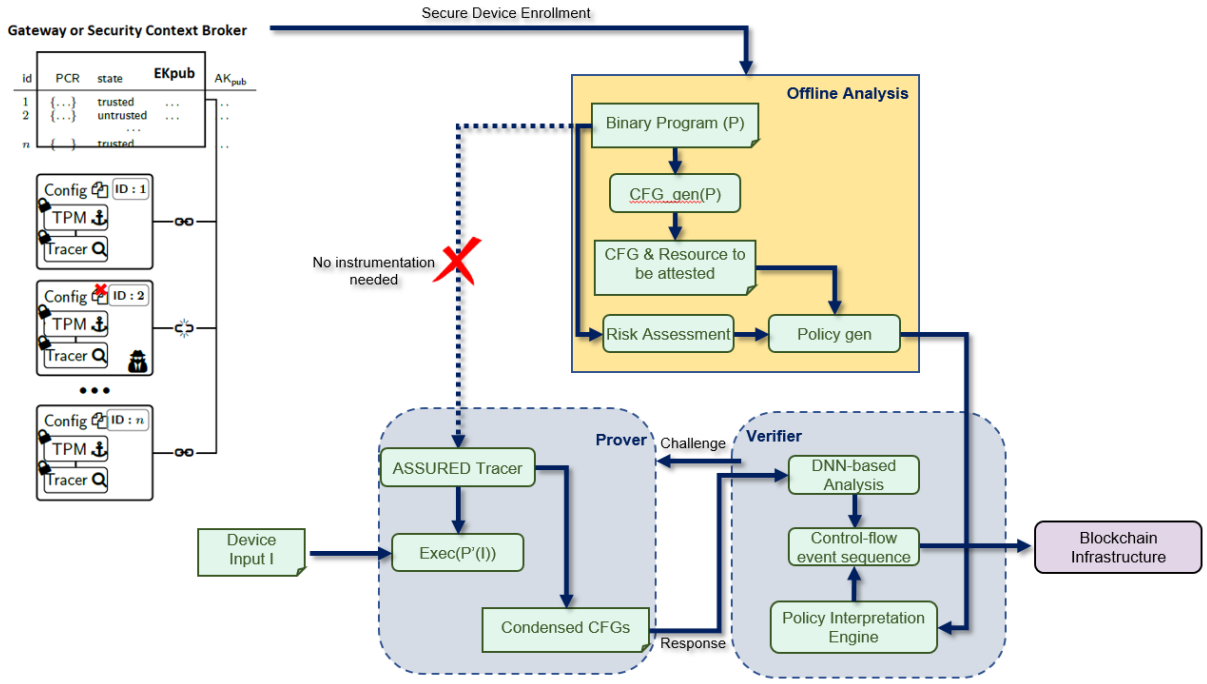



Figure 2.1: ASSURED Design- and Run-time Attestation & Integrity Verification

verifiers so as to act as the baseline of the normal sequence of states against which the run-time computed control-flow footprints will be assessed.

Before delving into the underpinnings and crypto operations performed by each attestation scheme (**Configuration Integrity Verification (CIV), Control-flow Attestation, Swarm Attestation, Direct Anonymous Attestation (DAA) and Jury-based Attestation**), presented in Chapter 5, we first provide a simple description of the information flow (Figure 2.1):

**Secure Enrollment:** As an entry point in the ASSURED operational assurance toolchain (upper left part in Figure 2.1), is the support for *secure enrollment* of a device in the overall ecosystem of "Systems-of-Systems" through the provision of **zero-touch configuration functionalities** (Section 5.3): *platforms, wishing to join a cluster, adhere to the compiled attestation policies by providing verifiable evidence on their configuration integrity and correctness*. For instance, consider the context of our envisioned "*Smart Manufacturing* use case" [25] where the $IndustrialPLC$, acting as the IoT Gateway of a specific manufacturing floor, wants to securely on-board a *sensor system* for estimating the location of a worker so as to trigger a $STOP$ event for the operation of a robotic arm in the case where a worker is getting closer to the robot arm. In other words, ASSURED provide guarantees that a node will be able to join a network (and participate in the underlying operations and services) ***if and only if*** it can prove to the Gateway that it is at a "correct state" - without, however, the Gateway needing to know the node's state beforehand. This enables us to support the secure enrollment and integration of heterogeneous devices and platform equipped with different computing resources and operating systems. The Gateway will be able to execute the appropriate attestation policy extracted through the appropriate smart contract leveraging its TPM-based Wallet [22] while the newly joining device needs to also be equipped with a certified root-of-trust; a TPM in our case but any hw- or sw-based trusted component would suffice.

**Run-time Attestation:** After been securely enrolled, each device in the target ecosystem can be periodically asked to attest its correct configuration or behavioural execution. For the latter, in ASSURED, we employ **property-based control-flow attestation** (Section 5.1) as a means

of operational assurance for a software resource of interest. For instance, continuing in the same example scenario as before, every time a *sensor system PLC* sends a $STOP$ event, this needs to be coupled with verifiable evidence on the correct execution flow of the specific function (e.g., $locationCalculation$) that yielded this event so as to have strong integrity guarantees on the correctness of the data based on which the safety-critical decision was made. The type of resources to be attested have already been identified by the Policy Recommendation Engine, modelled as attestation policies [27] and enforced as smart contracts [22].

**Swarm Attestation:** As an alternative to the attestation of a single device, ASSURED also considers the parallel attestation of a **swarm of devices** featuring the same processes to be verified. For instance, consider the case, of an administrator wanting to attest all of the devices in a specific manufacturing floor. Instead, of initiating remote attestation processes with each one of the deployed edge devices, a single process can be triggered for attesting the same properties of all devices in a privacy-preserving manner. A conceptual overview of swarm attestation is given in Section 5.4 while all of the details and workflow of actions will be documented in D3.6 [29].

**Direct Anonymous Attestation:** Besides operational assurance, one has to cater for a number of properties like anonymity, pseudonymity, unlinkability, and unobservability and the strict trust requirements of a wide variety of multi vendor devices and platforms. Essentially, in the case of a device that prior to sharing either operational or attestation data with other devices or the backend infrastructure, needs to have strong guarantees on their privacy protection; i.e., their ID or location will not be infringed. In this context, ASSURED leverages Direct Anonymous Attestation (DAA) (Section 5.2) as an enabler for the provision **privacy-preserving and accountable authentication services**. DAA is based on group signatures that give strong anonymity guarantees to a device that wishes to share its data.

**Jury-based Attestation:** ASSURED Jury-based Attestation scheme (Section 5.5) acts as a second line of defense towards detecting possible misbehaviors during the execution of a (run-time) remote attestation process; either Configuration Integrity Verification or Control-flow Attestation. Consider, for instance, the case where the Verifier (e.g., IoT Gateway) has been compromised and, thus, records a falsified attestation report for a new device trying to enroll to the network. This essentially will result in a negative output while executing the "*Secure Device Enrollment*" policy. However, this sets the challenge ahead: *How do we make sure of the correctness of the Verifier that is responsible for attesting a newly joined device?* If the Verifier is compromised, when the attestation result is been broadcasted back to the Blockchain Peer Node, the Prover will overhear the message and report to the SCB of the wrong result. In this case, we need to have an additional step for identifying which of the participating entities is lying: **verify the evidence provided by both the Prover and Verifier so as to detect and revoke the misbehaving entity**.

### 2.1.1 ASSURED Innovations

Besides the overall innovation of ASSURED in **orchestrating the use of various attestation schemes**, in tandem, towards enhancing the overall level of trustworthiness of a complex environment and creating privacy- and trust-aware service graph chains (thus, enabling the vision of shifting trust from the infrastructure to the edge), we also target specific innovations in the internal operation of these trusted computing building blocks. More specifically:

#### 2.1.1.1 Configuration Integrity Verification (CIV)

There are many proposals in the literature (cf. Chapter 4) that focus on verifying the correct configuration of a platform with related specifications providing the foundational concepts such as *measured boot* and *remote attestation*. However, many of the existing families of attestation solutions have strong assumptions on the verifying entity's trustworthiness, thus not allowing for privacy-preserving integrity correctness. Furthermore, they suffer from scalability and efficiency issues. It should be difficult for any (possibly compromised) Verifier to infer any meaningful information on the state or configuration of any of the devices comprising the service graph chain or wishing to enroll to a network. In this context, it is essential to ensure not only the security of the underlying host and loaded software processes but also their privacy and confidentiality - *an attacker should not be able to infer any information on the configuration of any of the binaries loaded in the same node*.

This dictates for an *oblivious* theme of building trust for such SoS where a Prover can attest all of its components in a **zero-touch manner** without the need to reveal specific configuration details of its software stack [33]. For instance, suppose that a Prover runs a Python interpreter. The Prover may wish not to reveal that it runs version 2.7.13 of the CPython implementation. One option would be to introduce ambiguity about the software stack components (e.g., by having the Prover only reveal that it has a CPython implementation), thus making it harder for an eavesdropper to exploit zero-day vulnerabilities in the Prover's code directly. However, an even stronger claim is to have the Prover not reveal *anything*, which would make it *impossible* for Verifiers to infer *anything*. However, this sets the challenge ahead: *How can a Prover prove its integrity correctness without disclosing any information about its software stack's configuration?* (Section 5.3).

ASSURED's overarching approach is to have a centralized entity (e.g., Security Context Broker (SCB)) that translates the received policies in what constitutes correct configuration of a device and what not, and then have that party setup appropriate cryptographic material (i.e., restrained attestation key templates) on each newly joined node in the network. The ability to then use such restrained keys is physically "locked" from the node until the node can prove its correctness - supply correct measurements that will "unlock" its usage. Once released, the node can use the key to sign nonces, acting as verifiable statements about its state so that other components can align their actions appropriately and an overall system state can be accessed and verified. Similarly, if Verifiers receive no response or the signature is not produced using the key initially agreed upon and advertised by the centralized entity, they can justifiably assume that the Prover is untrusted. Note that Verifiers need only to know that the Prover is in an authorized state, not what that state is. However, one main challenge of such approaches is the strong link between the restrained cryptographic material and the specific Configuration Integrity Verification (CIV) policies: Whenever an updated policy must be enforced, due to a change to the configuration of the overall system, a new attestation key must be created [63]. Managing and updating such symmetric secrets creates a *key distribution problem* which in ASSURED is alleviated through the enforcement of key update operations by smart contract functions as part of the deployed attestation policies [22].

#### 2.1.1.2 Control-Flow Attestation (CFA)

A core limitation of CIV-based solutions is that they do not ensure the integrity of the software's execution during run-time and, therefore, cannot capture attacks that target the program's control flow [44, 81]. These types of attacks are considered the most devastating since they try to exploit

memory- and data-related vulnerabilities (cf. Chapter 4) for altering the execution path of the underlying system processes; either by injecting new malicious code [32] or by dynamically generating malicious programs based on already existing benign code snippets. As a consequence, they can bypass the security of static attestation techniques since the measurement of a binary can remain unchanged even though the software's behaviour has been altered.

Compounding this issue, more advanced dynamic *control-flow attestation* solutions have been proposed that can protect against run-time exploitation techniques by steering away from static measurements and aim to check software behaviour; software that is running as expected by verifying the integrity of the entire control flow. While a number of research efforts have proven the security and trust guarantees provided by such approaches, there are still a number of challenges to be conquered especially when it comes to the efficiency, scalability and robustness of these techniques that question whether they can be applied in the real-world resource-constrained edge devices.

Such limitations mainly stem from the fact that these types of operational assurance methods try to verify the integrity, during run-time, of the **entire (untrusted) code base** of commodity platforms and operating systems. Considering that competitive IIoT application markets will always produce innovative and *large* systems comprising diverse-origin software-based components, with uncertain security properties, the best one can hope for is that a sub-set of such loaded software functions can be efficiently protected (in near real-time) against sophisticated run-time exploitation attacks.

This exact goal sets the challenge ahead: *Can we identify adequate behavioural and execution properties that can capture the chains-of-trust, needed for the correct execution of a system, and that reflect the security- and safety-critical code widgets to be verified from the untrusted code of the commodity platform or the cloud service provider?* This will, in turn, enable the provision of control-flow attestation of only these specific, critical software components that are comparatively small, simple and limited in function, thus, allowing for a much more efficient verification and safe co-existence.

ASSURED answers this question by leveraging **control-flow property based attestation**: The insight behind this approach is that we do not need the attestation of the entire device but only the execution properties of the security sensitive functionalities that are running on the device. The identification of which functionalities should be monitored is implementation dependent and depicted in the deployed attestation policies. The aim of this procedure is to check both behavioural properties and low-level concrete properties about the entity's configuration and execution, such as the current firmware version it is running, the version of its configuration file or presence of certain hardware properties, integrity of sensor measurements, execution paths to specific memory regions, ports and network interfaces, etc. Furthermore, some of these properties might need to be attested individually while others might require to be approached as a system of systems that need to be attested by the involved devices as a group.

Furthermore, existing schemes for control flow attestation use predetermined white lists of allowed control flows [3, 53, 88] and are limited to simple applications. However these simple applications are also easy to scan for vulnerabilities, which is not feasible for more complex applications, e.g., web- or FTP servers. To the best of our knowledge, ASSURED CFA is the first approach that utilizes Neural Networks to verify the control flow traces of complex applications (Section 5.1). Further, the preprocessing ensures that the CFA is compatible with memory-protections schemes that change the memory layout, e.g., ASLR (cf. Chapter 4).

#### 2.1.1.3 Real-time Device Data and Execution Monitoring & Tracing

A core building block is the successful execution of an attestation process are the execution stream monitoring and introspection capabilities necessary for tracing the control- and information-flow execution paths. In ASSURED, dynamic tracing functionalities will be provided, as programmable components, enabling the continuous monitoring of kernel shared libraries, system calls, shared data and memory address space, etc., and the in-depth investigation of the systems' behavior for detecting cheating attempts or if any type of exploits to the program and data memory. This provides the trusted anchor with the compiled control- and information-flow graphs that represent the runtime state of a remote device, against the configuration and execution properties of safety-critical components. The ASSURED tracer is a software component presented in detail in 3.1. The expected contributions for using the tracer with the different attestation mechanisms in ASSURED are discussed next:

- **Control-Flow Attestation:** Existing software-only approaches such as C-Flat [3] require program instrumentation in order to collect information on retired control flows. Using a tracer that introspects memory to recover this information has several advantages. No need for program instrumentation (as depicted in Figure 2.1), which is not always acceptable by users after deployment. Instrumentation may hinder performance as it requires stopping the program; in the case of C-Flat, for instance, transitioning into a secure world to process the retired control flow and return back to the normal world to continue the program execution. In ASSURED, we envision the tracer would continuously run and introspect normal world programs such that it will not impact performance. Finally, such instrumentation calls may be skipped due to hijack attacks and before the control flow attestation takes place to detect them. With a tracer, skipping tracing is not possible. To overcome instrumentation, prior work relied on specialized hardware [36]. The ASSURED tracer on the other hand does not have these requirements and can be used with commodity ARM processors.
- **Configuration Integrity Verification:** Existing approaches to detect rootkits in kernel due to configuration integrity violations use a co-processor approach [71], and virtualization techniques [78]. Unlike these approaches, using a software-based tracer does not require running the programs in a virtualized environment, nor specialized hardware support.
- **Unified tracing for heterogeneous attestation schemes:** To the best of our knowledge, the ASSURED tracer is the first software-based tracer to be used to capture traces for different runtime properties of the system in a secure way utilizing TrustZone hardware-backed isolation primitives and live memory forensics. This design enables flexible trace data collection unlike hardware-only tracers.

#### 2.1.1.4 Direct Anonymous Attestation (DAA)

As aforementioned, in ASSURED, we propose the use of Direct Anonymous Attestation (DAA) (Section 3.3) for enhancing the privacy guarantees in complex supply chain ecosystems. The novelty of our proposed solution is its decentralized approach in shifting trust from the infrastructure to edge devices. Applying DAA in SoS enables enhanced privacy protection than is possible in current architectures through user-controlled linkability. While DAA has been standardized in ISO/IEC 20008-2:2013, it cannot be applied to secure supply chain ecosystems in this form, because it does not support creation of pseudonyms. The effort to modify DAA for the supply chain context has only begun with some recent academic publications that show how DAA could be used together with pseudonym certificates towards achieving adaptable unlinkability. However

this was described in a very high-level, without showing the updates needed for each DAA phase, or the mapping the current TCG specification to prove the feasibility of the solution. Furthermore, the revocation phase remained completely open and unsolved. ASSURED comes to close the gap of modifying and enhancing DAA for making it applicable to SoS:

- We define the detailed protocol specification for each DAA phase separately and the detailed mapping of the flow of TPM commands that need to be executed per phase (Section 5.2) in order to achieve all of the defined security and privacy properties needed in a SoS scenario, and
- **Design a more efficient Revocation Process beyond the use of CRLs.** We put forth the definition of the models and workflow of actions for a **novel revocation scheme**, leveraging the capabilities of our DAA-based architecture, that significantly advances the state-of-the-art: **More specifically, it does not require the use of CRLs, removing therefore, all the computational and communication overhead that comes with it.** We provide the details (Chapter 6) as well as the exchange of messages required for the successful completion of such a revocation process, coupled with a **rigorous security and privacy analysis**, with minimal trust assumptions on the correctness of the host vehicle (whose credentials are to be revoked). However, we have also highlighted how to weaken even further these trust assumptions, thus, resulting in the **first instance of such a revocation process (in the literature) with near zero-trust assumptions.** As is also the case for our overall DAA-based architecture, our scheme has been designed to be **TC agnostic** meaning that the defined models and sequence of actions can be executed over any type of trusted component as long as it exhibits the properties and provides the functionalities defined by the TCG specification. Furthermore, our efforts in designing such an efficient scheme also identified an issue with the current TPM specification and, more specifically, with the *policies- and sessions-related* core TPM services and how they are managed for enabling the communication with the attached host [64]. A problem that is based on the identification of *an infinite hash loop* was solved by updating the internal functionalities of some TPM commands and building blocks (and the introduction of a new internal key). **This elegant solution does not require any modifications or updates to the specification of the existing TPM commands that would limit the applicability of our approach.** Thus, it will be included in the next specification document of the TPM WG of TCG with a reference to our work that contributed to the overall solution.

## 2.2 Edge Device Architecture

In this section, we proceed with describing our system model and especially the architecture of what is expected to run in the edge devices. Recall that, as was described in D3.1 [21], the architecture landscape of the edge device is a combination of hardware and software components that need to work in tandem for supporting the execution of our advanced attestation schemes. The architecture of the edge device is depicted in Figure 2.2 and presented next.

### 2.2.1 Hardware components

#### 2.2.1.1 System on Chip (SoC)

We envision ASSURED edge devices would contain in their package at least the following hardware components: an ARM processor, which supports the TrustZone technology, random access

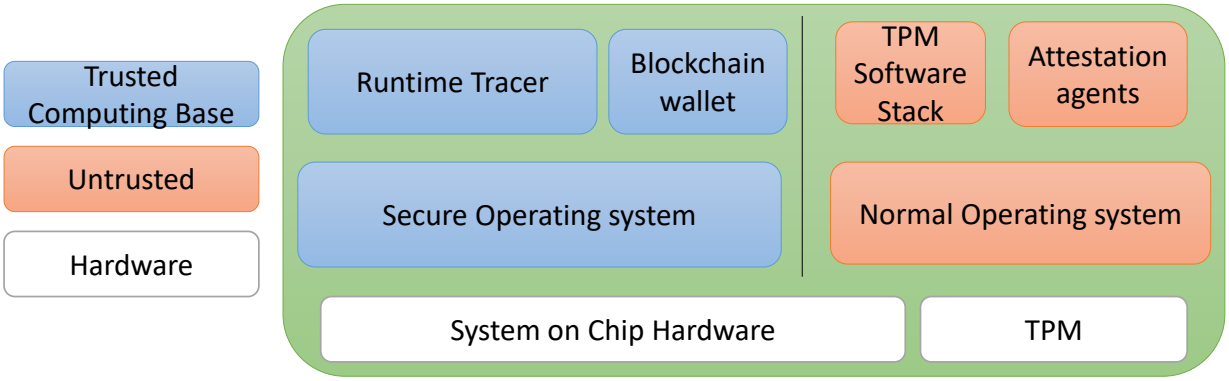



Figure 2.2: Edge Device Architecture.

memory, network interface, and non-volatile storage. To support efficient and fine-grained control flow acquisition we also consider the processor to support the CoreSight technology with the program trace macrocell.

**ARM TrustZone.** ARM TrustZone provides a low-cost solution to include a dedicated security computation region in an SoC. Specifically, TrustZone builds on hardware-backed access control that lets programs switch between two states, referred to as worlds: the secure world, and the normal world. The hardware ensures complete isolation between the worlds such that the programs executing in the normal world cannot leak or tamper with code or data that is part of the secure world. The world switch is controlled by a special configuration of the hardware: a non-secure (NS) bit that places the executing core in the appropriate world. Finally, in TrustZone the main memory is partitioned across the worlds, such that each world effectively has access to its dedicated memory range. In ASSURED, we use the TrustZone architecture to provide strong security guarantees for the ASSURED tracer by deploying it in the secure world. Effectively, with TrustZone, the ASSURED tracer cannot be compromised by adversaries controlling programs in the normal world, including privileged programs such as the normal world operating system.

**ARM CoreSight.** The CoreSight architecture provides a system-wide solution for real-time debugging capabilities and trace collection. In ASSURED, we focus on the tracing collection made available by CoreSight. Coresight-enabled systems can include the following trace sources: Embedded Trace Macrocells (ETMs), AMBA Trace Macrocells, Program Flow Trace Macrocells (PTMs), System Trace Macrocells (STMs). When tracing processor execution, instead of generating a trace for every retired instruction, PTM generates the traces only for certain instructions, and certain events, which are collectively referred to as waypoints. Waypoints result in a change in the program flow. Specifically: all indirect branches, conditional and unconditional direct branches, all exceptions, and instructions that change the security state of the processor. When a waypoint occurs in the core execution, the PTM generates trace data that describes the waypoint. This data is stored in the memory and can be later post-processed to reconstruct the execution stream given access to the source code of the program being traced. Finally, similarly to PTM, ETM is a Coresight tracing architecture that can be configured to emit traces in a fine-granularity. That is, it is possible to configure ETM to filter some instructions from being traced, thereby compressing the trace packets that are generated. We envision using ETM and PTM in the different edge devices to support fine-grained control flow tracing via the ASSURED tracer.

#### 2.2.1.2 Trusted Platform Module (TPM)

The TPM is a hardware chip that serves as the root of trust of a platform and the core building blocks of the TPM-based Wallet that is created in ASSURED for secure key management and secure on-chain interactions with the distributed ledgers (for both reading the "*attestation smart contracts*" but also recording the results of an attestation process). ASSURED provision of confidentiality requires that a hardware Trusted Platform Module ensures that the information stored in TPM hardware is better protected from external software attacks. For this, ASSURED will provide solutions based on hardware-enforced Trusted Execution Environments (TEEs) relying on the TPM technology. The TPM can encrypt/ decrypt any user data for secure storage. Access to the data is controlled by a securely stored cryptographic key. A Platform Configuration Register (PCR) is a memory location in the TPM that has some unique properties. The size of the value that can be stored in a PCR is determined by the size of a digest generated by an associated hashing algorithm. The integrity measurements are made by creating a SHA-1 digest of the system code to be loaded. This digest is stored in one of the PCR registers, which are initialised to zero. The TPM technology is required by ASSURED use cases to enable secure remote attestation and authentication to enhance the confidentiality and integrity of the exchanged data. Platform authentication will be performed by the TPM and is required in the ASSURED framework to prove that the platform is with a legitimate identity. The TPM reports its integrity measurements with a signature on these measurements to a remote verifier. TPM signs data using its Attestation Key. The connected device sends the signed data plus the identity credential to a third party. The TPM may sign using either an asymmetric or a symmetric algorithm. The TPM creates attestations about the state of the host system, e.g., certifying the boot sequence the host is running on. These attestations are achieved through ordinary signatures or through complex anonymous signatures known as Direct Anonymous Attestation (DAA). A remote verifier (context broker in ASSURED framework) verifies the Attestation Quote to check if the integrity value (PCR) is correct and the signature is really signed by this TPM.

### 2.2.2 Software components

#### 2.2.2.1 Real-time Monitoring & Tracer

The tracer is a software component, which is part of the Trusted Computing Base (TCB) [21] of the edge device. The tracer continuously introspects the programs executing and collects information to be used as part of the attestation schemes of ASSURED: control flow graphs for the control flow attestation and hashes of code pages for the configuration integrity verification. The tracer signs the traces in the secure world and passes them to the attestation agents to perform the requested attestation operation. We provide more details on the tracer component in Section 3.1.

#### 2.2.2.2 TPM-based Blockchain Wallet

The trusted platform module TPM will be coordinated by the host software towards forming the TPM-based Blockchain Wallet in ASSURED framework. Blockchain wallets are the central building block of ASSURED and form the basis for enhanced security, privacy and reliability guarantees for ledger management and maintenance. ASSURED blockchain wallets communicate with the rest of the Blockchain components of the ASSURED ecosystem to perform secure access control to the block chain ledger. Blockchain-control services in ASSURED framework will be achieved through the the TPM-based security that supports privacy-preserving protocols for securing different levels of privacy assurance as well as the enforcement of security and attestation

policies through smart contracts read and executed through the Blockchain wallet. Blockchain wallets provide a trusted execution environment that allows an isolated, secure execution of code mainly for protecting the execution of security-relevant code. This is supported the TPM through cryptographic algorithms such as hash functions, symmetric and asymmetric encryption/decryption, digital signatures etc..

Another innovation of ASSURED, is that such a wallet is designed to be aligned with the Self-Sovereign Identity (SSI) vision: In a basic SSI architecture, one core challenge is the **verification of the integrity and origin of the presented Verifiable Credentials (VCs)**: *How can someone be sure that the presented VCs really belong to the claimed entity?* Of particular interest is the case of IoT, since SSI brings users and devices in the center and gives them full control over their identity (**selective disclosure**). On a technical level, this translates to users having control of their own VCs through their Wallets which can ensure that credentials and (private) keys can only become available to this user or another actor that acts on behalf of this principal. However, since it is only the user (as the Identity Owner) that knows the (private) key associated with a VC, the level of control and assurance over a VC relies solely on possessing a software-based key, creating trustworthiness issues: **How can a Verifier be sure of the correctness of the User/Holder that presents a VC that the respective key has not been compromised?**

In ASSURED, the TPM-based Blockchain Wallet is linked with the DAA towards the protection (through hw-based keys) of the verifiable credentials and attributes that are required by the devices and users for securely interating with the Blockchain infrastructure [22]. We enable the use of crypto operations of the DAA (ECC, Blind Signatures, Zero Knowledge Proofs) towards providing additional trust assurances on the use of attributes: Issued VCs and attributes are binded to a specific Wallet and can only be used if our DAA-Bridge Extension can verify the correctness of the Wallet which, in turn, translates to the respective (private) key of this VC not been compromised. A DAA scheme enables a signer to prove the possession of the issued credential to a verifier by providing a signature, which allows the verifier to authenticate the signer without revealing the credential and signer's identity.

#### 2.2.2.3 Attestation agents

The attestation agents are the different attestation mechanisms used by the ASSURED attestation toolkit. These consists of control-flow attestation that analyzes control flow traces to verify the state of the prover, direct anonymous attestation which is able to verify the membership of a device for a group without disclosing privacy related information, swarm attestation which can attest large groups of devices, jury based attestation that utilizes multiple verifiers if the prover disagrees on the result of the first verifier and configuration integrity attestation which is able to attest on a data level.

#### 2.2.2.4 TPM software stack

The TSS is a TCG software standard that provides a standard API for accessing the functions of the TPM. Any local or remote communication with the TPM-based Blockchain wallet is achieved using TSS. TSS provides all the necessary TPM commands for various crypto primitives. It also reduces the programming complexity of applications that desire to send individual TPM commands. The TSS in the context of ASSURED will be providing the necessary standard API stack for "driving" the TPM towards supporting features such as registration, login and authentication using trusted hardware/wallet for accessing the ledger, smart contract read/write/execution, management of proofs, e.g., identity, reputation value, status, in consensus algorithm.

### 2.2.3 Architecture justification

We consider edge devices in ASSURED would include TPMs as it enables secure boot and measured boot capabilities on these platforms. Effectively, the TPMs act as the hardware root of trust for the devices and facilitate bootstrapping the trust in each device by other ASSURED components. While it is possible to use other forms of a hardware root of trust, TPMs are widely available and the support for integrating them into existing SoCs is widely available.

Enforcing the security of legacy general-purpose programs is an open research problem. In fact, many classes of attacks are used to compromise such programs. Instead, in ASSURED we choose to provide high assurance on the state of devices and programs executing within them. Specifically, on edge devices, we rely on the trustworthiness of the tracer and the blockchain wallet alone to enable the use of the different attestation mechanisms available with ASSURED. To maintain a minimal TCB, we, therefore, exclude other programs executing on the devices from the TCB.

Finally, we decide to use TrustZone to provide hardware-enforced security guarantees to programs executing in the devices' TCB. TrustZone provides confidentiality and integrity guarantees to programs executing in the secure world such that even if an adversary compromised the rest of the device it does not affect them.

# Chapter 3

# Attestation Schemes

Having defined the overall information flow of the attestation toolkit in ASSURED, in this chapter we provide a high-level overview of the designed mechanisms and their positioning in the overall ASSURED ecosystem.

## 3.1 ASSURED Real-time Device Data and Execution Stream Processing & Monitoring Programmable Component

We first start with the core building blocks that enables the entire attestation process - the **execution stream monitoring and control-flow tracer**. The tracer, which is depicted in Figure 3.1 and used in ASSURED is a software component executing in a trusted execution environment (TEE). Therefore, it is part of each edge device's TCB acting as a trust anchor, which is inaccessible to attackers who potentially can compromise the rest of the device. The tracer runs continuously as a daemon process in the TEE, tracking the volatile physical memory of the device.

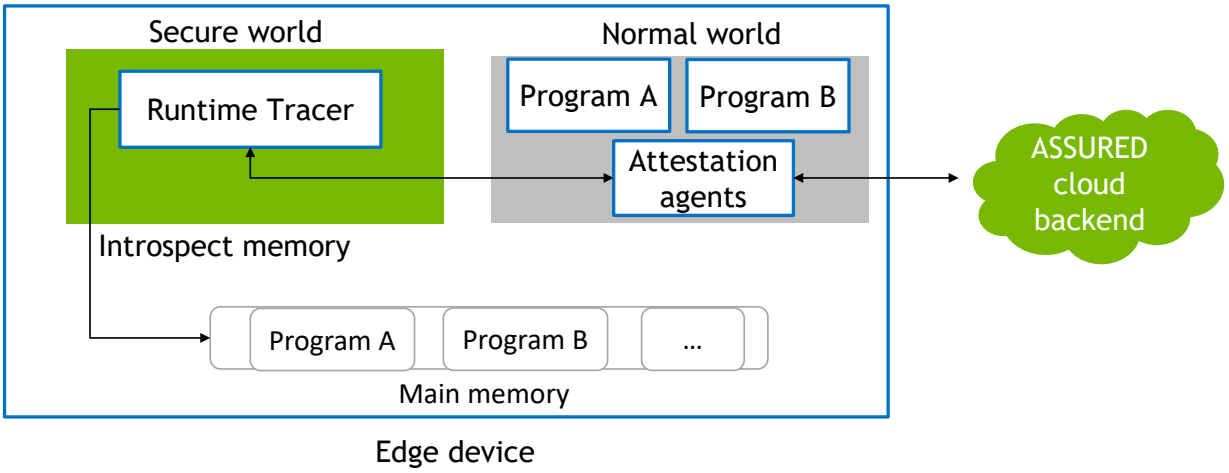



Figure 3.1: Overview of the ASSURED tracer

The tracer is pre-configured with specific device information: **symbol names and their corresponding offset in memory for both the normal world's operating system and programs executing in the normal world**. Utilizing this exact virtual memory information, the tracer bridges the semantic gap and recovers virtual-to-physical address mappings such that it can recover high-level semantic information about the device and programs executing in it under the assumption

of an a priori knowledge of the operating system services and their respective implementation. For example, to recover the current process list of a device the tracer uses the $task\_struct$ structure defined as part of the Linux operating system kernel. The $task\_struct$ contains information about a specific process such as its unique identifier, its name, its virtual memory addresses, etc. Furthermore, the $task\_struct$ acts as a linked list node in Linux and thus contains a pointer to the next $task\_struct$ node. While these pointers contain virtual memory addresses, the tracer armed with the knowledge of virtual-to-physical address translation can traverse the linked list, perform software address translation and recover the details of all currently running processes on the device.

Similarly, to the above process list example, the tracer effectively performs live memory introspection. The tracer uses public information on implemented operating system services to recover high-level semantic information from raw physical memory access. For example, the tracer can also detect the code and libraries that were loaded into memory for each process and infer runtime control flow decisions made by the programs.

The high-level algorithm of the tracer is depicted in Algorithm 1 and a general description is provided next.

**Algorithm 1** The ASSURED Tracer high-level algorithm.

**while** True **do**
  $p \leftarrow get\_security\_policy()$ ▷ Security policy received from the attestation agent.
  **if** $p$ is CFA **then**
    $pid \leftarrow get\_pid(p)$
    $t \leftarrow cfa\_trace(pid)$ ▷ Track the control flow for process $pid$
  **else if** $p$ is CIV **then**
    $t \leftarrow civ\_trace()$ ▷ Get the measurement of loaded libraries and binaries
  **end if**
  $signed\_t \leftarrow sign(t)$
  $res \leftarrow verify\_attestation(signed\_t)$
  **if** $res$ is False **then**
    $mem\_dump \leftarrow get\_mem\_dump(pid)$ $attestation\_fail(signed\_t, mem\_dump)$ ▷ Sends the memory dump and failed trace to the attack validation component
  **end if**
**end while**

As the tracer continuously runs, it maintains an active security policy that should be enforced on the device, which translates to the forensic technique that should currently be employed to detect the state of the device. This security policy is decided by the cloud-backend of the ASSURED framework and is propagated to the tracer through the attestation agents. The latter transmits the requested policy to the tracer in a secure and trustworthy fashion.

Finally, according to the active policy, the tracer creates an authentic report, denoted as a *trace*, which is then signed and transmitted securely to the attestation agents to invoke the requested attestation mechanism. The attestation is completed with a verification that the trace is trustworthy by validating the signature and validating the trace data adheres to the attestation mechanism requirements.

As aforementioned, in ASSURED, the following attestation mechanisms are used.

**Configuration integrity verification (CIV).** This attestation mechanism validates that all the binaries loaded in the device have the expected content. That is, that no malicious adversary loaded an invalid program or changed any of the code pages of deployed programs as meant by the end-user. To support CIV, the tracer measures all the code pages belonging to a given program, which includes the program binary and associated libraries. The tracer computes the measurement based on a well-established cryptographic hash function, taking as input all the code pages belonging to the program. The tracer infers the virtual address ranges for the code pages as the Linux operating system stores this information in the $task\_struct$ process structure in the $mm$ field.

The tracer repeats this operation for all the programs that currently execute on the device, by traversing the process list embedded in the $task\_struct$ structure, and generates a measurement value that represents the current state of the programs in the device. Effectively, in this attestation scheme, the tracer acts as the prover. The corresponding verifier gets the signed trace and validates the measurement matches a pre-established measurement value for the programs that should execute on the device.

**Control flow attestation (CFA).** This attestation mechanism validates that a specific program control flow is valid. To support CFA, the tracer tracks the control flows of a given program and generates a trace report of the latest control flow graph of the program. In ASSURED, we consider two alternatives to track the control flow of a program.

- **Stack trace reconstruction.** We employ the state-of-the-art forensic technique to recover stack traces of invoked functions using register information stored in the main memory. Specifically, the Linux operating system, which is deployed as the operating system in the normal world of the edge devices stores register information at the bottom of the kernel stack as part of the kernel virtual address space on privileged mode switches. This information is needed by the operating system to correctly restore the user context when returning to the program that is executing in userspace. Fortunately, the kernel stores the user program stack pointer register as one of the saved register values. The stack pointer register points to the top of the stack and matches the latest user program function that was invoked. To recover the full stack trace we traverse the user stack from top to bottom using the frame pointers. Specifically, the calling convention in Linux stores the frame pointer before the return address on the stack, and potential argument if used by the function. The frame pointer points to the previous frame pointer and so forth up to the bottom of the stack. Therefore, we identify the frame pointer and use it to traverse the entire user stack, recovering all the functions called by the user. While this approach does not require any special hardware support it has two important limitations for constructing an accurate control flow graph of a program. First, the control flow graph only contains functions and not basic block branching decisions made by the program. Second, the mode switches between user programs and the kernel may not be frequent enough such that short-lived functions could be missed from the control flow graph by this approach. Therefore, we also consider using hardware tracing features such as ARM Coresight to recover fine-grained control flow graphs as discussed next.

- **Hardware-assisted tracing.** We envision using the ARM Coresight tracing technology to configure the processor to always emit branch instructions to a circular buffer in a predefined physical memory location known to the ASSURED tracer. To recover the control flow graph of a program, The ASSURED tracer would continuously introspect this buffer and recover

the branch instructions executed by the processor. The ASSURED tracer would infer the branches that are corresponding to the program to be traced for the control flow attestation and using symbol information it has on the binary and libraries of the program recover the functions and basic blocks that were executed during the program run. Using this traced information the ASSURED tracer would emit the control flow graph to a signed trace that would be used by the control flow attestation verifier.

Finally, in case an attestation for a trace fails, either for CIV or CFA, it signals that the device might have been compromised. Therefore, the tracer sends the failed attestation trace and a memory dump trace to the ASSURED cloud backend services to assess the security state of the device. To reduce network overhead by sending an entire snapshot of the device's physical memory, the memory dump contains only the recorded state of the working memory of the program that failed the attestation process at the time the failure occurred. The memory dump is recovered by the tracer by traversing the $mm$ field in the $task\_struct$ structure. The traces generated by the tracer follows the format depicted in Table 3.1.

| Validation entity | Format | Description |
|---|---|---|
| CIV | [lib_i:<br>[hash, [invalid byte_0,..,<br>invalid_byte_n]] | List binaries and libraries. Each list node is a tuple with the binary name, its corresponding measured hash value followed by a list of invalid bytes that are zeroed out before the measurement. |
| CFA | [(symbol_0, library), ... (symbol_n, library)] | List of tuples corresponding to visited symbols and the library or binary they are part of. |
| Memory dump | [(page_i, [byte_0, ..., byte_4095]] | List of tuples pages corresponding to page addresses and their content. We consider each page has a size of 4KB |

Table 3.1: Tracer output for each validation entity

## 3.2 Control Flow Attestation

Control Flow Attestation (CFA) is about being able, in a two parties system, to detect any kind of manipulation of Control Flow Graph (CFG), which describes the flow of execution on an application (see Figure 3.2 for a high-level example).

The two actors are:

- **Verifier** is the trusted party, who assures the integrity of the CFG;
- **Prober** is the untrusted party, who provides the traces that have to be verified.

With Control Flow Integrity (CFI) we refers to those techniques that aim to prevent a verity of malware attacks from redirecting the flow of execution (CFG) of a program. In order to achieve

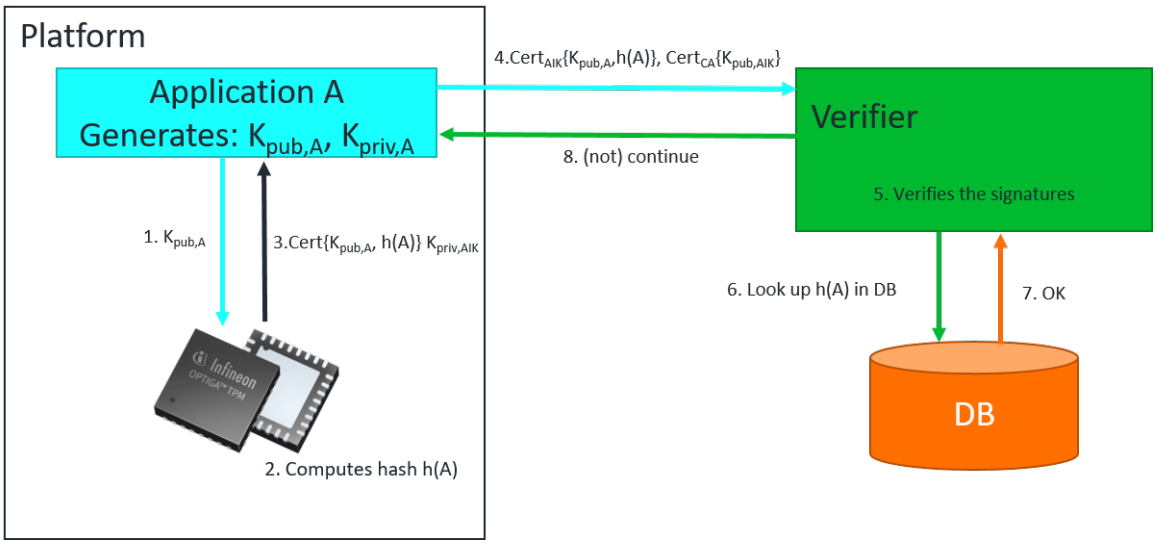



Figure 3.2: Remote Attestation Example

CFA, it has to be defined which CFI techniques to adopt. At the moment of writing, CFI has two main problems: accuracy and performances. Both are mostly due to the fact that it is not trivial to compute the CFG since source code may be not available and because of dynamic code which changes CFG at run-time.

The most common CFI techniques are:

- *Code randomization* is an attempt to make code reuse harder by shuffling the location of the code to be reused. As show in [50, 69, 75] even simple information leak can reveal all the process;
- *Coarse-Grained CFI* is CFI applied on top of a coarsely grained built CFG. It suffers of weak security guarantees and can be bypassed [48];
- *CFI through code analysis* is not sound for protection, mainly due to dynamic code;
- *Shadow Stack* is a more deep approach based on the idea of using a separate and protected stack (Shadow Stack) where to store function calls return address. Once a function is about to return, function's return address is compared with the one stored in the Shadow Stack and if these two mismatch then an anomaly is detected. The use of this technique deeply impact on performance and the system architecture since it has to be re-designed (i.e. ISA has to be extended) in order to include instruction for managing such operations. Nonetheless, as [20] sustains, their use is fundamental for any practical CFI implementation.

An alternative approach were proposed by [87], which suggested to use a Deep Neural Network (DNN) for detecting abnormal control flows. This study was mainly focused on Code Reuse Attacks (CRAs) and used Intel Processor Traces (IPT) for building a fine-grained CFG. As assumption they consider only trusted applications. In conclusion, they were able to achieve up to 98.9% of accuracy. This approach has some limitations: The neural network needs a high amount of training examples for the training which need to be manually labeled. Further, every time an application changes, the data collection and labeling need to be repeated, causing a significant overhead.

Further, for training a DNN with a supervised algorithm also attack examples are required, making the algorithm not effective against new attacks. To solve these problems, we train a DNN in a semi-supervised mode, requiring only benign traces. During the detection phase, traces that differ significantly from the training data are considered to be malicious. Using only benign traces for the training, in addition with a scheme that automatically collects benign traces, e.g., by using techniques like fuzzing, allows the system to adapt dynamically to changes of the application and automatically train for new applications.

The CFA is part of the verifier in the ASSURED framework. The runtime attestation is either initiated by a prover that wants to show its trustworthiness to other components or is initiated by the attestation toolkit based on the defined policies, e.g., if a new device joins the system. After the control-flow trace is evaluated, the verifier send the attestation result to the TPM wallet.

## 3.3 Direct Anonymous Attestation

As aforementioned, seeking to design secure and privacy-preserving supply chain architectures, comprising of thousands of autonomous and intelligent edge nodes, besides operational assurance - one has to cater for a number of properties like anonymity, pseudonymity, unlinkability, and unobservability and the strict trust requirements of a wide variety of multi vendor devices and platforms. Towards this direction, in ASSURED we will employ advanced cryptographic primitives (namely Direct Anonymous Attestation) together with trusted computing techniques for facilitating the strict privacy considerations encountered in a variety of emerging applications. Consider, for instance, the following example in the context of the envisioned *Public Safety Scenario*: A user by leveraging his/her mobile device can send immediate feedback about an incident that takes place in his/her vicinity; e.g., maybe an ongoing fire or even theft can be reported (via the ASSURED Blockchain infrastructure) to the cloud-based backend processing system. However, in order to motivate users to participate in such crowd-sensing systems, it is imperative to protect their privacy. **Data should be authenticated in terms of their origin - originate from a valid and enrolled user (whose device can provide a verifiable device on its correct operation) - but should not be linked back to his/her ID and/or location**. But this openness is a double-edge sword: any of the participants can be adversarial and pollute the collected data, seeking to manipulate (or even dictate) the system output. Faulty, distorted information can lead to wrong decisions, possibly rendering such public-safety systems useless.

In this context, DAA is a **platform authentication mechanism that enables the provision of privacy-preserving and accountable authentication services**. By leveraging group-based signatures, any verifying entity can verify a platform's credentials in a privacy-preserving manner using the previously described DAA algorithms; without the need of knowing the platform's identity. The Elliptic-curve cryptography (ECC) based DAA is comprised of five algorithms SETUP, JOIN, SIGN, VERIFY and LINK.

### 3.3.1 The Need for "Privacy-by-Design" in Complex Systems-of-Systems

Privacy requirements have been well documented in the European Telecommunications Standards Institute (ETSI) TS 102 941 [40] highlighting the following properties:

- *Anonymity:* ability of an edge node to use a resource or service without disclosing its identity.

- *Pseudonymity:* ability of an edge node to use a resource or service without disclosing its identity while still being accountable for that action.
- *Unlinkability:* ability of an edge node to make multiple uses of resources or services without others being able to link them together (i.e., infer mobility patterns).
- *Unobservability:* ability of an edge node to use a resource or service without others, especially third parties, being able to observe that the resource or service is being used.

In this context, **the actual identity of the sender is not required for ensuring the trustworthiness of a transmitted message**. It rather suffices to verify the **origin correctness**; a message has been sent by a valid "node participant". Indeed, since exchanged messages might contain sensitive data (cf. *Public Safety Scenario*), what is required is that certificates should not contain any identifying information that could *trace* them back to a particular device or platform. In an attempt to address this challenge, intensive efforts in academia and industry, led to the proposal of **PKI-based solution**s [46] **with privacy-friendly authentication services** through the use of short-term anonymous credentials, i.e., **pseudonyms**. *The common denominator in such architectures is the existence of trusted (centralized) infrastructure entities for the support of services such as authenticated node registration, pseudonym provision, revocation, etc.*

While it has been proven the security guarantees provided in such architectures, there are a number of challenges inherent to PKIs when it comes to **privacy**, **scalability**, and **operational assurance** [45]. In its core, the possibility of security breaches has the potential to seriously weaken the technical security protection measures of PKIs, since in their current version the assumption is on the existence of a number of centralized trusted entities. However collusion or security incidents affecting certification authorities have grown more frequent in the recent past [38], **so the existence of a PKI architecture does not guarantee per se the enactment of trust between the peers and additional measures are necessary to reinforce a scalable community of trust** [9].

Therefore, **what is needed is to provide efficient, reliable and in timely and privacy-preserving communications to all edge nodes**. The reliance on infrastructure entities within the overall architecture for such services raises questions towards a system's availability and scalability in the case of a technical fault or attack. In this context, **ASSURED leverages anonymous credentials through the use of Direct Anonymous Attestation (DAA) addressing all the aforementioned limitations, i.e., privacy, security, and accountability.**

One of the biggest advantages of ASSURED DAA scheme is its **semi-decentralized nature** resulting in shifting the trust from the back end infrastructure to the edge nodes. Applying the DAA protocols results in the redundancy (and removal) of the PKI-based architecture: *edge nodes can now create their own pseudonyms, and DAA signatures are used to self-certify each such credential that is verifiable by all verifiers*. Furthermore, nodes have total control over their privacy, as no trusted third-party is involved in the pseudonym creation phase. This means that it is infeasible for any third-party to reveal the identity of another device assuring that pseudonym resolution is not possible in our solution.

### 3.3.2 Direct Anonymous Attestation Building Blocks

DAA [12] is a platform authentication mechanism that enables the **provision of privacy-preserving and accountable authentication services**. DAA is based on group signatures that give strong anonymity guarantees. The key security and privacy properties of DAA are:

- *User-controlled anonymity*: Identity of user cannot be revealed from the signature.
- *User-controlled linkability*: User controls whether signatures can be linked.
- *Non-frameability*: Adversaries cannot produce signatures originating from a valid trusted component.
- *Correctness*: Valid signatures are verifiable, and linkable, where needed.

In general, a Direct Anonymous Attestation (DAA) scheme consists of an Issuer (ASSURED Privacy CA), a set of signers and a set of Verifiers (Figure 3.3). The Issuer creates a DAA membership credential for each signer. In practice, a DAA credential corresponds to a signature of the signer's identifier produced by the certification authority. A DAA signer consists of the (Host, TPM) pair. Their membership to the DAA community and trustworthy state is proved by providing the Verifier with a DAA signature of the TPM representation of the Host state. The DAA signature includes a zero-knowledge proof-of-knowledge, which is a cryptographic construct used to convince the Verifier that the signer possesses a valid membership credential, but without the Verifier learning anything else about the identity of the signer. In contrast to other privacy-preserving constructs, like group signatures [34, 65], DAA does not support the property of traceability, wherein a group manager can identify the signer from a given group signature. Furthermore, when the DAA issuer also plays the role of a Verifier, the issuer does not obtain more information from a given signature than any arbitrary Verifier. However, to prevent a malicious signer from abusing anonymity, DAA provides two alternative properties as the replacement of traceability. One is the rogue signer detection, i.e. with a signer's private key, anyone can check whether a given DAA signature was created under this key or not. The other is the user-controlled linkability: two DAA signatures created by the same signer may or may not be linked from a Verifier's point of view. The linkability of DAA signatures is controlled by an input parameter called the basename. If a signer uses the same basename in two signatures, they are linked; otherwise, they are not.

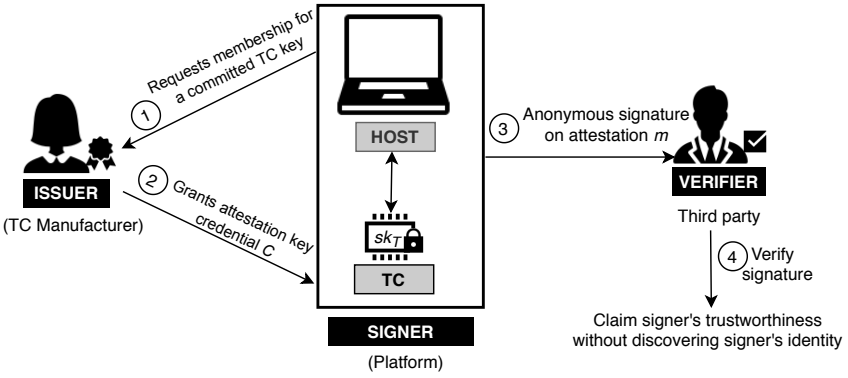



Figure 3.3: An overview of the entities involved in a DAA protocol

The Issuer is a trusted third-party responsible for attesting and authorizing platforms to join the network (through the execution of the described zero-touch configuration integrity verification process - Section 5.3). *In the context of ASSURED, the role of this entity is undertaken by the Privacy CA who is responsible for certifying the correctness of a device prior to verifying it a registration token that can then be used to request credentials from the Blockchain-related CA*. A verifier is any other system entity or trusted third-party that can verify a platforms' credentials in a privacy-preserving manner using DAA algorithms; without the need of knowing the platform's identity. The Elliptic-curve cryptography (ECC) based DAA is comprised of five algorithms SETUP, JOIN, SIGN, VERIFY and LINK.

- SETUP - The system parameters must be chosen and the Issuer needs to generate its keys. The system parameters and the Issuer's public keys are then published and available to the cluster and to anyone who needs to verify the validity of a signature.
- JOIN - A Host using a TC joins the group and obtains an Attestation Key Credential (AKC) for an ECC-DAA key created by the TC. The key can then be used to anonymously sign a message, or attest to data from this TC.
- SIGN - Using the ECC-DAA key, for a range of signing operations.
- VERIFY - Verifying a signature and returning true (valid) or false (invalid).
- LINK - Checking two signatures to see if they are linked and returning true (linked) or false (un-linked).

A DAA scheme enables a signer to prove the possession of an issued credential to a verifier by providing a signature, which allows the verifier to authenticate the signer without revealing the credential and the signer's identity. In a nutshell, DAA is essentially a two-step process where, firstly, the registration of a device executes and during this phase the device chooses a secret key (SETUP). This secret key is stored in secure storage so that untrusted code cannot have access to it. Next the device talks to the issuer so that it can provide the necessary guarantees of its validity (JOIN). The issuer then places a signature on the public key, producing the Attestation Identity Credential (AIC) $cre$. The second step is to use this $cre$ for anonymous attestations on the platform (SIGN), using Zero-Knowledge Proofs [47]. These proofs convince a verifier that a message is signed by some key that was certified by the issuer, without knowledge of the TC's DAA key or $cre$ (VERIFY).

### 3.3.3 DAA Applications in the ASSURED Framework

The intuition behind employing DAA in ASSURED is for enabling **privacy-preserving platform authentication**. When a device wants to join the network for then been able to exchange either *operational and/or threat intelligence data* in a privacy-preserving manner (achieving properties including **anonymity, unlinkability, untraceability**), this can be done by leveraging short-term anonymous credentials - such as *pseudonyms* - that are created under one master ECC-based DAA Key. Data bundles then shared with other devices either directly or through the ASSURED Blockchain infrastructure, can be (anonymously) signed under these short-term credentials. One of the core innovations of DAA is that the level of privacy is controlled by the device itself through configuring the type of signatures leveraged. **This enables to better shift trust from the infrastructure to the edge**.

More specifically, ASSURED framework will employ the DAA protocol to enable ASSURED components and devices not only to authenticate themselves to a verifying edge device and to prove that the component is in a trustworthy state, but also to do so in a privacy-preserving manner. More concretely, the DAA provides the TPM-based Wallet with the ability to sign its register values in an anonymous way, whilst still convincing the Verifier that it possesses valid DAA credentials.

For instance, ASSURED new device enrolment framework asks for an attestation report that allows the peer to verify that the new joining device has a legitimate TPM and has the correct attestation record. The peer challenges the TPM who responses by creating Direct Anonymous Attestations DAA about the state of the new edge device and reporting the PCR values. The goal of such attestations is to convince the block chain node/ verifier that the new edge device it

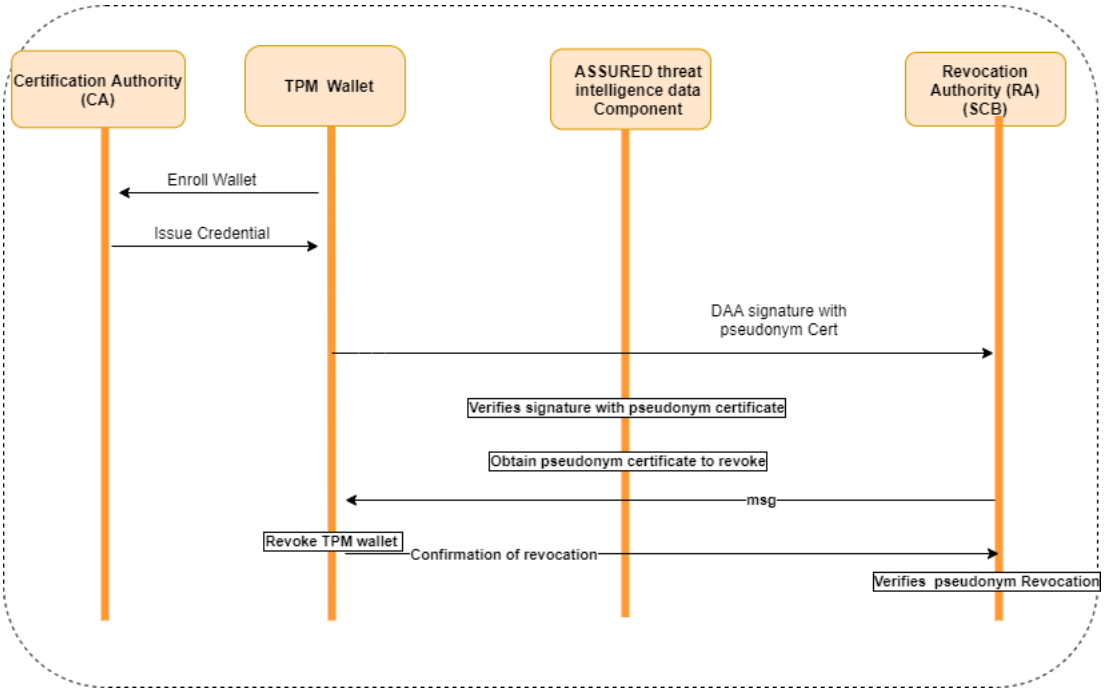



Figure 3.4: Overview of TPM Wallet Pseudonym

is communicating with is running on a trusted TPM wallet and using the correct software. After authenticating and attesting an edge device, the broker merges the access request and uploads the DAA results on the block chain ledger, then grants an access to the edge device to download and executes the smart contract. The DAA supports linkability that is required later whenever this new edge device, after granting access to the ledger, fails to upload a correct attestation report, the broker can then link this report to the initial DAA report that was created when enrolling this device. This linkability feature is essential for revoking corrupt devices while preserving the anonymity of honest devices.

**TPM wallet Pseudonyms**: To create DAA signatures, a TPM wallet should have a valid credential issued from a Certification Authority CA (Privacy CA), the protocol for obtaining a credential is intentionally not privacy-preserving as the the CA authority needs to be aware of the wallet to be authenticated. However, successful completion of the protocol results in the TPM wallet solely owning a valid DAA credential. In ASSURED, we envision, the TPM blockchain wallet, upon registration with ASSURED block chain services, will create its own pseudonyms that are used to self-certify a TPM credential and this certification can be verified by any Blockchain user. This is called pseudonymity which is the ability of a TPM wallet to use a resource or service without disclosing the user's identity while still being accountable for that action, i.e. , any Revocation Authority RA can then remove misbehaving wallets without revealing the TPM wallet's identity.

The pseudonymity property will be adopted by ASSURED using unlinkable DAA. For the pseudonym creation that is done from the TPM wallet side, the TPM wallet chooses a fresh pair of public and private keys $(PK_{pseud}, SK_{pseud})$, self certifies $PK_{pseud}$ by creating unlinkable direct anonymous attestation, say $\sigma_1$, on $PK_{pseud}$ using its DAA secret key, and randomize its credential that was created by the CA to hide the TPM wallet identity. This signature together with the randomized credential and $PK_{pseud}$ constitutes the TPM wallet pseudonym. Such pseudonyms are anonymous and unlinkable, therefore any third party cannot identify or link any subsequent service

originating from the same wallet.

## 3.4 Swarm Attestation

The ultimate goal of the swarm attestation schemes is to provide scalable solutions that verify the trustworthiness of a large-scale network in a more efficient way than attesting devices individually. The existing swarm RA protocols (e.g., [10], [56], [59], [37], to mention only a few) differ from each other from various design parameters such as network topology, adversary model, attested memory regions, verification of exchanged communication data, number of verifiers, etc.

### 3.4.1 Network Topology

Based on the network topology, there are two types of IoT swarm networks, which can be categorized as follows:

- **Static Swarms.** Generally, swarm attestation schemes rely on the assumption that the network is static. Static swarm attestation approaches construct the interconnected network as a spanning-tree. In these schemes, each device statically attests its children and reports back to its parent the number of children that successfully passed the attestation protocol. In the end, an aggregated report with the total number of the devices successfully attested will be transmitted to the Verifier. The typical interactions of a static swarm attestation are depicted in Figure 3.5.
- **Dynamic Swarms.** To release the assumption of the static swarm attestation schemes that the network is static and interconnected, dynamic swarm attestation protocols enable the swarm attestation in dynamic networks. These schemes provide a mechanism to address the challenges introduced by highly dynamic networks by fusing consensus techniques [73], [68] in remote attestation. In these approaches, devices first share their respective "knowledge" with other devices, and then they use consensus mechanisms to agree on a common knowledge about the whole network. Here, devices interact synchronously at the attestation time, even though these schemes do not require the construction of a spanning tree for the collection of attestation report.

### 3.4.2 Adversary model

Attackers will often try to compromise devices and networks to steal confidential data, infect poorly secured devices to make them part of a botnet, get access to unauthorized places, etc. Remote Attestation protocols take into consideration hardware and software adversaries, which depend on the attackers' location and the type of attacks they can perform. For a complete overview of the adversary model addressed in ASSURED, the reader is invited to consult Section 4. Overall, we consider two main adversaries:

- **Remote software adversaries ($Adv_{SW}$)** - The remote attacker can run malicious code on the device to compromise the application running on the device or device firmware.
- **Physical intrusive adversaries ($Adv_{PHYS}$)** - The attacker has the ability to physically capture a device and clone it or extract secrets from it. For instance, the attacker can disassemble the device, get access to the pins of its micro-controller unit and probe the debugging

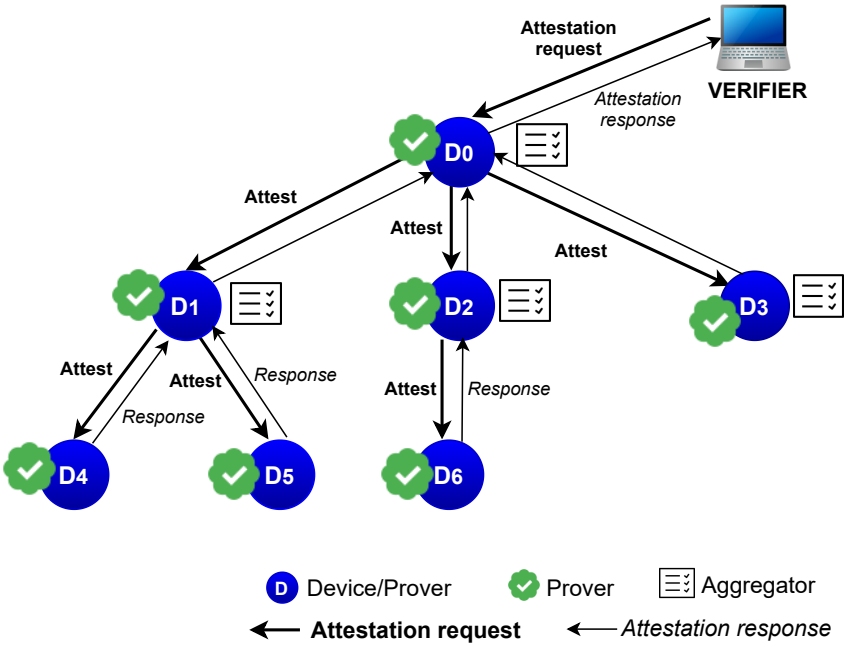



Figure 3.5: Overview of a typical static Swarm attestation

and testing pins to get access to different pieces of information about the device. In addition, the attacker has the ability to physically modify its components or add external hardware to the device and change its initial structure.

### 3.4.3 Attested Memory

Based on how the RA schemes use the memory contents during the attestation process, the protocols can belong to one of the following two categories:

- **Static Attestation** - Static attestation is used by most of the RA schemes and provides a straightforward approach on verifying the integrity of static objects. The algorithm computes a hash of the device's memory or takes a fingerprint of it. The main drawback of this method is that it does not apply to dynamic objects in the application. Hashing dynamic entities on a device is ineffective because they often change in an unpredictable way, making it impossible for the Verifier to keep track of their states. Runtime malware can modify dynamic objects, thus infecting the device and hiding from static attestation protocols.
- **Dynamic Attestation** - Dynamic Attestation focuses on checking the integrity of the dynamic properties of a running application in order to prove the integrity of the runtime system. Unlike performing a hash of the memory as it is the case of static attestation, the dynamic attestation schemes continuously track dynamic properties, such as the Instruction Pointer of the system [1], stack registers, or the application's control flow. One of the main challenges of dynamic attestation schemes is the difficulty in identifying the "good" dynamic properties of a system, based on which the runtime system integrity should be performed. Moreover, there is a need for a large number of dynamic entities during the attestation process, which need to be frequently measured, since their state changes all the

[1] The Instruction Pointer, or Program Counter, is a register of the processor (CPU) which indicates the memory address of the next instruction that will be executed in the program sequence.

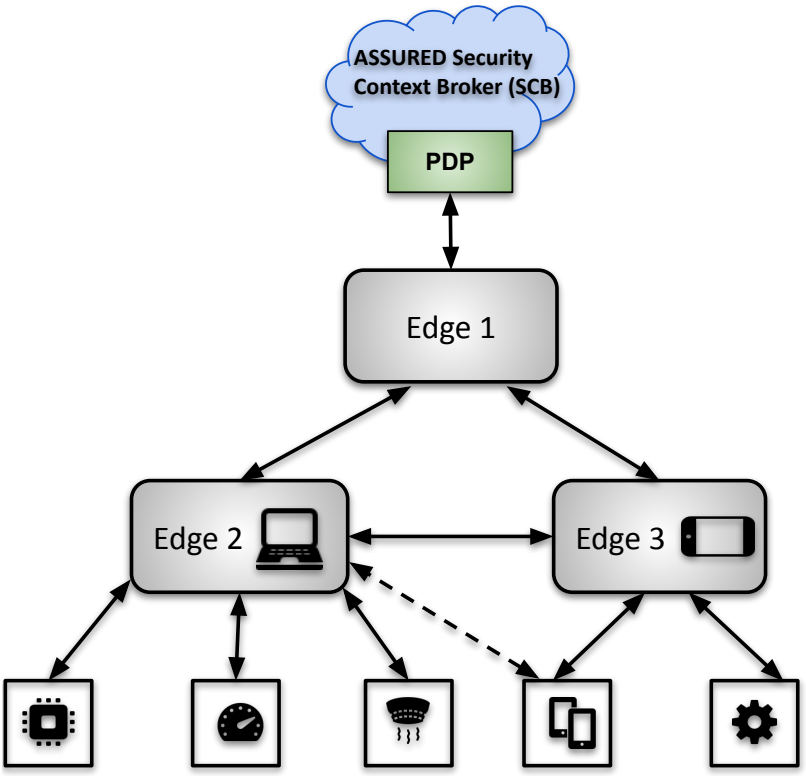



Figure 3.6: Overview of Swarm attestation in ASSURED

time. However, the main advantage of dynamic attestation is its ability to detect run-time attacks, compared to static attestation which is not able to do this.

### 3.4.4 Communication Data exchanged between Devices

Another approach to categorize RA protocols is based on the schemes that consider the data transferred between devices before or during the attestation process (i.e., distributed services) and those that do not (i.e., swarms). The ultimate goal of the protocols that verify the communication Data exchanged between devices is not only to attest a device's trustworthiness, it also makes sure that the exchanged data was not malicious and did not maliciously affect other interacting Provers.

### 3.4.5 Number of Verifiers

Based on the number of Verifiers that participate in the attestation process, there are two types of swarm RA protocols:

- **One Centralized Verifier** - RA scheme in which there is only one centralized Verifier, whose role is to collect the final attestation result obtained as a consequence of running the remote attestation process.
- **Many Distributed Verifier** - most of the RA protocols are based on one-to-one or one-to-many relations, where there is one Verifier that attests one or more Provers. However, the single Verifier can be considered a single point of failure. Thus, RA schemes that are based on distributed Verifiers have relations, such as many-to-one and many-to-many, i.e., multiple Verifiers attest one or more Provers.

| RA Protocol | Network | Adversary | Memory | Exchanged data | No. Verifiers | Privacy |
|---|---|---|---|---|---|---|
| SEDA [10] | static | $Adv_{SW}$ | Static | ✗ | 1 | ✗ |
| DARPA [56] | static | $Adv_{SW}$,$Adv_{PHYS}$ | Static | ✗ | 1 | ✗ |
| SANA [4] | static | $Adv_{SW}$ | Static | ✗ | 1 | ✗ |
| LISA [16] | static | $Adv_{SW}$ | Static | ✗ | 1 | ✗ |
| SCAPI [58] | static | $Adv_{SW}$, $Adv_{PHYS}$ | Static | ✗ | 1 | ✗ |
| SALAD [59] | dynamic | $Adv_{SW}$, $Adv_{PHYS}$ | Static | ✗ | Many | ✗ |
| PADS [5] | dynamic | $Adv_{SW}$ | Static | ✗ | Many | ✗ |
| WISE [7] | static | $Adv_{SW}$ | Static | ✗ | 1 | ✗ |
| slimIoT [8] | static | $Adv_{SW}$, $Adv_{PHYS}$ | Static | ✗ | 1 | ✗ |
| RADIS [31] | static | $Adv_{SW}$ | Dynamic | ✓ | 1 | ✗ |
| ESDRA [61] | dynamic | $Adv_{SW}$ | Static | ✗ | Many | ✗ |
| US-AID [54] | dynamic | $Adv_{SW}$, $Adv_{PHYS}$ | Static | ✗ | Many | ✗ |
| SHeLA [72] | dynamic | $Adv_{SW}$ | Static | ✗ | 1 | ✗ |
| HEALED [55] | static | $Adv_{SW}$ | Static | ✗ | Many | ✗ |
| SIMPLE+ [6] | dynamic | $Adv_{SW}$ | Static | ✗ | 1 | ✗ |
| PASTA [60] | static | $Adv_{SW}$, $Adv_{PHYS}$ | Static | ✗ | Many | ✗ |
| DIAT [1] | dynamic | $Adv_{SW}$ | Dynamic | ✓ | Many | ✗ |
| SARA [37] | dynamic | $Adv_{SW}$ | Static | ✓ | 1 | ✗ |
| ARCADIS [49] | dynamic | $Adv_{SW}$ | Dynamic | ✓ | 1 | ✗ |
| **ASSURED** | **Dynamic** | $Adv_{SW}$, $Adv_{PHYS}$ | **Dynamic** | ✓ | **Many** | ✓ |

Table 3.2: Properties of the state-of-the-art Swarm Attestation Protocols

### 3.4.6 Swarm attestation in ASSURED

In the ASSURED framework, devices attest each other and the aggregated results are reported to the central ASSURED components, as shown in Figure 3.6. In particular, each device may act both as Prover and Verifier to support the aggregation of the attestation result across the network. For instance, edge devices can attest each other to generate an aggregated result to be validated by the Policy Decision Point (PDP) in the ASSURED Security Context Broker (SCB). In ASSURED, the aggregation process on each edge device is managed by the Trust Aggregation Overlay component. It is crucial for the swarm attestation in the ASSURED framework to guarantee privacy-preserving property. To achieve this, the swarm attestation will rely on DAA, which is based on group signatures that allow remote attestation of a device associated with a Trusted Component (TC) while offering strong anonymity guarantees. Specifically, DAA allows any verifier to check that attestations originate from a certified hardware token without learning anything about the identity of the TPM.

During the design phase, the stakeholders define the verification policies and provide them to the SCB. In the setup, first the blockchain wallet is created, which will establish a relation with the trust aggregation overlay. The Trust Aggregation Overlay, in turn will register the new device with the security context broker, which will respond with the initial attestation policy to be applied to the new device. This information is passed on to the device's blockchain wallet which will configure the tracing mechanism accordingly (if needed). During operation, the trust aggregation overlay can perform verifications of the device, if a violation of the currently active verification policy is detected an alert is propagated to the risk assessment component via the security context broker, which triggers the risk assessment process and prompts the update of security configurations and policies. Similar to the single device attestation case, the risk assessment and threat intelligence components can request additional information and update the attestation and verification policies, which might be proxied through the trust aggregation overlay.

#### 3.4.6.1 Comparison of properties of Swarm attestation in ASSURED

To highlight their fundamental differences among the state-of-the-art swarm RA solutions, Table 3.2 presents an overview of their main properties. As it can be seen from Table 3.2 all the protocols work on static networks, in which the devices do not move or change neighbors during the attestation process and are fully connected. i.e., all Provers are online and actively participate in the process. This is the simplest type of network architecture, which is starting to change and evolve due to the dynamic networks, such as interconnected cars, drones and robots that map areas, etc., that becoming frequently deployed. Half of the analyzed swarm RA schemes are able to function properly in dynamic networks, where the IoT devices can relocate and change neighbors during the attestation procedure. Furthermore, only few of the existing swarm RA protocols perform attestation of dynamic parts of the device memory.

## 3.5 Jury-based Attestation

Analogous to swarm attestation (cf. Section 3.3), jury-based attestation targets large networks of devices. However, instead of letting one or more distinct and external verifiers attest the swarm, jury-based attestation is an approach to let devices attest each other in a scalable manner. This is particularly useful in cases in which devices from different parties need to collaborate. Establishing individual trusted verifiers may be hard to do in practice, e.g., different manufacturers may not trust each other enough to establish a common verifier. However, designing a distributed, network-wide device-to-device attestation without a commonly trusted entity is challenging. On the one hand, the naive solution of simply letting devices attest other devices when needed, potentially leads to an exponential amount of attestations. Thus, a noteworthy attestation result, e.g., a detected compromise, should be publicized among the network to establish a network-wide shared state, such that, e.g., compromised devices are known across the network. However, as the verifying devices themselves may act maliciously, simply broadcasting such a result is not secure. On the other hand, when using a distributed approach, the approach itself needs to be resilient against abuse by participating compromised devices.

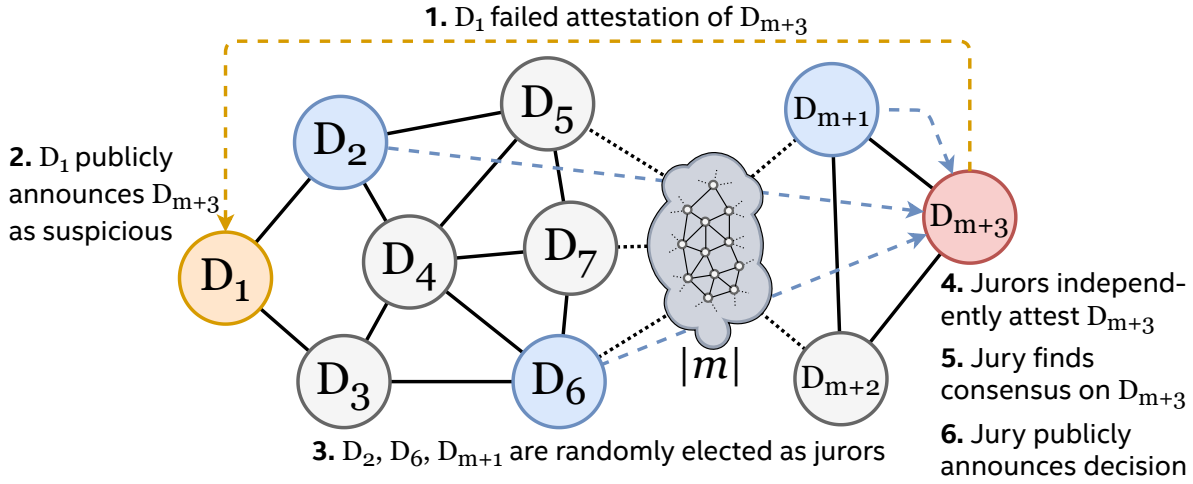



Figure 3.7: Example of one round of a jury-based attestation

Jury-based attestation tackles these challenges [2]. Figure 3.7 shows an exemplary round of a jury-based attestation. In Step 1, node $D_1$ attests $D_{m+3}$, e.g., when receiving critical data, and finds that $D_{m+3}$ cannot produce a valid attestation. Thus, in step 2, $D_1$ will announce to

the network, e.g., via broadcast, its suspicion that $D_{m+3}$ may be compromised. In step 3, the network will execute a distributed election to elect the *Jury*, in this case with three jurors. Each elected juror will then individually attest $D_{m+3}$ (step 4). Step 5 is the execution of a consensus protocol among the Jury to agree on the result, which is finally announced among the network in step 6. With this approach, the network's decision-making—if the suspected device is indeed compromised—is delegated to the Jury. This results in a constant overhead regardless of network size, aside the inherent communication overhead with larger networks.

In the context of ASSURED, jury-based attestation is invoked if there is a dispute regarding a particular attestation. For example, the verifier claims its attestation has failed and sends this results to the blockchain; yet, the prover claims it has actually produced a valid attestation. In such a case, the jury-based attestation provides the means to resolve this dispute in a secure manner.

Essentially, ASSURED Jury-based Attestation scheme acts as a second line of defense towards detecting possible misbehaviors during the execution of a (run-time) remote attestation process; either Configuration Integrity Verification or Control-flow Attestation. Consider, for instance, the case where the Verifier (e.g., IoT Gateway) has been compromised and, thus, records a falsified attestation report for a new device trying to enroll to the network. This essentially will result in a negative output while executing the "*Secure Device Enrollment*" policy. However, this sets the challenge ahead: *How do we make sure of the correctness of the Verifier that is responsible for attesting a newly joined device?* If the Verifier is compromised, when the attestation result is been broadcasted back to the Blockchain Peer Node, the Prover will overhear the message and report to the SCB of the wrong result. In this case, we need to have an additional step for identifying which of the participating entities is lying: **verify the evidence provided by both the Prover and Verifier so as to detect and revoke the misbehaving entity**.

# Chapter 4

# Adversarial and Trust Model

In this chapter, we describe the core adversarial model considered in ASSURED that covers threats and risks targeting different assets comprising the supply chain ecosystem. This is based on the detailed analysis and threat modelling documented in D1.4 [23] and D2.1 [28]. Thus, we will opt out from listing all of the identified vulnerabilities in the context of **software, network and physical attacks but we will focus on describing the endmost goal of an adversary in manipulating the execution correctness of the target device**. Essentially, allowing an adversary to use memory-handling related bugs for corrupting the control flow of the program, e.g., to violate the confidentiality of handled data or to control the underlying system.

## 4.1 System Model

The system is composed of a network infrastructure where the SCB spawns and governs a set of heterogeneous devices. Each device $\mathsf{D}$ is associated with three $\mathcal{T}$Cs: a TPM, serving as its trust anchor, an attestation agent ($\mathcal{AA}\mathsf{gt}$) to service inquiries, and a secure tracer ($\mathcal{T}\mathsf{rce}$) to measure the current state of the device's configuration (see Definition 1), ranging from its base software image, platform-specific information, and other binaries.

**Definition 1** (Config)**.** *A device's configuration set represents all of its uniquely identifiable objects (blobs of binary data).*

To proactively secure $\mathsf{D}$'s participation, SCB intermittently demands a $\mathsf{D}$ to re-measure parts (or all) of its configuration into its TPM's PCRs (either normal or NV-based) to justify its conformance with the currently compulsory policies. To track active PCRs, $\mathsf{D}$s maintain a separate list for normal (PCRS) and NV-based PCRs (NVPCRS). Each $\mathsf{D}$ also begins with three persistent TPM key handles: (*i*) a TPM storage key (SK) to enable the creation of AKs, (*ii*) the $\mathsf{D}$'s unique EK, which was agreed upon with the SCB during deployment, and (*iii*) the public part of SCB's EK to authenticate the SCB. Further, to authenticate traces from the tracer, we assume a secret symmetric hash key ($\mathsf{hk}$) shared between SCB and each $\mathcal{AA}\mathsf{gt}$ to enable $\mathcal{AA}\mathsf{gt}$s to authenticate their involvement in measurements. Note, however, that there are other possibilities to achieve a similar level of authenticity, e.g., using locality (see Section 3.2.1 in D3.1). Nonetheless, for the purpose of this deliverable, we instantiate the protocol by assuming a shared secret for authenticating the tracer. The hash key (shared secret) is assumed to reside in secure storage, inaccessible to any software, except for privileged code of the local $\mathcal{AA}\mathsf{gt}$.

On SCB, each device (besides its identity) is initially represented by the certified public part of its EK, the hash key shared with the device's $\mathcal{AA}\mathsf{gt}$, and two sets of mock PCRs, one representing

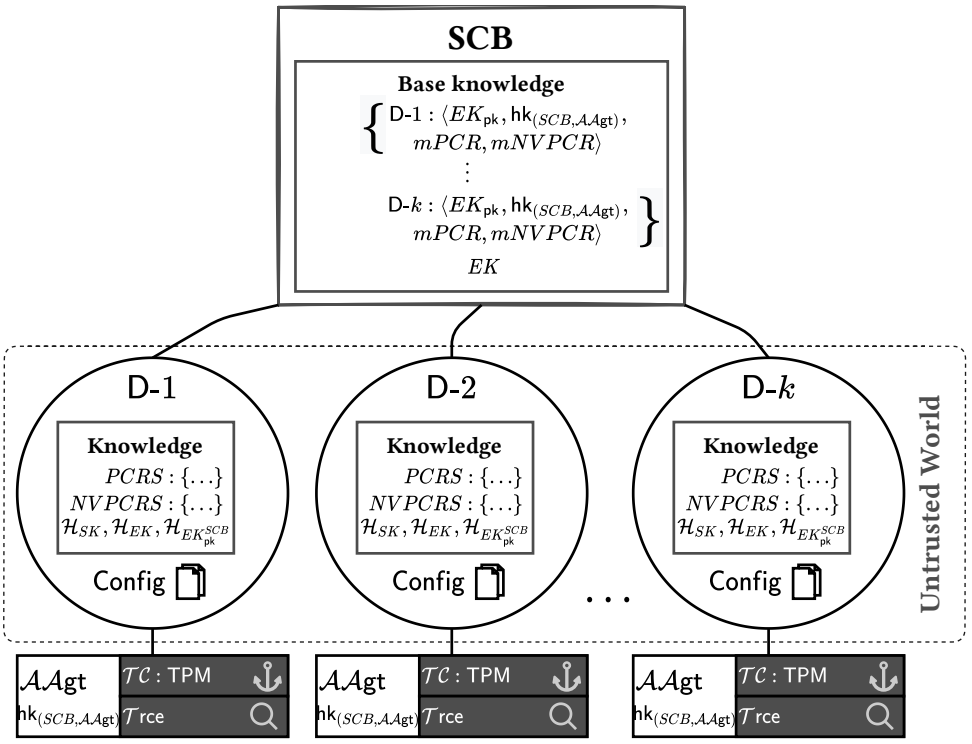



Figure 4.1: Conceptual (initial) system knowledge model.

the mock (emulated) state the device's normal PCRs (mPCR), and another of the NV-based PCRs (mNVPCR).

## 4.2 Adversarial Model

We focus on an adversary $\mathcal{A}$ that can launch a variety of **run-time attacks** that exploit program vulnerabilities to corrupt a program's control and data planes in an attempt to cause malicious behavior (see Figure 4.2 as an example). Therefore, we do not make any restricting assumptions for $\mathcal{A}$. However, we assume she does know the source code of the targeted application, allowing her to search for vulnerabilities that can be exploited; e.g., leverage buffer overflows for compromising the secrecy of the created keys [32]. Furthermore, we assume that she can fully control the input of the targeted application.

The most common attacks–also known as control-hijacking or code-pointer overwrite attacks–target a program's control plane and explicitly divert its execution path to execute unintended code. There are broadly speaking two variants: *code injection and code-reuse attacks*. With code injection, an adversary crafts and injects a payload into a device's memory and redirects a benign program's control flow to execute the payload. As an example, consider that in Fig. 4.2 an adversary has injected to alter the control-flow of the target device. However, being an early attacking methodology, code injection is easily defeated using common mechanisms such as Data-Execution Prevention (DEP), where executable memory regions cannot be written to during runtime. For the latter variant, however, it gets more difficult. Without injecting code, code-reuse attacks reuse existing program code to achieve some unintended behavior–using control plane maneuvers such as Return-Oriented Programming (ROP) [79], Jump-Oriented Programming (JOP) [11], and Counterfeit Object-Oriented Programming (COOP) [77].

With ROP and JOP, an adversary fabricates a new program by stitching together a chain of benign

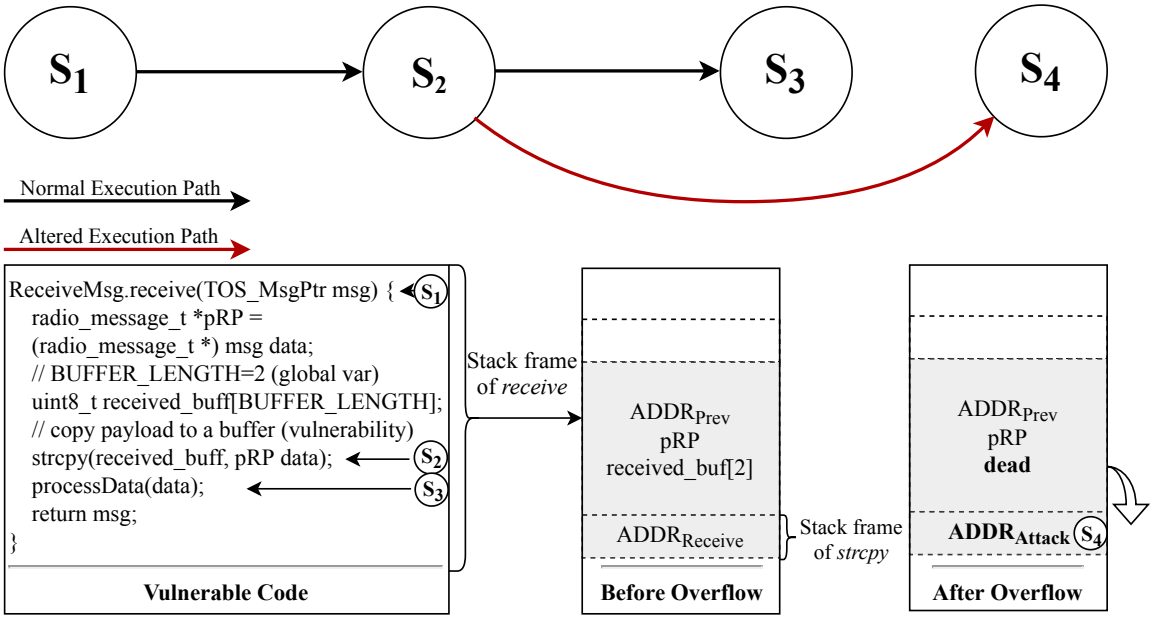



Figure 4.2: Normal and Altered Control Flows

pieces of existing code (called gadgets) that end in either function returns (ROP) or indirect jumps or function calls (JOP). The chain is then written into memory (e.g., through a stack overflow vulnerability), where, once it is triggered (e.g., from replacing a function's return address with that of the first gadget), the gadgets execute in sequence.

Besides runtime attestation schemes, using shadow stacks, canaries, Control-Flow Integrity (CFI), and Code-Pointer Integrity (CPI) remain prominent techniques aimed to locally mitigate code-reuse attacks. Whereas shadow stacks and canaries attempt to prevent the initial exploitation of a memory corruption vulnerability, CFI enforces a program to follow a legitimate path, and CPI ensures the integrity of a program's code pointers. However, both CFI and CPI remain insufficient for even more specialized attacks. For example, in COOP, an adversary uses a loop gadget to invoke a chain of C++ virtual functions which, due to C++ semantics and the fact that no code pointers are manipulated or injected, remains difficult to detect as being malicious unless we also consider the program's semantics [77]–which is often impractical. Thus, a more practical prevention strategy against COOP–besides employing defense techniques that prevent the initial hijacking–is concealing necessary information from the adversary [77], e.g., by applying fine-grained code randomization or address space layout randomization (ASLR).

So far, we have only reflected attacks that explicitly mislead program execution down imaginary or illegal paths, which we can detect by repeating a taken path in the respective program's CFG. However, a more subtle class of attacks exists, called *non-control-data attacks*, which disregards a program's control plane for its data plane. Without touching the execution path, such attacks corrupt data variables to make programs yield unexpected outputs or indirectly drive program execution down an unauthorized path. Since the path remains untouched, these attacks effectively subvert detection strategies predicated on the axiom that illegal execution paths include visibly unexpected turns in a reference CFG.

The attacking methodology behind non-control-data attacks is the application of Data-Oriented Programming (DOP) [19, 51] which we can call impure or pure, depending on whether we influence the program execution path (impure) or alter only data variables with no effect on the path (pure). For example, in Fig. 4.2, an adversary can corrupt some data variables in the $ReceiveMsg$ function (e.g., the $buff$ array) so as to change the result of the evaluation and, therefore, indi-

rectly lead the execution flow down the path of her choosing. Similarly, the adversary can corrupt the loop counter variable to modify the number of loop iterations when copying the bytes of the received payload to the internal memory buffer. Both attacks are impure as their presence affects the execution path. While the attacks will remain undetected when we differentiate legal paths from illegal only by whether they conform to the program's CFG, we could detect these attacks if we already suspect a certain evaluation result. Although such supplemental information is not always available, existing run-time attestation schemes inherently assume that verifiers have general knowledge about a program's expected behavior (e.g., loop execution and authentication information) to aid in the verification process. In ASSURED, for instance, the **ML-based verifier will train its classification model based on such knowledge on what is the expected behavior of an application**. This is usually given by the system administrator as a list of **trusted reference values** that is shared through the Blockchain infrastructure [22].

However, no amount of knowledge aids existing schemes [1,3,35,36,52,62,66,82,86] in detecting pure DOP attacks. Here an adversary, who knows the location of sensitive data, could mount a pure DOP attack and swap the data variable's destination address with the address of the sensitive data and thence unnoticeably exfiltrate the data.

In ASSURED, the goal is to enhance the attestation schemes presented in Chapter 5 with further traces and system data monitored by the implemented Tracer: In addition to recording the execution path, as is the current status of the ASSURED Tracer (Section 3.1), we will enable a Prover to also perform local integrity verification of all program critical variables which are identified through control-dependent variable discovery or annotated by the programmer. Data integrity will be continuously verified by maintaining a hash map of all critical variable, an whenever a variable is loaded, its value will be compared to its previously stored entry value. Any mismatch between a variable's load and store sites signifies a DOP attack and is propagated together with the program's execution path to enable verifiers to "detect" pure DOP attacks. This additional feature is currently under investigation and will be documented in the next version of this deliverable (D3.3).

**A running example:** As an example for better understanding an attacker's capabilities, consider the scenario of Secure Aerospace which comprise an ecosystem of thousand Control Units (CUs). Each CU must attest to each other in a web of interconnected nodes that propagate trust from edge nodes (usually sensors) to more central management nodes. The example follows the functionality of secure break landing in which the CUs contained in the tires gather information from the tire sensors and report them to the CU that handles the breaks. The break CU, depending on the status of the tire that is derived from the data sent by the tire CU, will take action and apply the breaks accordingly. Let a code snippet that runs in the tire CUs contain a buffer overflow vulnerability (Figure 4.2). In this example, the tire CU is the Prv and the breaks CU is the Vrf. The tire CU must attest to its running integrity before it is trusted by the breaks CU: if the properties received from the attestation are consistent with the $S_1, S_2, S_3$ control-flow state sequence, then the attestation process is successful and any data reported from the tire ECU is trusted. Any other set of properties that might result from any deviating control-flow like $S_1, S_2, S_4$ will be rejected and will flag the tire CU as compromised and untrusted.

The properties should contain just enough details to figure any control-flow changes. For instance in Figure 4.2, we demonstrate a basic code injection scenario where the control-flow is altered. The normal execution flow should start from the function entry ($S_1$ / ReceiveMsg.receive()), continue to a memory sensitive function ($S_2$ / strcpy()) and finish with a data processing function ($S_3$ / processData()). In the scenario of code injection, the attacker will overflow the stack frame up to the point that she can overwrite the return address of the function with the address of $S_4$ ($ADDR_{Attack}$) which is the function that she will redirect the execution flow to. The system we

propose, stores a healthy image of the control-flow in the form of its properties and afterwards, during the run-time of the application, it is able to detect any changes made from code injection attacks like the aforementioned one.

The code injection attacks belong in an attack vector that directly manipulates control data (return address), thus, changing the control-flow. In the same category, return oriented programming (RoP) aims to once again alter control data, but this time the attacker tailors his execution target by chaining already installed code in a specific sequence. This type of attacks, can easily be detected by techniques such as control-flow integrity (CFI), code-pointer integrity (CPI) [3] and control-flow attestation which our solution is based on. However, as aofrementioned, CFI and CPI are not able to catch non-control-data attacks which focus on corrupting data variables which are used to determine the control-flow that will be taken (e.g. variables in an "if" statement).

## 4.3 Trust Assumptions & Security Requirements

A common requirement in attestation protocols is that each device in the system is equipped with hardware support for remote attestation [3]. As described in Chapter 2, in ASSURED we leverage a TPM as part of the underlying Trusted Computing Base (TCB) for protecting both the **management of the keys** (used during the attestation process) but also for the **secure on-chain interaction with the ASSURED Blockchain infrastructure**; i.e., for downloading the attestation policy, through the deployed smart contracts, as well as for recording the result of each attestation process on the ledger, thus, creating a log history of device monitored states. This requirement is needed towards establishing the trusted computing base on which the attestation process will measure the integrity of the device/service while also communicating the results of this measurement securely. Furthermore, we require that the device is resistant to non-invasive attacks [28] (i.e, side-channel attacks, fault inducing attacks) while the system should be able to identify offline nodes that have been absent for a long time and could be victims of such exploitation attempts (micro-probing, reverse engineering) [80].

Unlike other proposed architectures, in ASSURED we follow the **zero trust** paradigm: *We do not rely on the assumption that a network perimeter is representative of a secure boundary and no implicit trust should be granted to users or services based solely on their physical or network location*. On the contrary, we treat each device as a possible point of intrusion (near-zero trust assumptions), thus, enforcing "*attestation policies as a code*" as granular as possible - when it comes to the resources to be attested (e.g., loaded binaries, specific software processes, etc.) and continuous monitoring and automated mitigation of threats. This also applies to the devices acting as the Verifier - we do not make any implicit assumptions on their trustworthiness (unlike existing schemes where the Verifier is considered trusted by default). In the case of a disagreement between the attestation report recorded by a Verifier and the traces reported by a Prover, ASSURED employs the newly designed Jury-based Attestation (Section 5.5), as a second line of defense, for identifying which of the participating entities is lying: **verify the evidence provided by both the Prover and Verifier so as to detect and revoke the misbehaving entity** (Section 7.6).

Overall, we do not assume the network to be trusted, making it necessary to use encrypted and verified communication protocols. On the Prover side, we assume only the components that are part of the TCB to be trusted (i.e., Tracer and TPM), therefore, $\mathcal{A}$ cannot manipulate them nor does it know the integrated cryptographic material. However, we do not trust any other component on the Prover side.

**The detailed (formal) definition of the trust model (besides the traditional data confidentiality, integrity and availability under weakened assumptions) considered in the entire ASSURED ecosystem has been established in D3.1 [21].** In what follows, we just give a quick summary of the core security, trust and privacy requirements that all security solutions need to achieve:

**R1. Memory Safety.** All accesses performed by loaded processes/services in the underlying memory map of the host device are "*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.

**R2. Control-flow Safety.** 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.

**R3. Type Safety.** All function calls and operations have arguments of correct type, thus, protecting against data-oriented exploits.

**R4. 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.

**R5. Anonymity** Ability of a platform to use a resource or service without disclosing its identity.

**R6. Pseudonymity** Ability of a platform to use a resource or service without disclosing its identity while still being accountable for that action

**R7. Unlinkability** Ability of a platform to make multiple uses of resources or services without others being able to link them together (i.e., infer mobility patterns in the context of the "*Smart Manufacturing*" use case).

**R8. Unobservability.** Ability of a platform to use a resource or service without others, especially third parties, being able to observe that the resource or service is being used.

## 4.4 Attestation Properties

We do not assume any component to be trusted, except of the aforementioned assumptions. Therefore, in order to trust the system ASSURED needs to verify different properties, as defined in details in D1.3 [26] and summarized as outpurs from the Tracer in Table 3.1. First the **static binary** needs to be attested, preventing static code injection, malicious software update and malware. This also includes the attestation of dynamically loaded libraries and hardware components. To attest the **dynamic properties**, the data and control flow need to be attested. To attest them, the following properties need to be considered: *code and data measurement during runtime aiming at disclose runtime injecting attacks, configuration files fed to attested programs during runtime, data exchanges with other entities such as interactive input during runtime, as well as system-level events such as software update history, reboots*.

# Chapter 5

# Design of ASSURED Attestation Schemes

Considering the adversary model and requirements specified in Chapter 4, we now need to describe in detail the workflow of actions and underlying crypto operations of the newly designed attestation schemes. Throughout the following section, we consider the symbols and abbreviations put forth in Annex 8.1.

## 5.1 Control Flow Attestation

Recall that the Control Flow Attestation System consists of two main actors:

- **Untrusted Prover**: that is the *prover*, who requires a verification of a CFG;
- **Untrusted Verifier**: that is the *verifier*, who verifies a CFG providing as result a measure of probability like:

  $$\{\texttt{benign} : p,\ \texttt{malicious} : 1 - p\}$$

  which can be used, according to a threshold, by the prober for applying possibly a countermeasure.

### 5.1.1 Trace Verification

As described in D3.1 [21] (Section 3.2.1), the verifier either holds a secret key in the secure runtime environment, with which it is able to verify that the traces are signed by the correct corresponding key by the prover or uses a locality protected signing protocol with a TPM to directly sign or verify the traces inside the TPM. The public part of the signing key is sent from the prover to the verifier through usage of the private ledger. The traces are double signed, first by the tracer application and then by the TPM-based wallet managing the ledger. Upon receiving signed traces the verifier first has to verify the signature with the corresponding key and reject wrongly signed traces.

1. The prover sends the public part of its signing key(s) $K_{pu}$ to the verifier which stores it on the private ledger. Note that the tracer application as well as the TPM-based wallet has to publish their key.
2. The secure execution environment inside the prover sends its securely collected and signed traces to the verifier (through the TPM-based wallet), where the signature is $s_T = SIGN_{K_{pr}}(T)$, where $SIGN$ is a (double) signature algorithm over the traces $T$ with key(s) $K_{pr}$ (i.e. either

signed by a TPM or by the shared key inside the TEE and again signed by the TPM-based wallet). Note, key $K_{pr}$ has to be verified against the key received through the policy.

3. The verifier verifies the signature by checking $VERIFY_{K_{pu}}(T, s_T)$ using its shared or aggregated key(s) $K_{pu}$ and the corresponding verification method $VERIFY$.

4. If the assertion succeeds the attestation starts, otherwise the attestation will be aborted.

### 5.1.2 Machine Learning

The system works in two separate phases:

1. **Training Phase**: during this phase benign memory traces are collected, for a specific application in a specific version. In the ASSURED context, a trusted party (e.g., an administrator) is collecting these benign traces and is storing them in a database where through a policy publication the verifier gets access to. A database is then built thanks to the feature extraction and preprocessing step. Using this dataset a machine learning system is trained on benign traces. This processes is represented in Figure 5.1;

2. **Detection Phase**: this second one is the operative phase, it starts when an untrusted party collect traces, signs them through TPM and sends them over a trusted channel to verifier. Once trusted party receives the data, verifies the sign: if it correct, feed them into the trained predictor, otherwise discards the traces and send back an error message. If traces are fed into the model, once obtained the prediction, it is sent back to the prober through the same trusted channel. This processes is depicted in Figure 5.2;

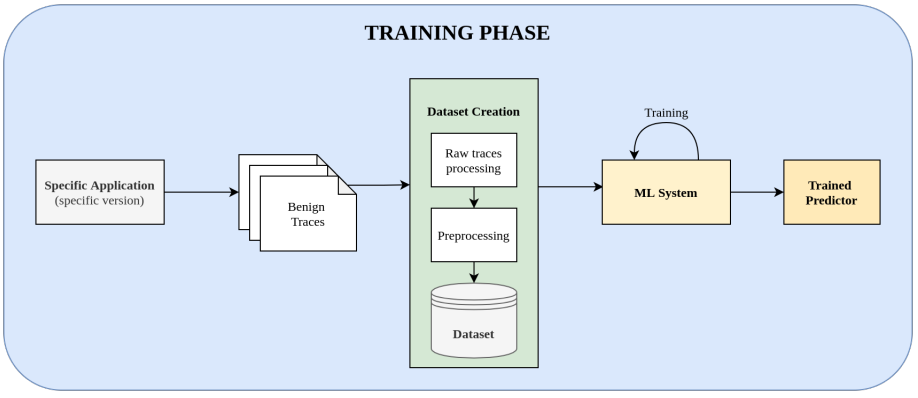



Figure 5.1: Example Control Flow Attestation System - Training Phase

### 5.1.3 Data Preprocessing

Data set creation is a fundamental step to go through for allowing the machine learning system to work. In particular, starting from raw traces, address are normalized by subtracting the base address. Nonetheless, dealing with memory addresses can be challenging since they can differ. A solution to this point is about mapping them in order to make them independent. Once it has been done, addresses are converted to symbols and inserted in a dictionary, assigning a number to every address, like:

$$\{0 : \texttt{0x0804bfb0}, 1 : \texttt{0xac937ce0}, \ ...\}$$

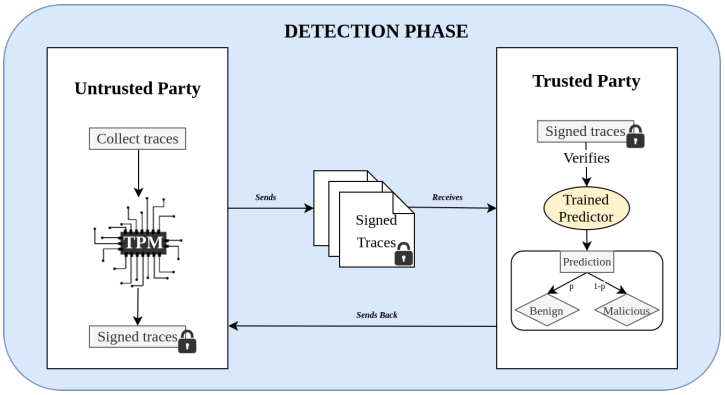



Figure 5.2: Example Control Flow Attestation System - Detection Phase

Furthermore, it has to be take into consideration that might happen that new not known addresses occur. For facing this problem, the best solution is about reserving some address for that.
In order to deal with all the different symbols, it has been used One-Hot Encoding approach. An example of that can be found in Table 5.1

| Symbol | One-Hot Encoding |
|---|---|
| symbol1 | 001 |
| symbol2 | 010 |
| symbol3 | 100 |

Table 5.1: Example of One-Hot Encoding with a vocabulary size of 3.

After having completed all the steps above, are prepared sequences of symbol of a specified length (i.e. time window), that will form the actual data set.

### 5.1.4 Training and Inference Phase

Since the task is about being able to distinguish between benign and malicious CFG, using a classifier is not a good idea. This is due to the fact that malicious data can look in very different ways according to the kind of attack, so, it is a better idea to train a learner only on benign data, so to be able to learn internal pattern of them. In such way, it is possible for the learner to state if a particular trace looks benign or not. Indeed, malicious trace are used only for evaluation.

With raising complexity of the software, also the benign behavior is harder to learn, making it more challengenging to detect benign traces. However, this can be compensated by a higher number of benign traces in the training dataset. In addition, for very complex scenarios, also the Neural Network that is used for the attestation could be adapted, e.g., by increasing the number of layers or the number of neurons in the individual layers.

Inference procedure requires trusted data from an untrusted party, so, for solving this main point, it is possible to use a Trusted Platform Module (TPM). Thanks to TPM data security and integrity is guaranteed: it manages to collect memory traces that have to be verified, and apply to them an unique signature. In particular, it is computed a cryptography hash based on data to be transmitted. Once completed this preparatory phase, data has to be transmitted over an encrypted channel to the verifier.

Once the trusted party receives messages from an untrusted party has to validate the signature received, if it is retained valid, then the verifier will start the preprocessing phase (Section 5.1.3) in

order to extract sequences from the memory logs received. After these steps, data are fed into the trained machine learning model for estimating the likely-hood of data being benign or malicious. Computed results are then sent back to the party who asked for the verification: analyzing the report received, untrusted party might decide to take actions in order to prevent malicious events.

### 5.1.5 Attestation Phase

The verifier then creates an attestation report as specified in D4.1 [22], signs it with its key (where the public parts is broadcasted via the private ledger) and pushes it on the private ledger for other participants to acquire and verify. The attestation report will include amongst other things, fields to identify and guarantee freshness (*ReportID*, *RecordedAt*, *NonceUsed*), a pointer to used traces (*ControlFlowDBPointer*), the signature to make the report unforgable (*Signature*) as well as the result of the attestation (*Result*).

**Algorithm 2** Remote Attestation - Attestation Phase

**Require:** $AttestationModel$,$AttestationPolicy$,$verifierKey$,$ControlFlowDBPointer$, $InSignature$
**Ensure:** $AttestationReport\{ReportID, RecordedAt, NonceUsed, ControlFlowDBPointer, AttestationSignature, Result\}$

$reportID \leftarrow getNextReportID()$
$nonceUsed \leftarrow generateNonce(AttestationPolicy)$
$recordedAt \leftarrow getTimestamp()$

**if** $\neg\, VERIFY(InSignature, ControlFlowDBPointer, nonceUsed) \parallel$
$\neg\, validateDataPointPointer(ControlFlowDBPointer)$ **then**
$Result \leftarrow \emptyset$
$attSignature \leftarrow SIGN(verifierKey, reportID, nonceUsed, recordAt, ControlFlowDBPointer, Result)$
**return** $AttestationReport\{reportID, recordedAt, nonceUsed, ControlFlowDBPointer, attSignature, Result\}$
**end if**
$Result \leftarrow AttestationModel.predict(*ControlFlowDBPointer)$
$attSignature \leftarrow SIGN(verifierKey, reportID, nonceUsed, recordAt, ControlFlowDBPointer, Result)$
**return** $AttestationReport\{reportID, recordedAt, nonceUsed\ ControlFlowDBPointer, attSignature, Result\}$

## 5.2 Direct Anonymous Attestation

### 5.2.1 ASSURED DAA Scheme

In our ASSURED DAA scheme, it is only the Privacy CA as the Issuer and the TPM that we assume as trusted; as aforementioned, **the Privacy CA is responsible for authenticating edge nodes through the JOIN protocol**. In our context, **edge nodes are the combination of a *host*, that is the normal computing platform "normal world", and the TCB that executes in the**

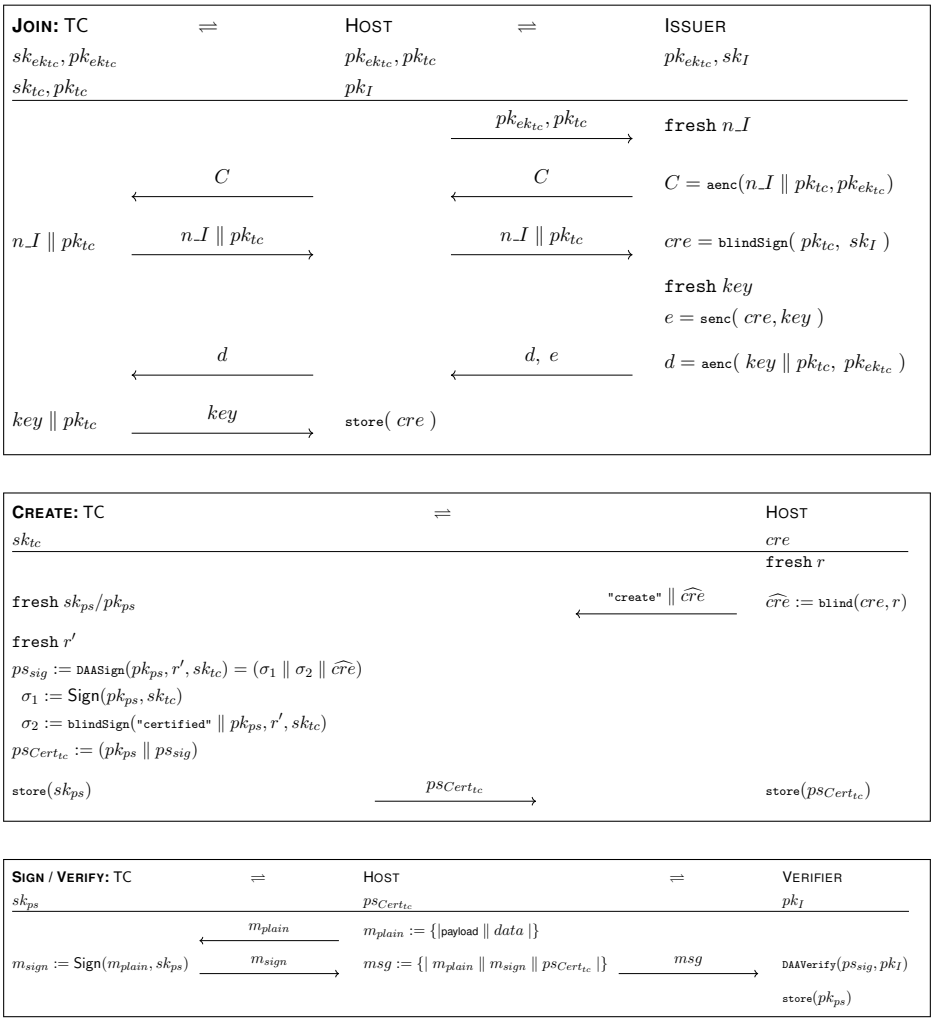



Figure 5.3: High-level Overview of the ASSURED DAA Protocol Interfaces.

**"secure world"**; together they form the device which we refer to from this point onwards as the edge node.

Figure 5.3 defines the implementation of our ASSURED DAA protocols. We first describe each protocol execution by defining the responsibility of all system actors and separate the roles of the TPM-based Wallet and host. This allows us to better reason against the required functionality of a TPM. Then, in Section 5.2.4, we provide a detailed documentation of the underlying crypto primitives of the various DAA phases.

### 5.2.2 Device Registration

The first step for a node acquiring its certificates (after the correct execution of the Configuration Integrity Verification towards the secure enrollment of only those nodes that are at a "correct state") consists of two phases: SETUP for generation of keys and the enrollment phase to the Privacy CA (JOIN). We assume that during manufacture time, the TPM will have a unique DAASeed installed, a non-monotonic counter $cnt$, and the hardware will be endorsed by the manufacturer through means of burning the endorsement key pair: $sk_{ek_{tc}}$ / $pk_{ek_{tc}}$ into the TPM. For the SETUP phase the Privacy CA publishes its public key $pk_I$ and the security parameters $K_I$. A

node's TPM generates a DAA key pair: $sk_{tc}$ / $pk_{tc}$ using $K_I$, and publishes its public key $pk_{tc}$. The TPM then releases the public keys $pk_{ek_{tc}}$ and $pk_{tc}$ to the fog node.

The details of the JOIN protocol are shown in Figure 5.3. By the end of the protocol the newly registered platform will have acquired a VID Certificate ($cre$) certifying that the node has a valid TPM which has been enrolled with the Privacy CA. To initiate the JOIN protocol, a node sends the Privacy CA its public key indicating it wants to join the network (Step 1). The Privacy CA responds to the vehicle with a fresh challenge $C$ which only the valid TPM can open. The node then forwards $C$ to its TPM via a secure I/O (Step 2). The TPM opens the challenge, confirms its validity, and sends the response to the host node, which in turn, responds to the Privacy CA with the recovered data items (Step 3). The Privacy CA verifies the received response, confirming that the device possesses a valid TPM Wallet. Following this verification, the Privacy CA creates the credential $cre$, and a fresh symmetric session $key$. The credential $cre$, encrypted with the session $key$, is sent to the node, along with an encryption of $key$ intended for the TPM, as $e$ and $d$ respectively in Step 4. Finally, the node uses the TPM to decrypt $d$, recovering the $key$. The TPM verifies the validity of $d$ and then releases $key$ to the node (step 5). The node can then decrypt $e$ (using $key$) to recover the certificate $cre$. Finally, it verifies $cre$ using $pk_I$ and stores it for future use.

By the end of this protocol, if successful, the node is an authenticated and legitimate member of the fog cluster, and ready to register to any of the provided services including the Blockchain-based data and attestation policy sharing.

### 5.2.3 Anonymous Credentials Creation

The creation of pseudonyms (CREATE in Figure 5.3) lies within the device nodes, allowing the shift of trust from a third party to locally within the end-points. This is made possible by all nodes being equipped with a TPM, that is responsible for generating the pseudonyms in an environment that enables protected execution, isolation and secure storage.

Creating new pseudonyms for a nodes does not require any external network communication, and all message exchanges in the CREATE protocol take place over secure I/O between the host and TPM. To initiate the creation process the host blinds the $cre$ with freshly generated random nonces, and sends a "create" request to the TPM with $\widehat{cre}$. Alternatively, the node can choose not to "blind" its credential and create pseudonyms which are linkable. While this is bad practice, it does demonstrate that anonymity, pseudonymity, unlinkability and unobservability are under the control of the fog node (based on the policies already circulated by the $\mathcal{O}$rc). Upon receipt of the pseudonym creation request, the TPM creates a fresh pseudonym key pair $sk_{ps}/pk_{ps}$ and fresh random $r$. Using the `DAASign` algorithm the TPM creates two signatures: $\sigma_1$ - the public pseudonym key signed with the DAA secret key $sk_{tc}$, and $\sigma_2$ - a blind signature of the certified $pk_{ps}$ key; ensuring the generated pseudonyms are not linkable. $\sigma_1$ is a "link token" which is created for the purpose of revocation, discussed in Section 6. Once the pseudonym signature is produced, $ps_{sig}$, the pseudonym certificate, $ps_{Cert_{tc}}$, is produced that is constructed from the public pseudonym key $pk_{ps}$ and the pseudonym signature. The TPM concludes by storing the generated pseudonym secret key $sk_{ps}$ and returns the pseudonym certificate to the host for use in networking communication.

By the end of this protocol a device can use its pseudonyms anonymously share attestation reports and other operational data with the other network participants (through the ASSURED Blockchain infrastructure) whilst still being held accountable for its use of the services.

### 5.2.4 ASSURED DAA Crypto Primitives

The first RSA-based DAA scheme was proposed in 2004 by Brickell, Camenisch, and Chen. Later, Brickell, Chen and Li proposed the first ECC-DAA scheme based on symmetric pairings [13, 14]. In what follows, for simplicity, we put forth the crypto specifics of the DAA protocol focusing mainly on the operations revolving around the correct creation and usage of the DAA Key [18]. The crypto details of the DAA Enhanced version including also the creation of the *pseudonymous* credentials can be found in [83]. We start by explaining the setup algorithm.

**Setup algorithm**: It takes the security parameter $1^t$ as input, then the algorithm executes as follows:

1. Generate the commitment parameters $par_c = (G_1, G_2, G_T, q, P_1, P_2, \hat{h}, H_1, H_2)$ that consist of:
   - Two random generators $P_1$ and $P_2$, and define the three groups $G_1 = \langle P_1 \rangle$, $G_2 = \langle P_2 \rangle$, and $G_T$ such that $\hat{h} : G_1 \times G_2 \rightarrow G_T$ where $\hat{h}$ is a pairing function.
   - Two hash functions $H_1 : \{0,1\}^* \rightarrow G_1,\ H_2 : \{0,1\}^* \rightarrow \mathbb{Z}_q$ where $q$ is a sufficiently large prime.
2. Generate Signature and verification parameters $par_S = (H_3, H_4)$: Consist of two additional hash functions namely $H_3 : \{0,1\}^* \rightarrow \mathbb{Z}_q$ and $H_4\{0,1\}^* \rightarrow \mathbb{Z}_q$.
3. Let $K_k$ be an issuer value derived by the issuer public values, the issuer parameters $par_I = (ipk_k, K_k)$ for each $i_k \in \mathcal{I}$:
   - Set the issuer private key $isk_k = (x, y)$ where $x$ and $y$ are two random integers in $\mathbb{Z}_q$.
   - Compute $X = [x]P_2 \in G_2$ and $Y = [y]P_2 \in G_2$, and let $ipk_k = (X, Y)$ be the issuer public key.
4. Generate TPM parameters $par_T = (\mathbf{PK}_h)$ for each TPM embedded in some host $h \in \mathcal{H}$: The TPM generates a public and private key pair **(PK, SK)** for the associated endorsement key. TPM also generates a private secret value called the DAAseed.
5. Finally all the system public parameters $(par_c, par_S, par_I, par_T)$ are published.

Note that each TPM has a single DAA seed, but can create many DAA secret keys, even associated to a single user. The TPM DAA secret key is generated by using DAAseed, $K_k$ and an additional number 'cnt' as inputs.

**The Join Protocol** This works as follows:

1. The issuer sends a request to the TPM using the following steps:
   - Chooses $k_M \leftarrow MK$
   - $c_I = ENC_{\mathbf{PK}}(k_M)$
   - $n_I \leftarrow \{0.1\}^t$
   - Sends a $comm_{req} = (c_I, n_I)$ and $str = X \| Y \| n_I$ to the TPM.
2. The TPM responds as follows:
   - Chooses a DAA TPM secret key $sk_T \leftarrow PRF(DAAseed \| K_I \| cnt)$

- $k_M = DEC_{\mathbf{SK}}(c_I)$, if $k_M = \perp$ then aborts.
- $str = X \| Y \| n_I$
- $Q_2 = [sk]_T P_1$
- The TPM chooses $u \leftarrow \mathbb{Z}_q$ uniformly at random, then computes
  - $U = [u]P_1$
  - $v = H_2(P_1 \| Q_2 \| U \| str)$
  - $w = u + sk_T(\mod q)$
  - $\gamma \leftarrow MAC_{k_M}(P_1 \| Q_2 \| v \| w)$
- The TPM sends the commitment $comm = (Q_2, v, w, \gamma, n_I)$ to the issuer.

3. The issuer checks that:
   - $n_I \in comm_{req}$, otherwise abort.
   - Computes $\gamma' = MAC_{k_M}(P_1 || Q_2 || v || w)$, if $\gamma' \neq \gamma$ then abort.
   - The issuer computes
     - $U' = [w]P_1 - [v]Q_2$
     - $v' = H_2(P_1 \| Q_2 \| U' \| str)$
     - If $v \neq v'$ then aborts.
   - $\forall sk'_T \in$ Roguelist, check if $Q_2 = [sk'_T]P_1$, then abort.
   - The issuer creates a credential as follows:
     - Chooses $r \leftarrow \mathbb{Z}_q$
     - Computes $A = [r]P_1,\ B = [y]A,\ C = [x]A + [rxy]Q_2$ and sends the credential $cre = (A, B, C)$ to the TPM.
4. The TPM computes $D = [sk_T]B$ and sends $D$ to the host.
5. The host verifies that $\hat{h}(A, Y) = \hat{h}(B, P_2)$, and $\hat{h}(A + D, X) = \hat{h}(C, P_2)$, otherwise aborts.

**The sign/ verify protocol**

1. The verifier $\mathcal{V}$ chooses a string $n_V \in \{0, 1\}^t$ and sends it to the host.
2. The host starts signing as follows:
   - Checks if bsn=$\perp$, then chooses an element $J \leftarrow G_1$, else $J = H_1(bsn)$
   - Chooses $l \leftarrow \mathbb{Z}_q$
   - Sets
     - $R = [l]A$
     - $S = [l]B$
     - $T = [l]C$
     - $W = [l]D$
   - Outputs $c = H_3(R \| S \| T \| W \| n_V)$ and sends $(c, J, S, msg, bsn)$ to the TPM.
3. The TPM proceeds with signing as follows:

- Computes $K = [sk_T]J$
- Chooses a string $n_T \leftarrow \{0,1\}^t$ and an integer $r \leftarrow \mathbb{Z}_q$
- Computes
  - $R_1 = [r]J$
  - $R_2 = [r]S$
  - $str = J\|K\|bsn\|R_1\|R_2$
  - $h = H_4(c\|msg\|str\|n_T)$
  - $s = r + h.sk_T \mod q$
- The TPM sends $(K, h, s, n_T)$ to the host.

4. The host outputs a signature $\sigma = (R,\ S,\ T,\ W,\ J,\ K,\ h,\ s,\ r,\ n_V,\ n_T)$ and sends it to the verifier.

5. 
   - The veifier checks that:
     - $\forall sk'_T \in$ Roguelist, if $K = [sk'_T]J$ return false.
     - If bsn $\neq \perp$ and $J \neq H_1(bsn)$ return false.
     - $\hat{h}(R, Y) = \hat{h}(S, P_2)$ and $\hat{h}(R + W, X) = \hat{h}(T, P_2)$, else return false.
   - The verifier computes
     - $R'_1 = [s]J - [h]K$
     - $R'_2 = [s]S - [h]W$
     - $c' = H_3(R\|S\|T\|W\|n_V)$
     - $str' = J\|K\|bsn\|R'_1\|R'_2$
     - $h' = H_4(c'\|msg\|str'\|n_T)$
   - If $h' \neq h$ return false.

## 5.3 Configuration Integrity Verification

As described in Section 2.1.1, the core objective of the Zero Touch Configuration (ZTC) functionalities is twofold: (*i*) to enable the Privacy CA and Security Context Broker (SCB) to securely enroll devices, and (*ii*) to enable enrolled devices to attest to their correct configuration during runtime when requested through a deployed attestation policy; i.e., executing configuration-oblivious inter-device Configuration Integrity Verification (CIV). These ZTC objectives are achieved by two separate protocols, namely: *secure enrollment*, and *Attestation by Proof*. During device enrollment, the Privacy CA requests and verifies the creation of a restrained asymmetric Attestation Key (AK) pair on a device's trusted security anchor (TPM), where the *use* of the AK is certified to require that a specified selection of TPM Platform Configuration Registers (PCRs) contain authorized aggregated Trusted Reference Values (TRVs), thus ensuring that the device's secret AK ($AK_{\mathsf{sk}}$) can be used only if that device is in a correct (authorized) state. This is based on the **policy-protected key usage** attribute described in D3.1 [21] as part of the overall ASSURED secure key management landscape. Once the public part of a device's AK has been validated and published on the smart contract (see D4.1 [22]), any other device can ascertain its correctness in an *oblivious* manner using a simple challenge-based remote attestation protocol (*Attestation by Proof*), where the verifier V sends the prover P a fresh challenge (nonce), and if P replies with a signature over the challenge using its secret AK, then V knows that P is in a correct state.

### 5.3.1 High-Level Overview

By conditioning a device's ability to attest on whether its configurations are authorized by SCB, the Oblivious Remote Attestation (ORA) scheme (see Figure 5.4), which enhances the *Attestation by Proof* functionality, enables arbitrary devices to verify the integrity of other devices while remaining oblivious to what constitutes their state. We preserve privacy as no exchange of platform or state details is required among devices.

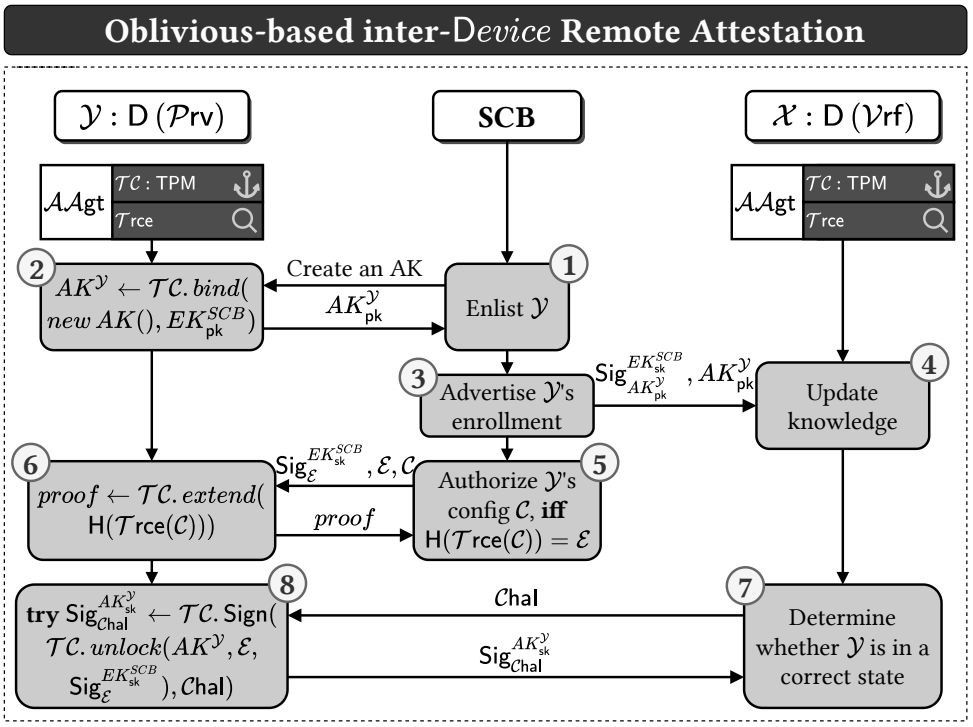



Figure 5.4: Holistic work-flow of the ORA protocol.

The scheme's work-flow (Figure 5.4) is as follows. When a new device, say $\mathcal{Y}$, wishes to join, SCB (after being notified by the Privacy CA) requests it to first create an AK (Step 1), $AK^{\mathcal{Y}}$, using its TPM, and lock it to a *flexible policy* bound to SCB's EK, ensuring that only SCB can permit $AK^{\mathcal{Y}}$'s use (Step 2). Once $AK^{\mathcal{Y}}$ is created, verified, certified, and published on the smart contract (Steps 3), any subscribers of attestation key creation events can include $\mathcal{Y}$ as an eligible (enrolled) peer (Step 4). Then, to enable $\mathcal{Y}$ to prove its configuration correctness using its AK, SCB authorizes (signs using $EK^{SCB}_{\mathsf{sk}}$) a policy digest $\mathcal{E}$ over $\mathcal{Y}$'s currently acceptable configuration state, and sends it to $\mathcal{Y}$ (Step 5). Given the update request, $\mathcal{Y}$ measures its actual configuration into its TPM (Step 6). When another device, $\mathcal{X}$, wants to determine whether $\mathcal{Y}$ is in a trusted state, it sends a challenge $\mathcal{C}\mathsf{hal}$ (e.g., a nonce) to $\mathcal{Y}$ (Step 7). If, and only if, $\mathcal{Y}$'s configuration measurements corresponded to what SCB authorized, access is granted to use $AK^{\mathcal{Y}}_{\mathsf{sk}}$ to sign $\mathcal{C}\mathsf{hal}$ (Step 8). Note that steps 5 and 6 can repeat any number of times to change $\mathcal{Y}$'s trusted configuration state.

### 5.3.2 Zero-Tough Integrity Verification Building Blocks

#### 5.3.2.1 AK Provisioning

Figure 5.5 shows the exchange of messages between the different actors in the AK-creation protocol, where SCB is portrayed as an oracle who supplies input to and verifies output from

the device ($\mathcal{X}$). Note, however, that in practice this communication will occur indirectly through the use of smart contracts. Nonetheless, the protocol begins locally on SCB, where a policy digest is computed over the Command Code (CC) of `TPM2_PolicyAuthorize` (specified in the specification) and the name of SCB's EK. Note that such policies are called *flexible* since any object $\phi$ bound to the policy can only be used in a policy session with the TPM after fulfilling some policy (e.g., that the PCRs are in a particular state) which the policy's owner (SCB in our case) has authorized (signed). The policy digest, together with a template describing the key's characteristic traits (e.g., attributes and type), is then sent to $\mathcal{X}$, who forges the AK within its TPM. Besides producing and returning the AK object, the TPM also returns a signed ticket over the object to denote that it was created inside the TPM. This "creation" ticket, together with the newly created AK object and $\mathcal{X}$'s EK, are then passed to `TPM2_CertifyCreation`, where the TPM vouches that it was involved in producing AK (if the ticket holds) by signing the AK object along with some internal state information. Then, due to AK's flexibility, where AK can remain the same throughout $\mathcal{X}$'s lifetime, it is stored persistently in TPM NV memory (using `TPM2_EvictControl`). Finally, $\mathcal{X}$ presents the AK and certificate to SCB, who verifies the certificate's signature and scrutinizes its details to ascertain that the AK was created legitimately. If everything holds, $\mathcal{X}$ is permitted to enter the network.

#### 5.3.2.2 Remote PCR Administration

Note that although normal (static) PCRs cannot be reset during run-time, an NV slot defined to imitate a PCR can be deleted and recreated depending on how it is created. We, therefore, require that NV PCRs be created with a flexible policy, similar to AKs, such that *only* upon deletion requests authorized by SCB can the NV index be undefined. Further, to ensure that only policies specifically authorized to undefine the NV index can be used to undefine it, the CC of `TPM2_NV_UndefineSpaceSpecial` is included as part of the NV PCR's authorization policy. Thus, the authorization policy of NV PCRs becomes:

$$\texttt{TPM2\_PolicyAuthorize}(\mathsf{name}(EK_{\mathsf{pk}}^{SCB})) \wedge \texttt{TPM2\_PolicyCommandCode}(CC_{NV\_UndefineSpaceSpecial})$$

Finally, to prevent a D from undefining arbitrary NV indices, SCB also embeds into the policy a Command Parameter (CP) digest over the name of the NV index to be undefined, which restricts the policy only to work on the correct NV index.

Figure 5.6 contains the message exchanges to attach PCRs to a D ($\mathcal{X}$). For normal PCRs, $\mathcal{X}$ is simply informed about which PCR to use and both SCB and $\mathcal{X}$ update their knowledge accordingly, i.e., SCB adds it to $mPCR^{\mathcal{X}}$ and $\mathcal{X}$ to its $PCRS$ structure. Otherwise, for NV-based PCRs, SCB sends: (*i*) a NV identifier, (*ii*) a NV template which describes the H algorithm and attributes of the NV slot, e.g., that modifications must happen using `TPM2_NV_Extend` (to imitate a PCR), and that a policy is required to delete the index, (*iii*) an authorization policy which gives SCB the exclusive right to authorize the deletion of the index, and (*iv*) an initial value (IV) to extend. The IV is necessary since newly-created NV indices cannot be read (or certified) until they have been written. Thus, to allow $\mathcal{X}$ to certify it, SCB sends an initial (deterministic) IV which $\mathcal{X}$ must extend the newly created NV-based PCR (NVPCR) with. Once the NVPCR is created, extended, and certified, $\mathcal{X}$ sends the certification details to SCB, who verifies that: (*i*) the certified information is of a TPM generated structure, (*ii*) the NVPCR's value is $\mathsf{H}(0 \ldots 0 \,\|\, IV)$, (*iii*) the NVPCR's name is as expected (i.e., that it contains the specified attributes and is bound to the correct authorization policy), (*iv*) the certificate is authentic. If everything holds, then the NVPCR was created correctly and is added to $mNVPCR^{\mathcal{X}}$ on SCB.

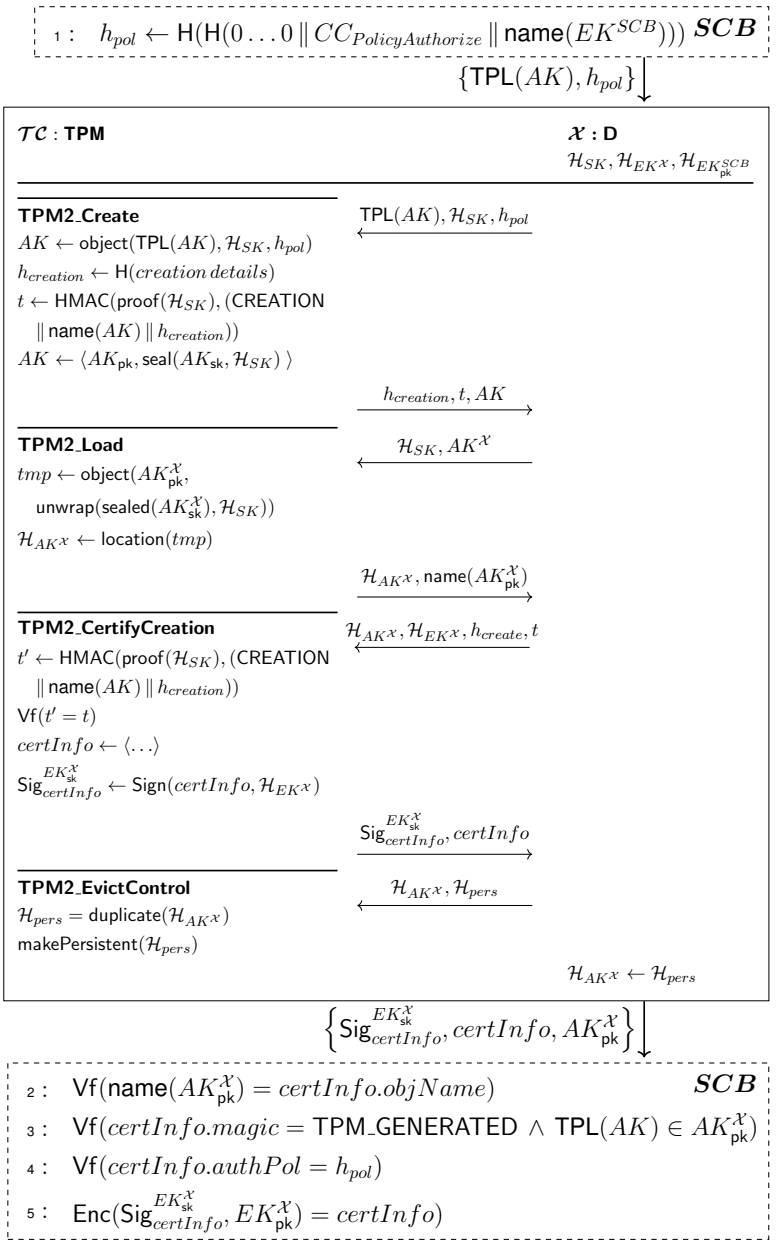



Figure 5.5: AK creation

**Algorithm 3** Authorizing NV index deletion

**Input:** $n, idx, \mathcal{H}_{EK}, mNVPCR$

**Output:** $\left\{ idx, h_{cp}, \mathsf{Sig}_{aHash}^{EK_{\mathsf{sk}}^{SCB}}, h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}_{\mathsf{H}(h_{pol})}^{EK_{\mathsf{sk}}^{SCB}} \right\}$

$h_{pol} \leftarrow \mathsf{H}(\mathsf{H}(0 \ldots 0 \,\|\, CC_{PolicySigned} \,\|\, \mathsf{name}(\mathcal{H}_{EK})))$

$\mathsf{Sig}_{\mathsf{H}(h_{pol})}^{EK_{\mathsf{sk}}^{SCB}} \leftarrow tpm.\mathsf{Sign}(\mathsf{H}(h_{pol}), \mathcal{H}_{EK}))$

$h_{cp} \leftarrow \emptyset$

$\forall \langle \mathcal{H}, h, \mathsf{name}(\mathcal{H}) \rangle \in mNVPCR :$

  **if** $\mathcal{H} = idx$ **then**

    $h_{cp} \leftarrow \mathsf{H}(CC_{NV\_UndefineSpaceSpecial} \,\|\, \mathsf{name}(\mathcal{H}) \,\|\, \mathcal{H}_{PPS})$

$aHash \leftarrow \mathsf{H}(n \,\|\, 0 \,\|\, h_{cp})$

$\mathsf{Sig}_{aHash}^{EK_{\mathsf{sk}}^{SCB}} \leftarrow tpm.\mathsf{Sign}(aHash, \mathcal{H}_{EK})$

**return** $idx, h_{cp}, \mathsf{Sig}_{aHash}^{EK_{\mathsf{sk}}^{SCB}}, h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}_{\mathsf{H}(h_{pol})}^{EK_{\mathsf{sk}}^{SCB}}$

To detach a normal PCR, SCB simply informs $\mathcal{X}$ about which PCR to remove from its $PCRS$. For a NVPCR, however, the process is more tricky. To detach a NVPCR, SCB requests $\mathcal{X}$ to start a fresh policy session and return the session's TPM-generated nonce ($n$). With $n$ and one of $\mathcal{X}$'s NVPCRs ($idx$), SCB runs Algorithm 3, which: (*i*) authorizes a policy ($h_{pol}$) requiring that `TPM2_PolicySigned` be executed with a digest ($aHash$) signed by SCB (lines 1 and 2), (*ii*) signs $aHash$ (as described in Part 2 of the TPM 2.0 specifications), which is over $n$, an expiration (set to 0), and a CP digest, $h_{cp}$ ($cpHash$ in Algorithm 6), where $h_{cp}$ restricts the session to the `TPM2_NV_UndefineSpaceSpecial` command (as required by the NV index's authorization policy, see Figure 5.6) with $idx$ as a parameter (lines 3 to 8). Thus, to perform the deletion, $\mathcal{X}$: (*i*) verifies that $h_{pol}$ was signed by SCB, (*ii*) executes `TPM2_PolicySigned` which: (*ii-a*) updates the session's policy digest to indicate that the command was executed with a digest signed by SCB, and (*ii-b*) sets the session's $cpHash$ to $h_{cp}$, (*iii*) runs `TPM2_PolicyAuthorize` with $h_{pol}$, which, if it matches the session's policy digest, sets the session's digest to state that a policy authorized by SCB was fulfilled, (*iv*) runs `TPM2_PolicyCommandCode` to restrict the session's CC, (*v*) runs `TPM2_NV_UndefineSpaceSpecial` which deletes the NV index *if everything holds (i.e., the NV index's authorization policy is fulfilled)*, (*vi*) removes the NV index from its local knowledge.

Note that the nonce ($n$) is just a random and unauthenticated number, and the authorized policy generated by SCB carries no information that restricts it to $\mathcal{X}$'s TPM. Thus, if two TPMs $A, B$ have the same NV index defined (with the same attributes and bound to the same authorization policy), then the session could belong to either $A$ or $B$, and the authorized policy would succeed. There are two easy solutions to this issue: (*i*) create an *authentic* channel between $\mathcal{X}$ and SCB using software, or (*ii*) include additional (unique) data when creating a NV index such that an authorized policy is unique to a specific D.

#### 5.3.2.3 Supervised Updates

To enforce a configuration update, SCB uses the mock PCRs associated with $\mathcal{X}$ to emulate what the *expected* (thereby *trusted*) cascading effect of the update's measurement is and includes the result in a new policy. For example, let $r$ be a resource on $\mathcal{X}$ (also known to SCB) and $i$ be a PCR attached to $\mathcal{X}$ which will house $r$'s measurement. On SCB, the current (mock) value of $i$ is assumed to be $v$. Thus, the expected value in PCR $i$ after measuring $r$ is $\mathsf{H}(v \,\|\, \mathsf{H}(r))$. The measurement-update protocol is shown in Figure 5.8, where, given a Fully Qualified Path Name

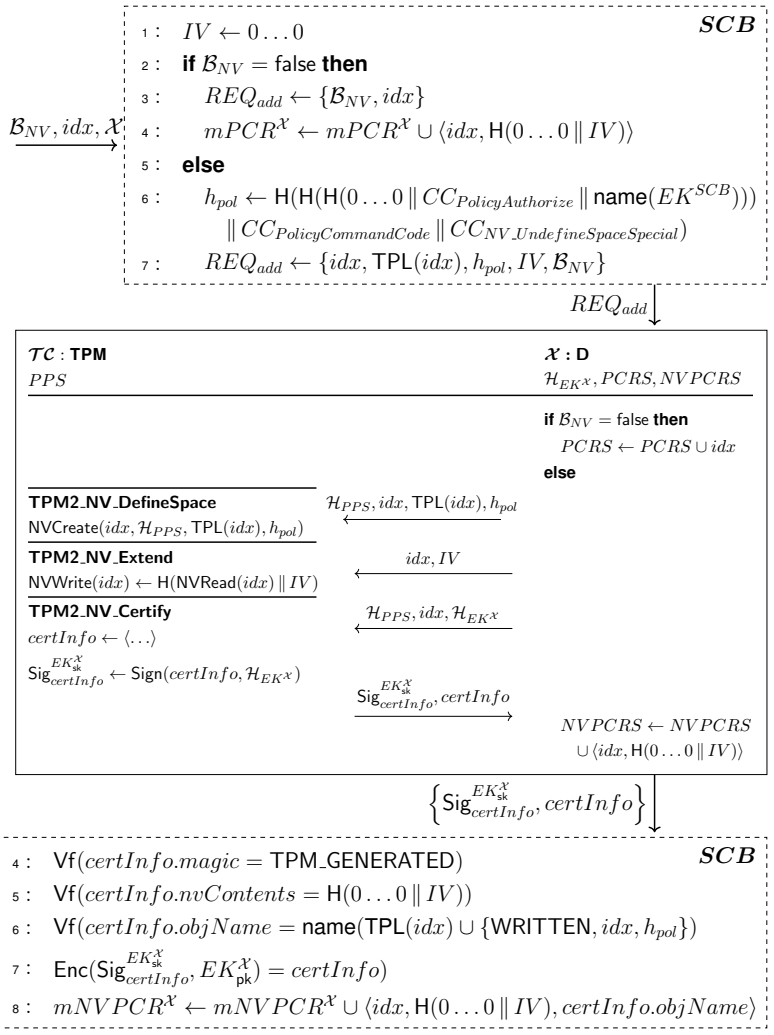



Figure 5.6: Attaching a normal or NV-based PCR

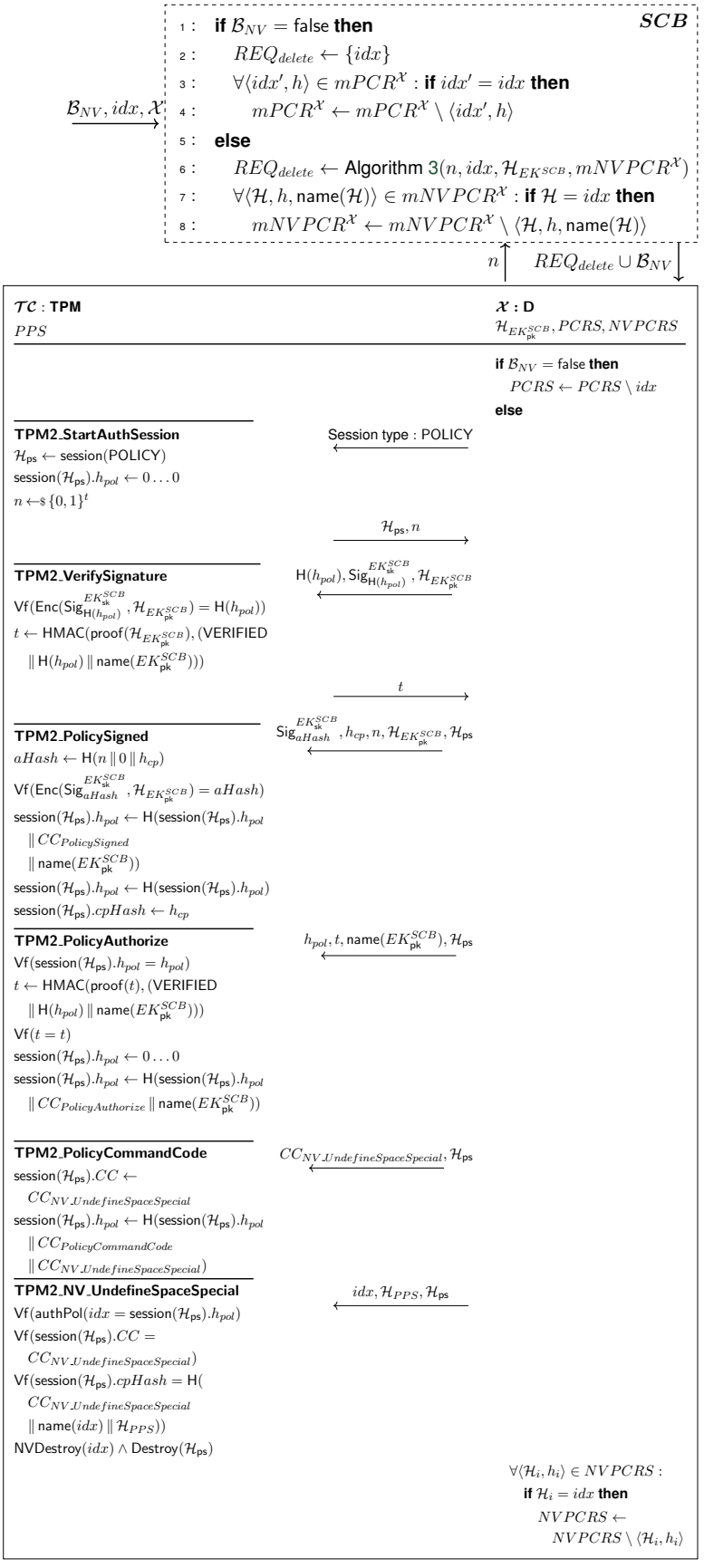



Figure 5.7: Detatching a normal or NV-based PCR

(FQPN) of some configuration on a $\mathcal{VF}$ ($\mathcal{X}$) and a target PCR ($idx$), SCB locally measures and authenticates (using the shared secret between SCB and $\mathcal{X}$'s $\mathcal{AA}$gt) the configuration measurement and then applies Algorithm 4 to compose and authorize the expected policy digest using the mock PCRs associated with $\mathcal{X}$. The authorized policy and details for $\mathcal{X}$ to perform the measurement locally (i.e., FQPN, PCR type, and $idx$) are then sent to $\mathcal{X}$. On $\mathcal{X}$, $\mathcal{AA}\text{gt}^{\mathcal{X}}$ intercepts the update request, measures FQPN using $\mathcal{T}$rce, and authenticates the measurement. $\mathcal{X}$ then proceeds to use its TPM to verify whether the supplied policy digest was signed using SCB's EK. If the signature is correct, the TPM returns a ticket denoting that the TPM has verified the policy digest's correctness.

**Algorithm 4** Composing AK policy update requests

**Input:** $idx, \mathcal{B}_{NV}, h_{update}, mNVPCR, mPCR, \mathcal{H}_{EK}$
**Output:** $\left\{h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}^{\mathsf{k}}_{\mathsf{H}(h_{pol})}, idx, \mathcal{B}_{NV}\right\}$
  **if** $\mathcal{B}_{NV} = \mathsf{true}$ **then**
    $\forall\langle\mathcal{H}, h, \mathsf{name}(\mathcal{H})\rangle \in mNVPCR$ :
      **if** $\mathcal{H} = idx$ **then** $h \leftarrow \mathsf{H}(h \,\|\, h_{update})$
  **else**
    $\forall\langle idx', h\rangle \in mPCR$ :
      **if** $idx' = idx$ **then** $h \leftarrow \mathsf{H}(h \,\|\, h_{update})$
**end if**
$h_{\mathsf{pol}} \leftarrow 0 \ldots 0$
$\forall\langle\mathcal{H}, h, \mathsf{name}(\mathcal{H})\rangle \in mNVPCR$ :
  $args \leftarrow \mathsf{H}(h \,\|\, 0x0000 \,\|\, 0x0000)$
  $h_{pol} \leftarrow \mathsf{H}(h_{pol} \,\|\, CC_{PolicyNV} \,\|\, args \,\|\, \mathsf{name}(\mathcal{H}))$
**if** $mPCR \neq \emptyset$ **then**
  $h_{PCR} \leftarrow \emptyset, indices \leftarrow \emptyset$
  $\forall\langle idx', h\rangle \in mPCR$ :
    $h_{PCR} \leftarrow h_{PCR} \,\|\, h$
    $indices \leftarrow indices \cup idx'$
    $h_{pol} \leftarrow \mathsf{H}(h_{pol} \| CC_{PolicyPCR} \| indices \| \mathsf{H}(h_{PCR}))$
  $\mathsf{Sig}^{EK_{\mathsf{sk}}}_{\mathsf{H}(h_{pol})} \leftarrow tpm.\mathsf{Sign}(\mathsf{H}(h_{pol}), \mathcal{H}_{EK})$
  **return** $h_{pol}, \mathsf{H}(h_{pol}), \mathsf{Sig}^{EK_{\mathsf{sk}}}_{\mathsf{H}(h_{pol})}, idx, \mathcal{B}_{NV}$
**end if**

To prove to SCB that the correct PCR ($idx$) was extended, $\mathcal{X}$ starts an HMAC session and runs the extend command in audit mode to have the TPM internally witness (see Algorithm 6) the incoming CPs and outgoing Response Parameters (RP) into the session's audit digest ($cpHash$, $rpHash, auditDigest$ are described in Part 1 of the TPM 2.0 specifications). Once the command completes, $\mathcal{X}$ asks the TPM to certify the current session's audit digest with $\mathcal{X}$'s EK and sends it to SCB. To verify the audit digest (Algorithm 5), SCB first computes the expected audit digest, with the correct arguments and a successful Response Code (RC). If $\mathcal{X}$'s audit digest differs from the expected, or the signature is incorrect, $\mathcal{X}$ did poorly.

#### 5.3.2.4 Proof of Conformance

Equipped with an authorized policy, $\mathcal{X}$ can serve attestation requests. When another D, $\mathcal{Y}$, wants to determine whether $\mathcal{X}$ is correct, $\mathcal{Y}$ sends $\mathcal{X}$ a nonce $n$. If $\mathcal{X}$ responds with a signature over $n$

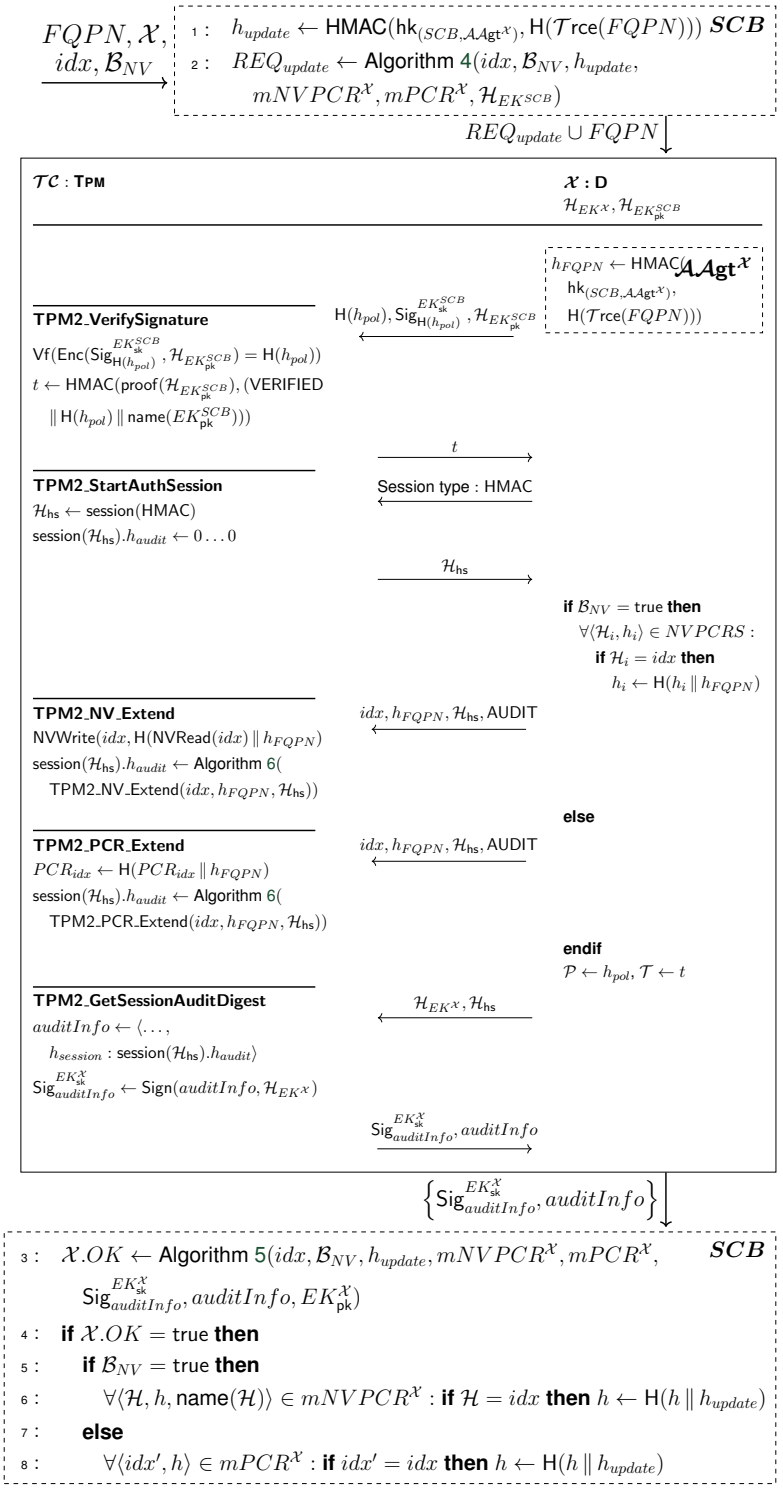



Figure 5.8: Measurement update

using its certified AK, then $\mathcal{Y}$ knows that $\mathcal{X}$ fulfills SCB's requirements. The sequence of steps performed by $\mathcal{X}$ are shown in Figure 5.9, where $\mathcal{X}$ first executes a series of policy commands (i.e., `PolicyPCR` and `PolicyNV`) to verify and measure the currently active PCRs (Section 5.3.2.2) in a session's policy digest. Once all PCRs have been accounted for, $\mathcal{X}$ runs `PolicyAuthorize` with the verified ticket (Section 5.3.2.3) and authorized policy (denoted $\mathcal{P}$). If the session's policy digest corresponds to the approved policy, then the TPM replaces the session's policy digest with the name (digest over the public area) of SCB's EK, which allows $\mathcal{X}$ to wield its AK and sign $\mathcal{Y}$'s

**Algorithm 5** Verify session audit digest

**Input:** $idx, \mathcal{B}_{NV}, h_{update}, mNVPCR, mPCR, \mathsf{Sig}^{EK_{\mathsf{sk}}}_{auditInfo}, auditInfo, EK_{\mathsf{pk}}$
**Output:** $\mathcal{B}$
  **if** $\mathcal{B}_{NV} = \mathsf{true}$ **then**
    $\forall \langle \mathcal{H}, h, \mathsf{name}(\mathcal{H}) \rangle \in mNVPCR$ :
      **if** $\mathcal{H} = idx$ **then**
        $cpHash \leftarrow \mathsf{H}(CC_{NV\_Extend} \,\|\, \mathsf{name}(\mathcal{H}) \,\|\, \mathsf{name}(\mathcal{H}) \,\|\, \mathsf{len}(h_{update}) \,\|\, h_{update})$
        $rpHash \leftarrow \mathsf{H}(\mathsf{success} \,\|\, CC_{NV\_Extend})$
  **else**
    $\forall \langle idx', h \rangle \in mPCR$ :
      **if** $idx' = idx$ **then**
        $cpHash \leftarrow \mathsf{H}(CC_{PCR\_Extend} \,\|\, idx' \,\|\, idx' \,\|\, authHash \,\|\, h_{update})$
        $rpHash \leftarrow \mathsf{H}(\mathsf{success} \,\|\, CC_{PCR\_Extend})$
**end if**
$h_{audit} \leftarrow \mathsf{H}(0 \ldots 0 \,\|\, cpHash \,\|\, rpHash)$
$\mathsf{Vf}(h_{audit} = auditInfo.h_{session})$
$\mathsf{Vf}(\mathsf{Enc}(\mathsf{Sig}^{EK_{\mathsf{sk}}}_{auditInfo}, EK_{\mathsf{pk}}) = auditInfo)$
**return** true

**Algorithm 6** Witness

**Input:** $\mathsf{CMD} : \mathcal{H}_0 \times \mathcal{H}_1 \times \mathcal{H}_2 \times params \rightarrow RC \times CC \times rparams$
**Output:** $h'_{audit}$ - updated audit session digest
  $cpHash \leftarrow \mathsf{H}(CC(\mathsf{CMD}) \,\|\, \mathsf{name}(\mathcal{H}_0) \,\|\, \mathsf{name}(\mathcal{H}_1) \,\|\, \mathsf{name}(\mathcal{H}_2) \,\|\, params)$
  $rpHash \leftarrow \mathsf{H}(RC(\mathsf{Eval}(\mathsf{CMD})) \,\|\, CC_{\mathsf{CMD}} \,\|\, rparams)$
  $h'_{audit} \leftarrow \mathsf{H}(h_{audit} \,\|\, cpHash \,\|\, rpHash)$
  **return** $h'_{audit}$

challenge.

## 5.4 Swarm Attestation

This section aims to provide a high-level description of the swarm attestation features in the ASSURED framework. ASSURED aims to provide various swarm attestation approaches based on the context in which the attestation is required. More details on the exact workflow of actions will be provided in D3.6 [29].

### 5.4.1 Basic Swarm Attestation Scheme in ASSURED

The basic swarm attestation scheme in ASSURED aims to provide attestation of a group of devices relying on some fundamental assumptions regarding the network properties and the attestation objectives that should be fulfilled. In particular, the basic swarm attestation considers *static attestation* of devices participating in *static network topology*, e.g., static attestation of devices deployed in smart manufacture. This network is structured as a hierarchical fog computing model where the devices have parent-child relationship, as shown in Figure 3.6. The attestation starts at the level of IoT devices which perform attestation and report the result to edge devices. In particular, each device may act both as Prover and Verifier to support the aggregation of the

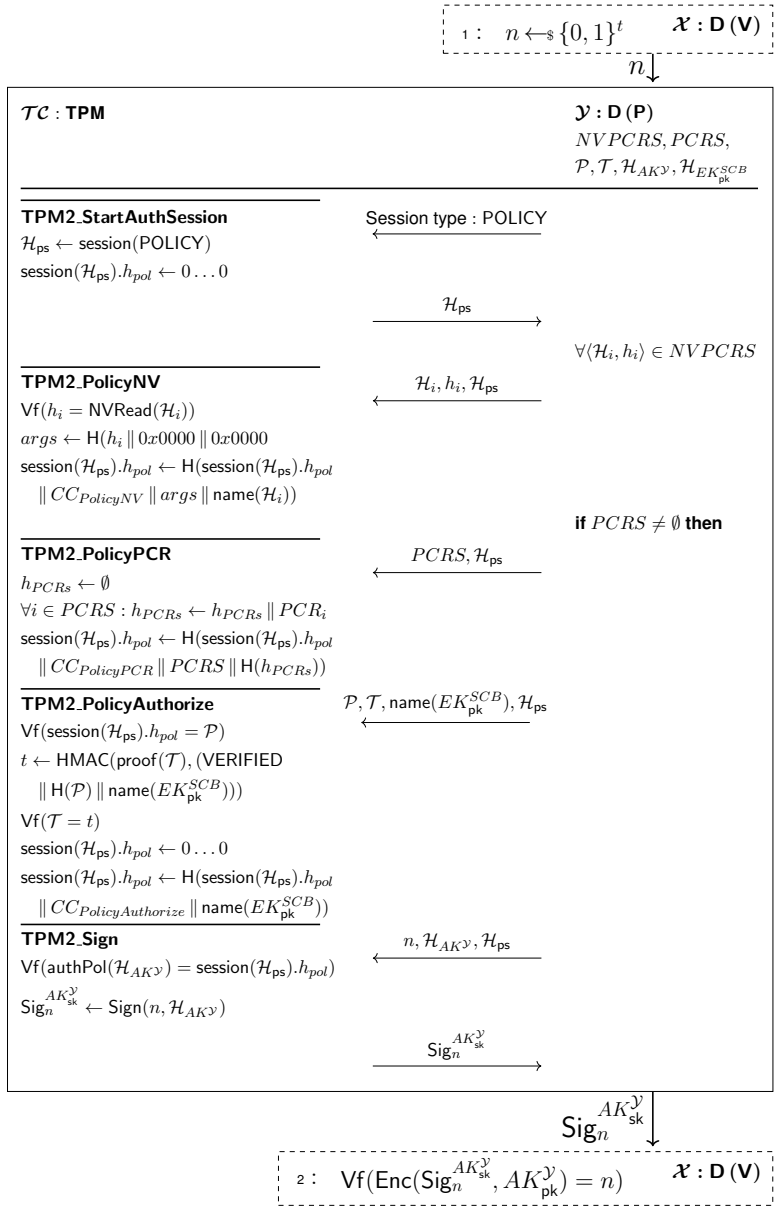



Figure 5.9: Oblivious Remote Attestation (ORA)

attestation result across the network. For instance, edge devices can attest each other, and the aggregated results are reported to the central ASSURED components. In addition, the basic swarm attestation scheme relies on a *centralized Verifier* that validates the *overall* state of the network. Such attestation outputs a Boolean result (e.g., 1 if all devices are healthy and 0 otherwise) without precisely identifying compromised devices by id. While the basic swarm attestation scheme does not provide privacy-preserving guarantees for the swarm attestation and does not consider the communication data exchanged among devices in the network, in the following, we discuss the main directions for enhancing swarm attestation in ASSURED based on the required properties in a given context.

### 5.4.2 Privacy-Preserving Swarm Attestation

In the context of various scenarios, it might be crucial for the swarm attestation in the ASSURED framework to guarantee privacy-preserving property. To achieve this, the swarm attestation will

rely on ring signatures that allow remote attestation of a device associated with a Trusted Component (TC) while offering strong anonymity guarantees. Specifically, DAA allows any verifier to check that attestations originate from a certified hardware token without learning anything about the identity of the TPM. To provide privacy-preserving guarantees of a swarm attestation scheme, ASSURED will rely on the adoption of relevant cryptographic tools e.g., *ring signatures*. Ring signature is a group-oriented signature in which the signer can spontaneously form a group and generate a signature such that the verifier is convinced the signature was generated by one member of the group and yet does not know who actually signed. Thus, a message signed with a ring signature is endorsed by member in a particular set of members. One of the security properties of a ring signature is that it should be computationally infeasible to determine which of the set's members' keys was used to produce the signature. In this way, the ring signature in ASSURED enables a verifier to verify the *overall* state of the network in a privacy-preserving manner.

#### 5.4.2.1 Precise Network State in a Privacy-Preserving Swarm Attestation

In various scenarios, the *overall* trustworthy state of the network might not be enough, especially when the network is reported to be compromised. Thus, it might be important for the verifier to identify precisely the compromised devices in the network. To enable this functionality, ASSURED may rely on *traceable (or linkable) ring signature* schemes. Generally, traceable ring signatures are associated with high overhead. To deal with such performance challenge, ASSURED will focus on making a trade-off between the security, privacy and computational overhead. For instance, when there is a low risk indicator, the default swarm attestation scheme can be a privacy-preserving scheme which checks the overall network state. When there is a high risk indicator or the overall network state is reported as compromised, ASSURED can run another swarm attestation approach that allows the identification of the compromised devices in a large network.

### 5.4.3 Dynamic Swarm Attestation

The aforementioned swarm attestation approaches perform static attestation in individual devices in the network. However, such schemes do not detect runtime attacks that corrupt control-flow pointers. In this direction, we have conducted the preliminary research [49] in the context of ASSURED project. We have proposed ARCADIS as a novel attestation protocol that focuses on performing control-flow attestation of an event-driven IoT system where events are unpredictable, asynchronous, and the physical clocks of the devices are not synchronized. The objective of ARCADIS is to detect not only the malicious IoT devices compromised by runtime attacks, but also other devices which are maliciously influenced due to their direct or indirect interactions with the infected device. This approach will be further extended to consider the presence of multiple verifiers in the attestation.

## 5.5 Jury-based Attestation

To elaborate on the details of jury-based attestation, this section discusses the three main components it requires. Figure 5.10 depicts how these components interact. To outline, in step 1 a single node attests another node, e.g., because it observed suspicious behavior. If the attestation failed, the node will broadcast its suspicion to the network, which we coin *blaming*. Such a blame

triggers step 2, i.e., the distributed election on all individual nodes to elect a set number of Jurors. Once this representative Jury has been elected, they start the consensus protocol among them in step 3. As a part of this agreement each Juror will independently attest the blamed node (step 4) and finally reach a consensus about the attestation result. Finally, in step 5, the Jury will broadcast its decision among the network. The following describes the three involved components in more detail.

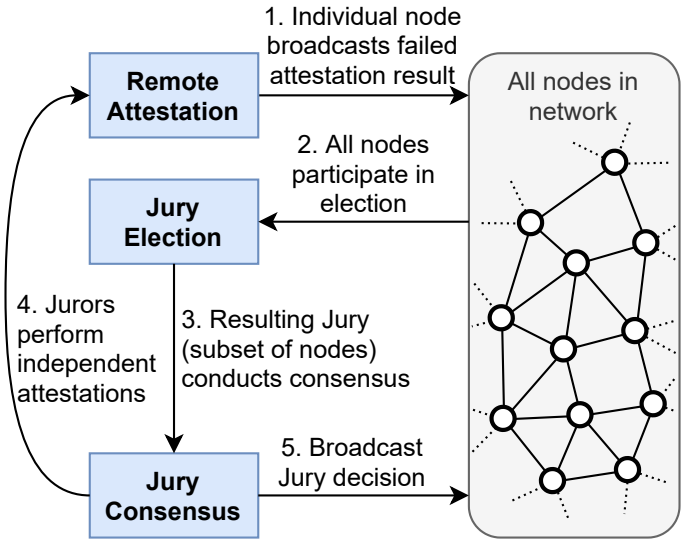



Figure 5.10: Overview of Jury-based Attestation regarding the Sequence of Actions by the individual Components

One is the remote attestation scheme. Here, any scheme that allows devices to directly attest each other is sufficient, i.e., a scheme that does not require a central authority and is lightweight enough to run on the network's devices. To elect the Jury, we require a distributed election scheme. To outline, these schemes work with publicly verifiable randomness, such that nodes can "draw" a number, compare each other's numbers, and finally let the lowest numbers win the election. Specifically useful in this regard is Intel's Proof-of-Elapsed-Time [57]. While designed to work with Intel's SGX, the approach can be adopted to other TEE platforms as well, such as ARM TrustZone. It was designed as alternative to the expensive Proof-of-Work election scheme used in popular blockchains. It works by first generating a random number based on a node's public identity. As a simplified example, one could hash a node's public key together with the current round number and only take a couple of bits of this hash as the election number; thus, this number is verifiable by the entire network. This number is then used to wait for the random amount of time inside a TEE. Once a node is finished waiting, the TEE will create an attestation that attests that the node executed this waiting code correctly, which is called a waiting certificate. Each node executes these steps for the election and will broadcast its individual waiting certificate. After some time, all nodes will receive and confirm all other nodes waiting certificates and converge on a common list of jurors. Nevertheless, the waiting approach also increases the efficiency, as nodes that waited longer, may have already seen lower waiting times when they are done waiting and can decide that their chances of winning are very low. Thus, this leads to a dramatic decrease of broadcasted waiting certificates across the network, lowering the communication overhead of the election.

After the Jury is elected, it individually needs to attest the suspicious device and find an agree-

ment. This is done with the final component of jury-based attestation via a Byzantine Fault Tolerance consensus protocol, such as PBFT [17]. To outline, these protocols guarantee *safety*, i.e., consistent decisions, and *liveness*, i.e., eventually reaching decisions, among $3F + 1$ jurors, with $F$ being the amount of malicious jurors that can be tolerated. However, traditionally these protocols are executed among the entire network. Yet, as we randomly elect a subset of nodes as jurors the security guarantees change to probabilities. Briefly, we need to particularly consider two critical cases. One is the probability to elect more than two-thirds of malicious devices as the Jury, as they can enforce malicious decisions for the network. Another is the case of electing more than a third but less than two-thirds, as this stalls the consensus, i.e., results in a liveness failure. However, as we elect a subset of nodes, we can trigger a re-election in the latter case, until a proper Jury is elected that can proceed.The probabilities of these events can be adjusted by two parameters. One is the size of the Jury. The bigger the Jury the less likely it is for the failure cases to happen. Another is increasing the required safety quorum, e.g., instead of requiring two-thirds to agree, we can raise this to four-fifth and increase the chances of requiring re-elections. Let's look at a concrete example. Suppose we aim for a chance of less than 1-in-a-billion to elect a malicious Jury. With a four-fifth safety quorum and a Jury size of $52$, we can tolerate up to $19\%$ of adversarial devices in the network, requiring $\sim 1.3$ elections on average. Therefore, the jury-based attestation approach can be adjusted via key parameters to fit the targeted use case.

# Chapter 6

# Revocation

## 6.1 Revocation in ASSURED

As described in Section 2.1, the endmost goal of the overall ASSURED attestation toolkit, is to enable the creation of **trust- and privacy-aware service graph chains** managed through security (attestation) policies that are enforced through policy-compliant Blockchain infrastructures. The former (*trustworthiness*) property is achieved through the provision of verifiable evidence that a device (or swarm of devices) works correctly both after loading (**loading-time integrity** achieved through Configuration Integrity Verification) and during run-time (**run-time integrity** achieved through Control-flow Attestation) referring to the whole process lifecycle of a device. For the latter, besides operational assurance, one has to cater for a number of properties like **anonymity, pseudonymity, unlinkability and unobservability**; especially, in the context of safety-critical applications (crf. "*Secure Collaboration of Platforms-of-Platforms* for Enhanced Public Safety" [24]) where critical decisions are based on information collected by distributed devices that at the same time have strict privacy requirements. For instance, separate data bundles - produced by the same device - **need to be sent in an authenticated but unlinkable manner**: *no data operations can be linked back to the origin device ID, hence, the use of a different pseudonym per data bundle (unconditional anonymity)*.

Addressing this challenge, current approaches are based on PKI-based solutions with privacy-friendly authentication services through the use of short-term anonymous credentials, i.e., *pseudonyms*. The common denominator in such architectures is the existence of trusted (centralized) infrastructure entities for the support of services such as authenticated platform registration, pseudonym provision, revocation and TPM-based Wallet. The location and identity privacy is protected by requiring that each device uses multiple pseudonyms, frequently changing from one pseudonym to another.

While such solutions have been well studied in the literature, there is a consensus on their limitations and drawbacks mainly due to the fact that they are based on a complex and centralized ecosystem of PKI entities that one needs to trust for issuing and distributing pseudonym certificates [45]. First, a technical and organizational separation of capabilities between the PKI entities is required to cope with internal attackers, resulting in a very costly solution to implement in practice. The bottleneck of having to connect to the back-end infrastructure to acquire pseudonym certificates is resolved by downloading a larger pseudonym pool size, which then provides less protection against Sybil attacks.

In ASSURED, as described in Section 3.3, we move towards a decentralized approach where **trust is shifted from the back-end infrastructure to the Edge**. The way we do this is by lever-

aging the use of Direct Anonymous Attestation (DAA) and the incorporation of trusted computing technologies. DAA is a cryptographic protocol designed primarily to enhance device/user privacy within the remote attestation process of computing platforms, which has been adopted by the Trusted Computing Group (TPM-based Wallet G), in its latest specification. It is based on group signatures that give strong anonymity guarantees to a device or user that wants to share operational data within an ecosystem. Consider, for instance, the following example in the context of the envisioned *Public Safety Scenario*: A user by leveraging his/her mobile device can send immediate feedback about an incident that takes place in his/her vicinity; e.g., maybe an ongoing fire or even theft can be reported (via the ASSURED Blockchain infrastructure) to the cloud-based backend processing system. However, in order to motivate users to participate in such crowd-sensing systems, it is imperative to protect their privacy. **Data should be authenticated in terms of their origin - originate from a valid and enrolled user (whose device can provide verifiable device on its correct operation) - but should not be linked back to his/her ID and/or location**. But this openness is a double-edge sword: any of the participants can be adversarial and pollute the collected data, seeking to manipulate (or even dictate) the system output. Faulty, distorted information can lead to wrong decisions, possibly rendering such public-safety systems useless.

Therefore, what is needed, is the design of an appropriate **revocation scheme** that can revoke the participation of a device within the system by de-activating any **credentials** and **short-term signature keys** created during their enrollment with the Privacy CA [23]: essentially, revoking the use of any *Attestation Key (AK)* and/or *pseudonyms*, created during the $DAA$ JOIN phase (Section 3.3.3)

Revocation is a standard consideration for any ICT system and supply chain. In case of a misbehaving platform, the wrongdoer can be evicted and be prevented from further participation. In the case of PKI-based solutions, the revocation can be done in standardized ways by adding the revoked certificates to a CRL, which is then published by the CA responsible for that trust domain. However, for devices using short-lived pseudonym certificates, things are more complicated. If a device possesses multiple certificates that are unlinkable, every single certificate needs to be put on the CRL, which would increase the bandwidth requirement to unfeasible levels. Nowatkowski et al. [67] have shown that the CRL list may grow as much as 2.2 GB, depending on the policy for the number of pseudonyms on devices.

In the remainder of this chapter, we provide a detailed documentation of the **decentralized revocation protocol**, that has been designed in the context of ASSURED, for revoking the short-term anonymous credentials (i.e., *pseudonyms*) generated by a device when employing the DAA protocol during their enrollment phase. Basically, a device can create a number of pseudonyms under the ECC-based DAA Key that can be used for later signing any operational or attestation-related data prior to sharing them via the Blockchain infrastructure. If a message $M$, signed under pseudonym $ps_i$, is deemed as malicious then all of the device's credentials must be de-activated through the device's TPM-based Wallet. Details on the **basic building blocks, mode of operation and workflow of actions** are provided as well as a **comprehensive mapping of the TPM commands** that need to be securely executed by the underlying TPM-based Wallet.

## 6.2 Design of Revocation Scheme

Unlinked, anonymous pseudonyms are hard to revoke in a privacy-preserving manner since, in this context, **privacy and revocation are contradicting requirements**: *How to be able to de-activate a misbehaving platform's credentials without disclosing its identity?* In this section, we

discuss how this paradox can be resolved by leveraging the capabilities of the underlying trusted component (ASSURED TPM-based Wallet), what it requires and the challenges that follow it. We have to highlight that throughout our description, we make the following assumptions: (*i*) There is already in place a misbehavior detection mechanism equipped with the necessary rules and policies for detecting any malevolent actions from insider attackers. ASSURED focuses on how to revoke the attacker's credentials once a misbehavior has been detected, (*ii*) A platform has already successfully engaged the DAA protocol with the Privacy CA for creating a a number of pseudonyms. While there are different variants of such short-term anonymous credentials that can be used for enhanced privacy, in ASSURED we confine ourselves to the scenario where devices are leveraging DAA-enabled credentials. However, this can be easily enhanced to also consider other types of privacy-preserving crypto primitives (check the full published version of the revocation scheme [64] for more details).

We start by introducing the Revocation Authority (RA) that shuns out misbehaving platforms from the system within the revocation domain that it's managing. This entity is important, as all pseudonyms are to be registered with this. It should not be possible for a platform to successfully use a pseudonym unless it has registered it with the RA. In the context of ASSURED, the role of the RA is undertaken by the Security Context Broker (SCB) [22]: Once the device is successfully enrolled with the Privacy CA where it certifies the created Attestation Key (AK) and the pseudonyms, it is then directed to the SCB for activating the pseudonyms protected under this specific AK. The registration consists of providing the SCB with unique values that can be used later to revoke the key. We call these values *revocation hashes*. Upon such a registration, the platform will receive Proof of Registration (PoR), which is provided with any signature from the particular pseudonym. Without PoR, any recipient should disregard the message, as the SCB could not revoke such a key. In case a recipient suspects malicious behavior of a platform, it reports the corresponding pseudonym to the SCB.

The SCB does not perform any **pseudonym resolution** to discover the identity of the misbehaving platform. Instead, it starts the revocation protocol by creating a signed revocation message using its secret key and broadcasts this to all platforms, containing the public pseudonym key that needs to be revoked. All platforms receive the revocation message, and forwards them to their corresponding TPM-based Wallet.

We differentiate between two kinds of revocations: **soft- and hard-revocation**. Soft revocation means the SCB revokes a specific pseudonym that was used for signing the message based on which the misbehavior policy violation was detected, while hard revocation means revoking all pseudonyms associated with a specific platform, thus, not allowing it to further take part in the overall system as an authenticated participant (basically, de-activating all pseudonyms created under a specific DAA AK).

### 6.2.1 General Concept

In this section, we make a high-level overview of our protocol, showcasing how we can implement **policy regulations** for governing the pseudonyms using the functionality of the TPM-based Wallet. More specifically, we present how we can leverage an internal, tamper-proof register of the TPM Wallet, where each bit represents the state of a pseudonym. This is essentially the instantiation of the **policy-based key usage** property defined in D3.1! [21].

We refer to the tamper-proof index of the TPM as *Revocation Index*. It has 64 bits, and each bit represents the state of a pseudonym, i.e., a set bit means revoked; otherwise, the pseudonym can be used for signing. We consider two cases: Revocation of a single pseudonym, namely soft

revocation, is rather straightforward. As we trust the TPM-based Wallet managing the keys, it will be asked to set the key's respective revocation bit to a "revoked state". Hard revocation refers to the possibility of being able to use one of the 64 bits to revoke all pseudonyms in a single round. This means that the DAA AK should be linked to the first bit of the Revocation Index, while the pseudonyms are linked to one of the other 63 bits, as well as the first (hard revocation) bit.

We must accommodate the possibility that different authorities govern different areas of an IoT ecosystem/network, i.e., an SCB in one domain should not be allowed to revoke pseudonyms linked to another domain. Therefore, we must protect each bit in the revocation index, allowing only predetermined SCBs to execute a revocation process. This can be achieved by building policies for each pseudonym, representing the command being executed (set bit) with particular parameters (which bit). A particular SCB must sign these to authorize a revocation. We refer to them as *revocation hashes* and each pseudonym is linked to the following: one for soft revocation and one for hard revocation. These revocation hashes are registered with the SCB, who must sign them before using them in the revocation process.

It should now be clear that the hard revocation hash must represent both setting the hard-revocation bit and the pseudonyms' unique soft-revocation bit. If this is not the case, the hard-revocation hash for all linked pseudonyms would be equal, as they are to be signed by the same SCB and setting the same bit. By including the soft-revocation bit, we "blind" the revocation hash.

### 6.2.2 Protocol Limitations

Since the size of the Revocation Index is 64 bits and given we need one for the hard revocation, we can revoke only 63 pseudonyms with this index, which in the long run is not enough.

A naive approach to support more pseudonyms would be to create more revocation indexes and having a new DAA AK for each one, following the same principle. However, this poses two distinct problems: first, the DAA $SETUP$ and JOIN phases are very time consuming and require communication with the Privacy CA, acting as the Issuer. Secondly, only 63 pseudonyms can be linked in the TPM-based Wallet, making hard revocation of a larger set impossible. Therefore, we create multiple revocation indexes using all of their bits for soft revocation and we maintain only one hard revocation bit to be the one defined in the initial revocation index. So all pseudonyms share the same hard revocation bit, meaning hard revocation hashes would be equal for all pseudonyms having soft revocation bits in other indexes.

An example of this is shown in Figure 6.1, where all pseudonyms linked to the DAA AK share a common hard revocation bit (first bit of $r_1$), while their corresponding soft revocation bits span several revocation indexes.

With the current implementation of the TPM, it is not possible to set multiple bits in different indexes, which means they would have to be executed as two commands, removing the "blinding" of the hard revocation hash.

To combat this challenge, we propose that the hard-revocation hash represents a command that sets the hard revocation bit and any other unique combination of bits in the index, allowing for $2^{63}$ pseudonyms with a unique hard-revocation index. This requires that the revocation index's bits are revocable only by a single RA; otherwise, an anonymizing mask could cause the unintentional revocation of keys linked to another revocation domain.

It should now be clear that we protect the revocation index with a set of policies. This, however, raises an interesting challenge: *When we write a policy that determines what parameters must*

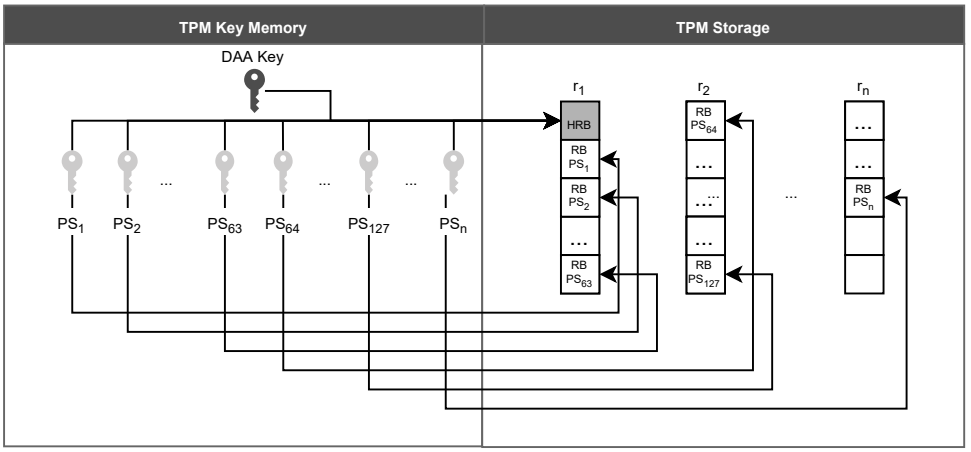



Figure 6.1: Pseudonyms in TPM linked to different indexes

*be used, a so-called command parameter hash must include the name*[1] *of the entity it applies to - in this case, the revocation index*. As the policy is intended *for* the index, it is included in its public part. The index's name depends on the policy, and the policy depends on the index: *we, therefore, have an infinite hash loop*.

To avoid this hash loop, we use a different approach[2] where a unique key, referred to as *Authorization Key* (AK), authorizes the policy (Figure 6.2). The process is that *any* policy signed by the AK is considered a valid policy. The revocation index's actual policy digest is the name of the authorizing key, and in order to satisfy that policy, one must have the policy signed by the AK. The index and revocation policies, therefore, are no longer coupled together. However, this approach raises another obstacle: *we must guarantee that the AK can only sign a single policy*. If not, the host can sign any policy to comply with and control the revocation index.

To address this challenge, we protect the AK by yet another policy. This policy must dictate how many times the host can use the key. We do this by creating an additional index, called the *Authorization Counter Index* (ACI), and protecting it by a PIN or password as an authorization value. The ACI contains two parts: Authorization Counter and Authorization Limit. We use this index to create the authorization key policy by leveraging `PolicySecret` functionality. This policy states that we must prove our knowledge of the ACI's secret by authenticating with it. Every time the password is used for this index, the internal counter increments. When the counter reaches the pre-determined limit, it fails. With this index, we can define the AK policy as the correct authentication with the ACI, meaning that we must perform (password) "proof-of-knowledge" for that index, therefore, incrementing the counter and limiting the use of the key to the authorization limit of $3+1$, where the first three authorization processes are for: i) writing the authorization limit, ii) authorizing the final policy digest, and iii) activating the revocation index. The last authorization is required for the AK to be used once as the Ephemeral Key in order to make the next ACI immutable. Now the host can still re-create the ACI and, therefore, reset the counter and infer additional authorizations. To overcome this limitation, we protect the update of the ACI by setting a policy that requires a signature from an Ephemeral Key - that is, a key that only exists during the initial setup, therefore, making the ACI immutable. A straightforward way is to create the ephemeral key in TPM-based Wallet's NULL hierarchy. In this way, the host cannot recreate the key since the NULL hierarchy seed is reset on reboot.

Looking over this set of required tasks, it becomes cumbersome that the reboot is necessary

[1] Name is a hash of the public parameters of an entity, including its policy.
[2] This was identified as a solution in collaboration with the TPM WG of TCG.

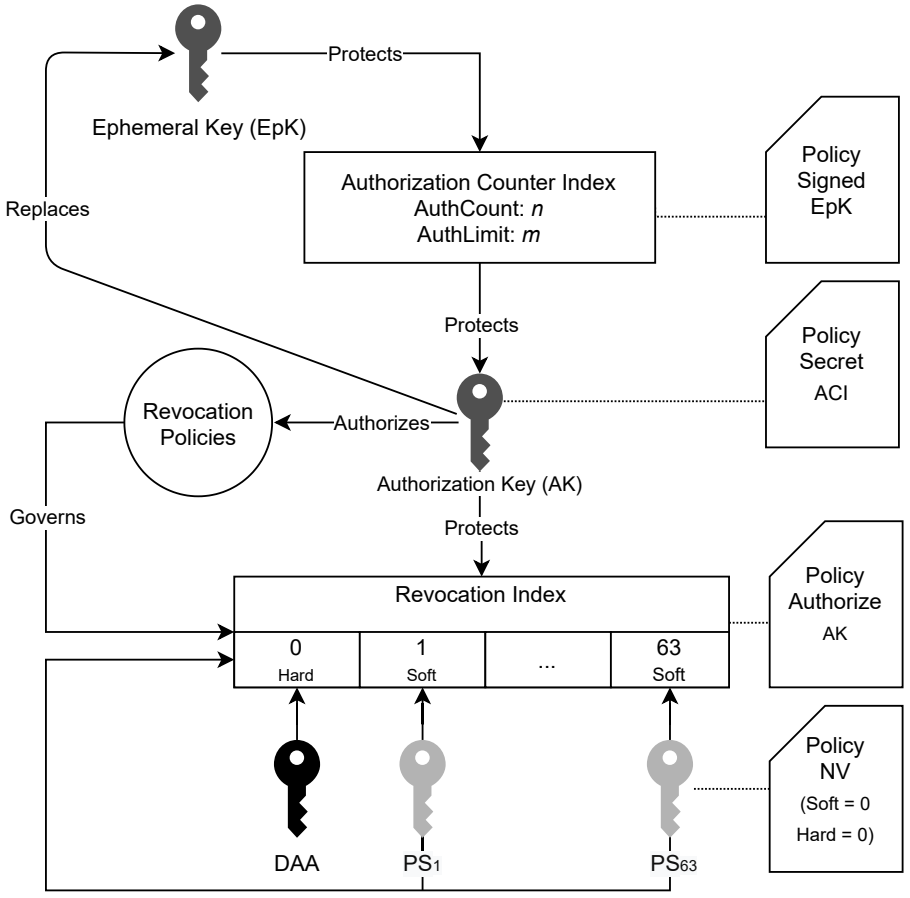



Figure 6.2: Protocol Functionality and Lifecycle

for each revocation index created. It is neither efficient nor safe to reboot a device after 64 pseudonyms have been used. One solution is to give the Authorization Key one additional authorization and use this as the ephemeral key for the next ACI. This has the same effect as a reboot and will render the ACI immutable after the initial write.

Following this approach, we can successfully initialize both the ACI and the revocation index. Before the host can start using the keys, it must initiate the revocation index by updating it; otherwise, it is unusable. In order to guarantee that this initial write-operation can only happen once, we will give the AK an additional authorization and sign a digest allowing the initial write.

### 6.2.3 Technical Setup

Recall that we have four distinct phases. First, we create the **primordial authorization counter index (ACI)**, then the **revocation index and its policies**, and last, we **activate the revocation index**. If intended, the platform can create more iterations of these phases, with slight change to the ACI phase.

#### 6.2.3.1 Authorization Counter Index

The initial phase of creating the Primordial Authorization Counter Index can be seen in Figure 6.3. As can be seen, the host must provide an index identifier, template, and authorization limit. As a potentially untrusted host can process such information, this underlines why a trusted entity

should verify the index in order to further weaken the trust assumptions regarding the trustworthiness of the host (Section 7.3). The first thing to be executed is the creation of the Ephemeral Key in the NULL hierarchy. This can be verified by a certification process that documents the key and TPM-based Wallet's validity, and the current boot count.

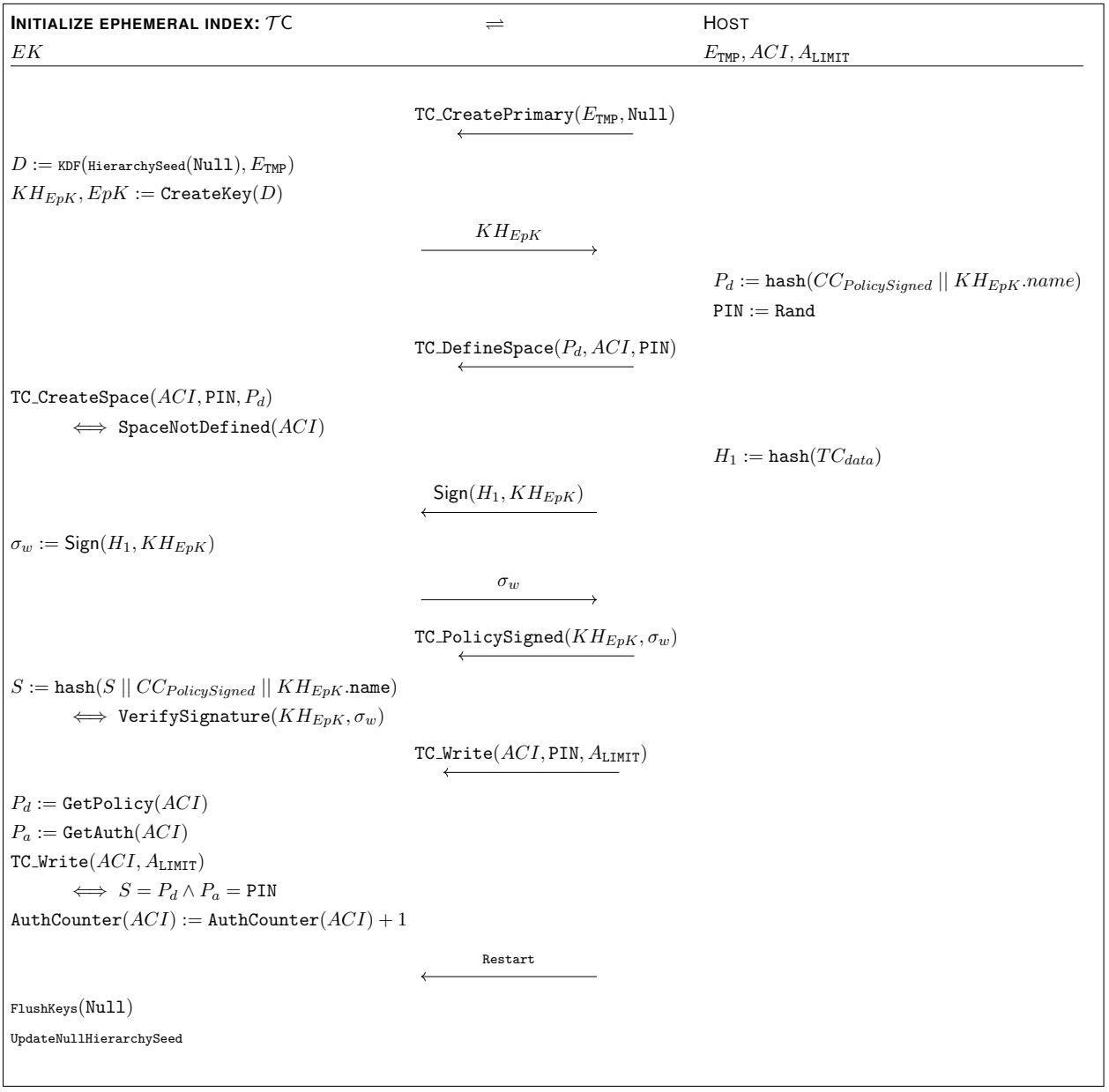



Figure 6.3: Initialize Primordial Authorization Counter Index

To create the index, the host creates a policy based on the Ephemeral Key and instructs the TPM-based Wallet to define the space with a random secret. The secret is essentially a password; however, the endmost goal is to use it as a usage counter. Hence, there is no need for secrecy.

The policy defined earlier must be complied with to write the authorization limit to the index, so the host uses the Ephemeral Key to sign a nonce. After complying with the policy, writing the authorization count, and inherently incrementing the authorization count, the key should be rendered inoperable by executing a reset. After the reset has been completed, the index can be certified by the privacy CA together when activating/certifying the endorsement key. Such a

certificate can prove that the index has a specific authorization limit, and the current boot count proves the reset has been executed. After the reboot, the ACI is guaranteed immutable and is prepared to act as a guard for limiting the number of times the upcoming authorization key can be used.

#### 6.2.3.2 Revocation Index

Before initializing the revocation index, we need to create the AK and link it to the ACI. As depicted in the first line of Figure 6.4, the host builds a `PolicySecret` policy based on the ACI name: In order for the host to use the key, it needs to provide the secret for the ACI, thus, resulting in the increment of the authorization counter. This policy is embedded in the template for the key, and in the future, only this template with this specific policy will allow the correct AK recreation.

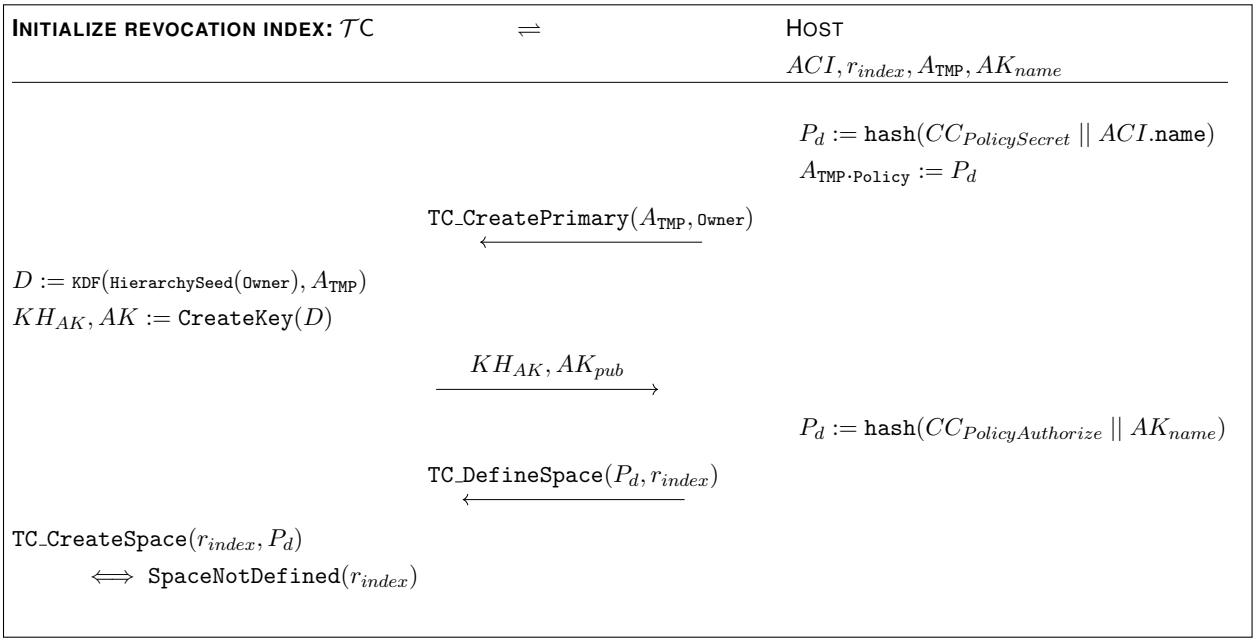



Figure 6.4: Initializing Revocation Index

As with other entities, the created key can be signed and added as the public part of the Endorsement Key for later verification. To create the index, the host calculates a new policy digest that links the index's usage to the AK by leveraging the `PolicyAuthorize` functionality - meaning that any policy signed by the authorization key is valid. The index is now built within the trusted component, and its policy depends on what the authorization key signs in the next phase.

#### 6.2.3.3 Generate Policy Digest

The policy digest to be authorized by the AK is calculated following the steps described in Algorithm 7. Recall that the policy is a compound policy built by logical `AND` and `OR` statements. We can visualize this logical composition, as depicted in Figure 6.5, where we can see that two policies must be true in order to produce $l_1$ or $l_2$.

The first would be setting the soft revocation bit while the latter for updating the hard revocation bits. Either of these will satisfy the `OR` operation, producing $b_1$, which in turn is an input to a final `OR` operation. More specifically, if the RA provided an authentic signature with a parameter either a hard- or soft-revocation hash (that is unique to a key), this is a valid branch for a single

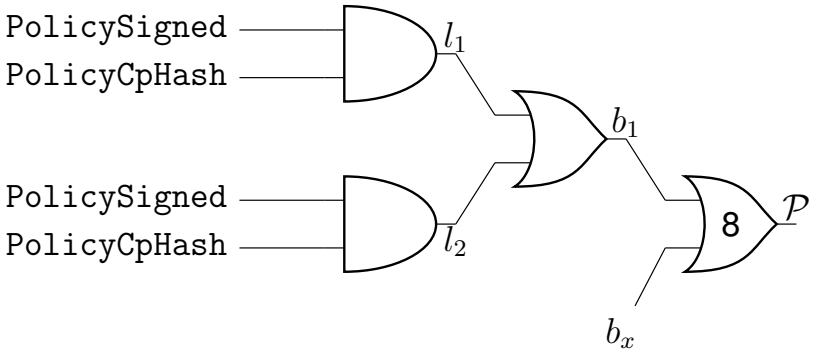



Figure 6.5: Structure of the policy to be authorized

pseudonym, and its revocation will take place. Each branch digest $b$ represents a valid soft- or hard revocation for a single pseudonym. An `OR` operation may take up to 8 input parameters; hence, it might be necessary to have multiple layers of these operators.

It is possible to calculate these by executing the commands in a trial session of the TPM-based Wallet, but we showcase this by calculating it on the host. From an implementation standpoint, we start by initializing our variables and continue to define our very first branch digest: the activation. Recall that we have to update the index before it can be used. To ensure this can only be performed once, we leverage the AK's use-limit property and allow an initial write to the index. We then continue into a loop where we iterate over all revocable pseudonyms: For each of the keys, we create two leaf digests, $l_1$ and $l_2$. We also initiate $S_{data}$ and $H_{data}$, representing the parameters used in the `SetBits` command: the bits being set. We increment the `anonymizer` and set the $H_{data}$, thereby ensuring the uniqueness of the parameter hash. After having anonymized the data, we can set the respective hard- and soft revocation bits and continue calculating the respective revocation hashes. We see $k.sri.name$, the pseudonym $k$'s soft revocation index's name, and $hri$ representing the hard revocation index. With this data, we can calculate the soft- and hard revocation leaf digests, $l_1$, $l_2$, and then calculate the branch digest $b$. This is inserted into $\beta$, the list of all branch digests, and then we loop again. When all $b$'s are inserted into $\beta$, we can calculate the final digest to be authorized during RA activation.

#### 6.2.3.4 Activating the Revocation Index

Before we can use the RI, we must update it, which in our case is setting it to zero. This policy is depicted in Figure 8.1 in Annex 8.2. Recall that the policy is built to allow a write to the index if the AK provides a signature over the command to execute. The policy is not authorized yet, so the first task that the host needs to perform is to prepare the values to be hashed and signed: hashing the final policy and generating the command parameter hash for the initial write. It then continues to gain access to the AK by providing the ACI's secret incrementing the AC.

Now the host can get a signature over the policy and acquire a verification ticket: A ticket guarantees that it is this particular TPM-based Wallet that has verified the signature, therefore, guaranteeing the command and its parameters have been correctly authorized.

To gain (writing) authorization privileges, the host initiates a new session and executes one of the index's valid policies, namely the `PolicySigned` with the previously acquired signature over the zero-write command parameter hash. The current session digest should now match the branch digest $b_0$, and the host executes `PolicyOR` with $\beta$. After a successful verification from the TPM-based Wallet, it will replace the session digest with a concatenation of all provided branch digests in $\beta$, which should be the index's authorized policy. The host executes `PolicyAuthorize` with

**Algorithm 7** Calculate Final Policy Digest

1. Initialize $\mathcal{P}$ as an array of hashes (capable of holding $n$ hashes where $n$ is the number of pseudonyms) and `anonymizer` as clear byte. Calculate activation branch digest $b_0 := H(H(b_0 \mid\mid CC_{PolicySigned} \mid\mid AK_{name}))$ and add to $\beta$
2. For $k \in \mathcal{K}$:
   (a) Initialize $l_1, l_2$ as a two digest buffers. Initialize $S_{data}$ and $H_{data}$ as 8 byte buffers and set all bytes to zero.
   (b) Set bit identified by $k.s_{bit}$ high in byte number 8 in $S_{data}$
   (c) Increment `anonymizer` and set $H_{data}$ to be the binary representation if it. Set bit identified by $k.h_{bit}$ high in byte number 8 in $H_{data}$
   (d) Compute soft- and hard revocation hashes
   $H_s := H(CC_{NV\_SetBits} \mid\mid k.sri.name \mid\mid k.sri.name \mid\mid s_{data})$
   $H_h := H(CC_{NV\_SetBits} \mid\mid k.hri.name \mid\mid k.hri.name \mid\mid h_{data})$
   (e) Compute soft revocation leaf digest as
   $l_1 := H(H(l_1 \mid\mid CC_{PolicySigned} \mid\mid k.RA.name))$
   $l_1 := H(l_1 \mid\mid CC_{PolicyCpHash} \mid\mid H_s)$
   (f) Compute hard revocation leaf digest as
   $l_2 := H(H(l_2 \mid\mid CC_{PolicySigned} \mid\mid k.RA.name))$
   $l_2 := H(l_2 \mid\mid CC_{PolicyCpHash} \mid\mid H_h)$
   (g) Add branch digest $b := H(CC_{PolicyOR} \mid\mid l_1 \mid\mid l_2)$ to $\beta$
3. Compute `finalPolicy` $:= H(CC_{PolicyOr} \mid\mid \beta_1 \mid\mid \beta_2 ... \mid\mid \beta_n)$
4. **Output** `finalPolicy`

the previously acquired ticket and signature. The TPM-based Wallet then verifies the ticket, the signature, and finally, the session digest (whether it matches the authorized digest). If this is the case, the TPM-based Wallet replaces the session digest with the AK's name: the policy digest for the index. The host can now execute a write operation, and the index has been activated.

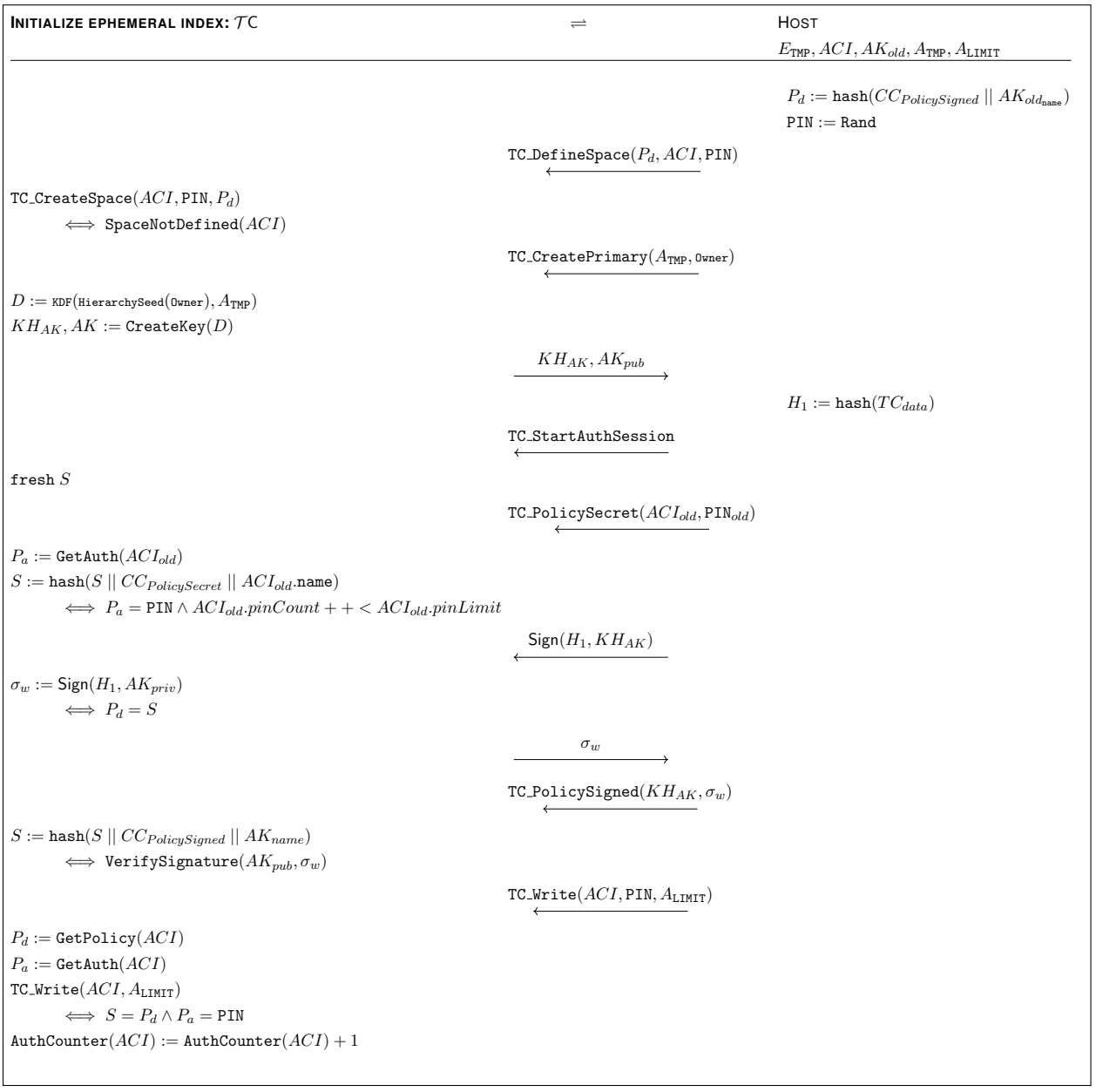
While it is now possible for the host to update the index, a possibly compromised host does not have any incentive to write anything but zeroes, as it would set pseudonyms in an initially revoked state. At the end of this phase, the index has been correctly activated and set. Furthermore, the final policy calculated in the previous step has also been authorized. It is now ready to be used for managing the revocation states of pseudonyms. As with the primordial ACI, the index has been initialized with an authorization limit and counter and is immutable. The index can also act as a protector for regulating a new AK.

#### 6.2.3.5 Initialize New Authorization Counter Index

To initiate a new ACI, we must guarantee immutability by using an old AK, as it will only have a single authorization left, based on the previously defined ACI, see Figure 6.6 Thus, we start by creating a policy digest based on `PolicySigned` and the name of the old AK. We then define a

bit-space with that policy. Since we have a single authorization left in our AK, we can now create this new key. Recall that in order to use this key, we must execute `PolicySecret` and provide the PIN to the previous ACI. Thus, we need to provide a signature, based on the old key, and execute the `PolicySigned` command.

Figure 6.6: Initialize New Authorization Counter Index

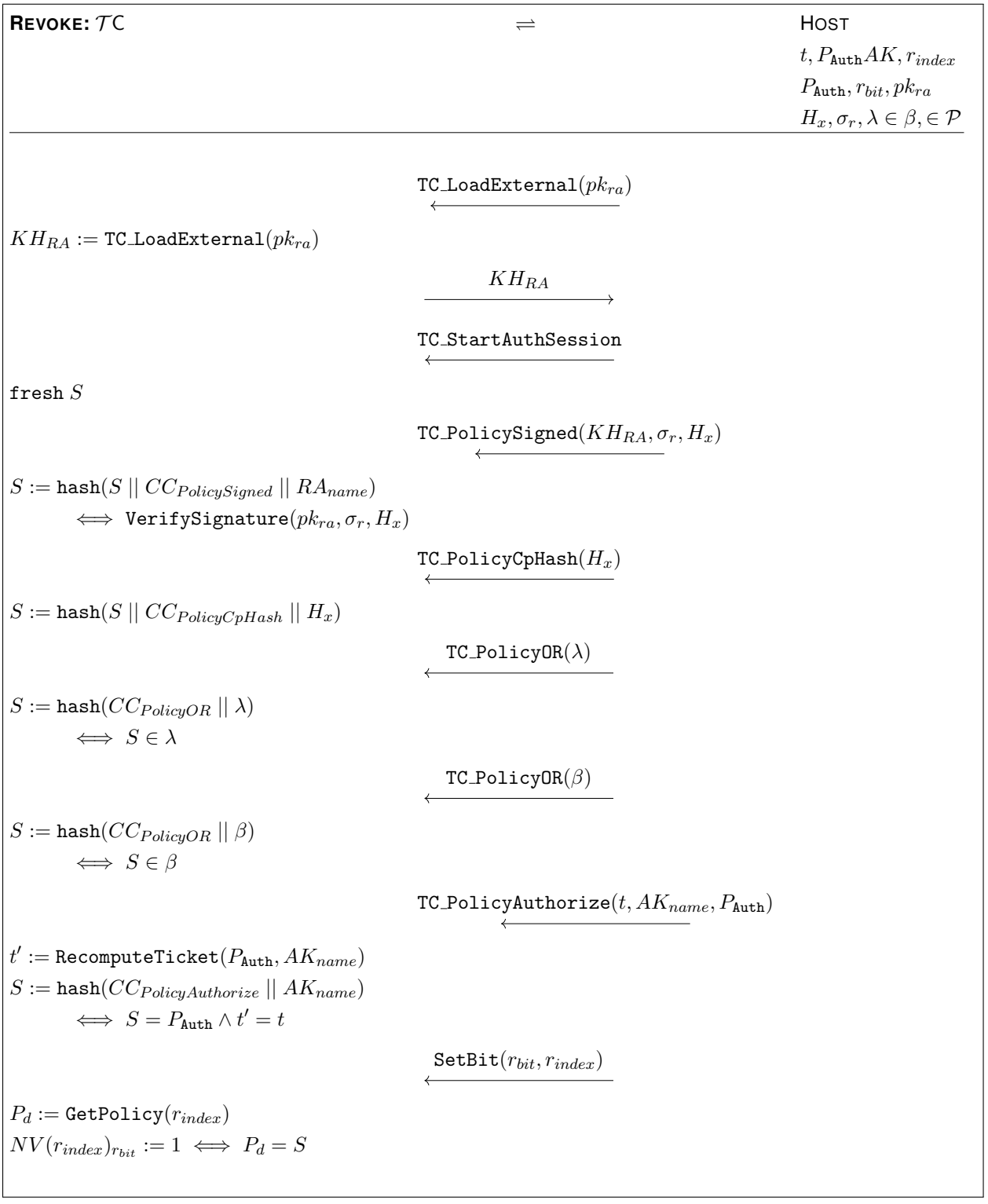



#### 6.2.3.6 Towards Near Zero-Trust Assumptions

Assuming near-zero trust assumptions for the platform requires additional validation of several processes that are executed in the host to protect against compromised hosts that try to manipulate the parameters given to the attached trusted component. This essentially considers a Dolev-Yao adversarial model, which allows an adversary to monitor and modify all interactions between the host and the TPM-based Wallet.

The critical operation to verify is the management of the policies generated on the host and that

Figure 6.7: Revoking Platforms Pseudonyms & DAA Key Pairs




these have been correctly calculated for protecting the appropriate indexes and keys linked to active pseudonyms (an adversary can create a policy for inactive pseudonyms in which case a revocation message will be received and handled correctly but without any actual revocation results). The second operation to protect is the content of the ACI in order to validate the authorization limit. Finally, pseudonyms should be verified to be governed by the correct RAs whose revocation hashes have been calculated correctly. These proofs can be done using the TPMs cer-

tification functionality and validated by a trusted entity, such as the Issuer. However, to acquire a high level of trust, the trusted party should not immediately release the correct pseudonym certificates. Instead, it should verify that the AK has been used to replace an ephemeral key and use its last authorization to write to the next-round ACI. Once finalized, the host cannot misuse the final authorization, and the system can be trusted entirely.

### 6.2.4 Revocation Flow

Upon receiving a (potential) revocation message, the host loads the SCB's public key into the TPM-based Wallet. The received message contains the public pseudonym key, $pk_{ps}$, which needs to be revoked, and a revocation hash and signature; i.e., of the respective revocation index, $r_{index}$ and (hard or soft) revocation bit, $r_{bit}$. See Figure 6.7. Recall that the policy for gaining write-access to the revocation index includes both `PolicySigned` (verifying that the message originates from the correct RA responsible for the trust domain where the platform also belongs to) and `PolicyCpHash` (correct revocation bit(s)): both these policies must be correctly satisfied before the TPM-based Wallet allows the successful revocation. Executing these processes should produce a valid leaf digest (the result of the hashing done in the TPM-based Wallet after `PolicyCpHash` is executed, also noted as $l_x$), validated by executing `PolicyOR` with a reference list of the possible leaf digest ($\lambda$) for that specific branch. Suppose the leaf digest matches a digest in the reference list. In that case, the session digest is replaced by a hash of the reference list: the branch digest, verified by an additional `PolicyOR`. We depict the execution of these policy commands in Figure 6.7 where it is highlighted that `PolicyOR` will replace the current session digest with a hashed concatenation of all provided reference branch digests ($\beta$) *if and only if* the current session digest is in the provided reference. If the reference list is unaltered, the hashed concatenation should be the authorized policy, which is verified by executing `PolicyAuthorize` with an authorization ticket, the name of the authorizing key, and, of course, the authorized policy. If the session digest matches the approved policy, then it is replaced by the hash of the command code of `PolicyAuthorize` and the authorizing key's name, which is the policy for the revocation index. The host can now execute a write to the index, but only with the parameters used in the `PolicyCpHash`, ensuring the correct bits are set; hence, *the right pseudonym(s) are revoked*.

Once all required pseudonyms, and their DAA key pairs, are deleted, the TPM-based Wallet responds to the platform with a signed revocation confirmation $\sigma_{rvk}$ which is then sent to the RA. Upon reception of the revocation confirmation, the RA verifies that this is signed by the same TPM-based Wallet that issued the pseudonym certificate that was revoked, thus, implying that the correct platform has revoked itself. The entire signature can be verified using the $DAA$ VERIFY. By the end of this protocol, there are strong guarantees that the platform in question has been revoked without the need for any pseudonym resolution. The SCB has verifiable evidence from the platform that it has performed the revocation enforced by the TPM-based Wallet.

We have to note here that in case an attacker intercepts this revocation message, or a malicious platform host blocks the revocation message intended for the TPM-based Wallet (Section 7.3), then the revocation process will not be triggered. However, this can be alleviated with the introduction of *heartbeat messages*.

# Chapter 7

# Security Analysis

After having defined the set of **run-time behavioral attestation services**, leveraged in ASSURED towards the creation of **privacy- and trust-aware service graph chains**, we now proceed with a formal analysis of their offered security and assurance guarantees based on the trust model and properties defined in D3.1 [21]. This constitutes the first step towards the definition of formal models capable for capturing the security properties for the **execution assurance of a single system** (remote attestation), and how these can be transferred to statements on the **security properties of compositions of systems** ("*Systems-of-Systems*" through swarm attestation) but also the **communication assurance between interacting systems**.

In the following sections, as a first step, we analyze the security properties of the complete landscape of ASSURED attestation mechanisms that address the need for verifiable evidence about a system and the integrity of its trusted computing base and provide formal statements about foundational concepts such as *measured boot, platform authentication, zero-touch configuration integrity verification, revocation, etc.*. We consider the analysis against the generic security properties defined in in D3.1 [21], including **confidentiality, authentication, integrity and availability, without though differentiating the internal cryptographc mechanisms of the underlying security token (ASSURED TPM-based Wallet) from the overall system cryptography**. This approach limits, to some extent, the reasoning that can be made about the level of assurance of the overall system by automated reasoning tools: *Because such roots-of-trust by definition are trusted, all internal operations including handling of cryptographic data can be idealized. However, this does not directly translate to the overall application security that leverages such RoTs.*

Compounding this issue, we have defined a methodology that enables the use of formal verification tools towards verifying complex protocols using trusted computing [42]. The focus is on reasoning about the overall application security, provided from the integration of the root-of-trust services, such as the ones offered by ASSURED, and how these can translate to larger systems when the underlying cryptographic engine is considered perfectly secure. Using the Tamarin prover, we have demonstrated the feasibility of our approach by instantiating it for the TPM-based simple remote attestation service, and in the next version of this deliverable we will leverage it for the formal verification of the more complex ASSURED attestation building blocks. We, essentially, idealized the internal functionalities of the TPM security token, except those that provide explicit cryptographic functionalities for the service being offered; i.e., **use of key hierarchies, authenticated sessions with the TPM and the Platform Configuration Registers (PCRs) for checking the current state of the host device**. For the details behind this abstract model, the reader is referred to [42]; as aforementioned, below we have put forth the more generic security analysis of our schemes based on the properties defined in D3.1 [21] against the adversarial model documented in D2.1 [28].

Recall that the goal of an adversary is to compromise and affect one or more ASSURED services maliciously while evading detection from the device acting as the Verifier. **The main objective is to prove that it is computationally infeasible for an adversary to forge the attestation result and manipulate the Verifier**. ASSURED enables the use of both *static* and *dynamic* attestation. In performing attestation, ASSURED aims to enable the remote attestation while preserving the user's privacy. To achieve this, ASSURED relies on using a Direct Anonymous Attestation (DAA) scheme (Section 5.2) to convince the Verifier that the signer possesses a valid membership credential without revealing the membership credential and the signer's identity. To formally analyze DAA scheme in ASSURED, we first describe the security model for DAA in the Universal Composability (UC) model comprising the expected security properties such as *unforgeability, anonymity, and non-frameability*. Then, we formally analyze these properties and present the security proof by using a sequence of games.

For the rest of the proposed attestation schemes, namely, Revocation scheme (Section 6), Configuration Integrity Verification (Section 5.3), Control-Flow Attestation (Section 5.1), and Jury-based Attestation (Section 5.5), we first highlight and describe the security properties and then provide the formal security analysis on how ASSURED fulfills each property. The details behind the Swarm Attestation protocol will be provided in the next version of this deliverable once the scheme has been finalized in D3.6 [29].

## 7.1 Universal Composability (UC) Model for Direct Anonymous Attestation (DAA)

We describe the security model for DAA given by Camenish et al. in [15]. Recall that as described in Section 3.3, the intuition behind employing DAA in ASSURED is for enabling **privacy-preserving platform authentication**. When a device wants to join the network for then been able to exchange either *operational and/or threat intelligence data* in a privacy-preserving manner (achieving properties including **anonymity, unlinkability, untraceability**), this can be done by leveraging short-term anonymous credentials - such as *pseudonyms* - that are created under one master ECC-based DAA Key. Data bundles then shared with other devices either directly or through the ASSURED Blockchain infrastructure, can be (anonymously) signed under these short-term credentials. One of the core innovations of DAA is that the level of privacy is controlled by the device itself through configuring the type of signatures leveraged. **This enables to better shift trust from the infrastructure to the edge**.

In this context, the security definition is given in the Universal Composability (UC) model with respect to the ideal functionalities $F_{daa}^{l}$ that operates in the trusted world as defined in [15]. In UC model, the ASSURED environment $\varepsilon$ should not be able to distinguish (with a non-negligible probability) between two worlds:

1. The real untrusted world where the ASSURED devices operate and execute the DAA protocol $\Pi$. We assume that the ASSURED network is controlled by an adversary $\mathcal{A}$ that communicates with $\varepsilon$.

2. The trusted and secure ideal world, in which all parties forward their inputs to a trusted third party such as the ASSURED Blockchain Peer Nodes (that validate the execution of the attestation process prior to recording it on the ledger), which is the ideal functionality that internally performs all the required tasks and creates the party's outputs.

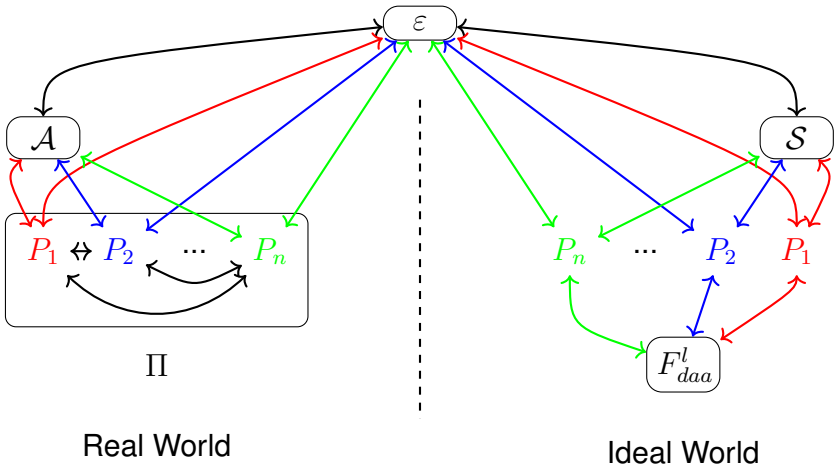



Figure 7.1: Universal composability security model: the real and the ideal world executions are indistinguishable to the environment $\varepsilon$.

A protocol $\Pi$ is said to securely realize the ideal functionality if for every adversary $\mathcal{A}$ performing an attack in the real ASSURED world, there is an ideal world adversary $\mathcal{S}$ that performs the same attack in the ideal world. More precisely, given a protocol $\Pi$, an ideal functionality $F^l_{daa}$ and an environment $\varepsilon$, we say that $\Pi$ securely realises $F^l_{daa}$ if the real world in which $\Pi$ is used is as secure as the ideal world in which $F^l_{daa}$ is used as shown in Figure 7.1 from [39].

### 7.1.1 Security Properties of ASSURED DAA

In general the security properties as defined in [15] for a DAA scheme adopted by ASSURED are the following: *Where the Privacy CA acts as the Issuer for veriyfing the validity of the TPM-based wallet of each device been enrolled in the network [22].*

**ASSURED DAA Unforgeability** This property requires that the Issuer is honest and should hold even if the host/device is corrupt. If all the TPMs are honest, then no adversary can output a signature on a message M with respect to a basename (bsn). On the other hand, if not all the TPMs are honest, say $n$ TPMs are corrupt, the adversary can at most output $n$ unlinkable signatures with respect to the same basename.

**ASSURED DAA Unforgeability Experiment**:
**Exp**$^{\text{ASSURED DAA Unforge}}$
$\mathcal{A} \leftarrow (bsn, M)$
returns $\sigma$ such that Verify($\sigma$)=1
Probability $|$**Exp**$^{\text{ASSURED DAA Unforge}} = 1|$ is negligible.

**ASSURED DAA Anonymity**: This property requires that the entire platform $(\mathsf{tpm_i} + \mathsf{host_j})$ is honest and should hold even if the issuer is corrupt. Starting from two valid signatures with respect to two different basenames, the adversary can't tell whether these signatures were produced by one or two different honest platforms.

**ASSURED DAA Anonymity Experiment**:
**Exp**$^{\text{ASSURED DAA anon, } b}$
For any tpm $\mathsf{tpm_i} \leftarrow (\mathsf{tsk_i}, \mathsf{bsn}, \sigma)$
For any tpm $\mathsf{tpm_k} \leftarrow (\mathsf{tsk_k}, \mathsf{bsn}', \sigma')$ with $bsn \neq bsn'$
$\mathcal{A} \leftarrow (\sigma, \sigma', bsn, bsn')$
returns $b = 1$ if $i = k$ or $b = 0$ if $i \neq k$
Probability $|\textbf{Exp}^{\text{ASSURED DAA anon, } b=1}|$ - Probability $|\textbf{Exp}^{\text{ASSURED DAA anon, } b=0}|$ is negligible.

**ASSURED DAA Non-frameability**: This requires that the entire platform $(\mathsf{tpm_i} + \mathsf{host_j})$ is honest and should hold even if the issuer is corrupt. It ensures that no adversary can produce a signature that links to signatures generated by an honest platform.

**ASSURED DAA Non-frameability Experiment**:
**Exp**$^{\text{ASSURED DAA Non-frameability}}$
$\mathcal{A} \leftarrow (bsn, M)$
returns $\sigma$ such that $\sigma$ that links to an honest paltform with TPM $\mathsf{tpm_i}$
Probability $|\textbf{Exp}^{\text{ASSURED DAA Non-frameability}}| = 1$ is negligible.

#### 7.1.1.1 The Ideal Functionality $F_{daa}^{l}$ in ASSURED [15]

Recall that as described in Section 3.3, DAA is based on group signatures that give strong anonymity guarantees to a device or user that wants to share operational data within an ecosystem. Consider, for instance, the following example in the context of the envisioned *Public Safety Scenario*: A user by leveraging his/her mobile device can send immediate feedback about an incident that takes place in his/her vicinity; e.g., maybe an ongoing fire or even theft can be reported (via the ASSURED Blockchain infrastructure) to the cloud-based backend processing system. However, in order to motivate users to participate in such crowd-sensing systems, it is imperative to protect their privacy. **Data should be authenticated in terms of their origin - originate from a valid and enrolled user (whose device can provide verifiable device on its correct operation) - but should not be linked back to his/her ID and/or location**. But this openness is a double-edge sword: any of the participants can be adversarial and pollute the collected data, seeking to manipulate (or even dictate) the system output. Faulty, distorted information can lead to wrong decisions, possibly rendering such public-safety systems useless.

Towards this direction, DAA is a **platform authentication mechanism that enables the provision of privacy-preserving and accountable authentication services**. By leveraging group-based signatures, any verifying entity can verify a platform's credentials in a privacy-preserving manner using the previously described DAA algorithms; without the need of knowing the platform's identity. The Elliptic-curve cryptography (ECC) based DAA is comprised of five algorithms SETUP, JOIN, SIGN, VERIFY and LINK.

In what follows, we explain the interfaces of the ideal DAA functionality (as described in Section 3.3), $F_{daa}^{l}$, in the context of ASSURED as follows:

**SETUP**: On the input(SETUP, sid) from the issuer $I$, $F_{daa}^{l}$ does the following:

- Verify that $(I,\ \text{sid}') = \text{sid}$ and output (SETUP, sid) to $\mathcal{S}$.

- SET Algorithms. Upon receiving the algorithms (Kgen, sig, ver, link, identify) from the simulator $\mathcal{S}$, it checks that (ver, link, identify) are deterministic and outputs (SETUPDONE, sid) to $I$.

**JOIN**:

1. JOIN REQUEST: On input (JOIN, sid, jsid, $\mathsf{tpm_i}$) from the host $\mathsf{host_j}$ to join the TPM $\mathsf{tpm_i}$, the ideal functionality $F_{daa}^{l}$ proceeds as follows:
   - Create a join session $\langle$jsid, $\mathsf{tpm_i}$, $\mathsf{host_j}$, request$\rangle$. Output (JOINSTART, sid, jsid, $\mathsf{tpm_i}$, $host_j$) to $\mathcal{S}$.
2. JOIN REQUEST DELIVERY: Proceed upon receiving delivery notification from $\mathcal{S}$, update the session record to $\langle$jsid, $\mathsf{tpm_i}$, $\mathsf{host_j}$, delivery$\rangle$.
   - If $I$ or $\mathsf{tpm_i}$ is honest and $\langle \mathsf{tpm_i}, \star, \star \rangle$ is already in Members, output $\perp$.
   - Output (JOINPROCEED, sid, jsid, $\mathsf{tpm_i}$) to $I$.
3. JOIN PROCEED:
   - Upon receiving an approval from $I$, $F_{daa}^{l}$ updates the session record to $\langle$jsid, sid, $\mathsf{tpm_i}$, $host_j$, complete$\rangle$. Then output (JOINCOMPLETE, sid, jsid) to $\mathcal{S}$.
4. KEY GENERATION: On input (JOINCOMPLETE, sid, jsid, $gsk$) from $\mathcal{S}$.
   - If both $\mathsf{tpm_i}$ and $\mathsf{host_j}$ are honest, set $gsk = \perp$, else verify that the provided $gsk$ is eligible by performing the following two checks:
     - If $\mathsf{host_j}$ is corrupt and $\mathsf{tpm_i}$ is honest, then CheckGskHonest($gsk$)=1.
     - If $\mathsf{tpm_i}$ is corrupt, then CheckGskCorrupt($gsk$)=1.
   - Insert $\langle \mathsf{tpm_i}, \mathsf{host_j}, gsk \rangle$ into Members, and output (JOINED, sid, jsid) to $\mathsf{host_j}$.

**SIGN**

1. SIGN REQUEST: On input (SIGN, sid, ssid, $\mathsf{tpm_i}$, $\mu$, bsn) from the host $\mathsf{host_j}$ requesting a DAA signature by a TPM $\mathsf{tpm_i}$ on a message $\mu$ with respect to a basename bsn, the ideal functionality does the following:
   - Abort if $I$ is honest and no entry $\langle \mathsf{tpm_i}, \mathsf{host_j}, \star \rangle$ exists in Members, else create a sign session $\langle$ssid, $\mathsf{tpm_i}$, $\mathsf{host_j}$, $\mu$, bsn, request$\rangle$. Output (SIGNSTART, sid, ssid, $\mathsf{tpm_i}$, $\mathsf{host_j}$, $l(\mu, \mathsf{bsn})$) to $\mathcal{S}$.
2. SIGN REQUEST DELIVERY: On input (SIGNSTART, sid, ssid) from $\mathcal{S}$, update the session to $\langle$ssid, $\mathsf{tpm_i}$, $\mathsf{host_j}$, $\mu$, bsn, delivered$\rangle$. $F_{daa}^{l}$ output (SIGNPROCEED, sid, ssid, $\mu$, bsn) to $\mathsf{tpm_i}$.
3. SIGN PROCEED: On input (SIGN PROCEED, sid, ssid) from $\mathsf{tpm_i}$
   - Update the records $\langle$ssid, $\mathsf{tpm_i}$, $\mathsf{host_j}$, $\mu$, bsn, delivered$\rangle$, and output (SIGNCOMPLETE, sid, ssid) to $\mathcal{S}$.
4. SIGNATURE GENERATION: On the input (SIGNCOMPLETE, sid, ssid, $\sigma$) from $\mathcal{S}$.

- If both $\mathsf{tpm_i}$ and $\mathsf{host_j}$ are honest, then ignore the adversary's signature and internally generate the signature for a fresh or established $gsk$.
  - If bsn $\neq \perp$, then retrieve $gsk$ from the $\langle \mathsf{tpm_i},\ \mathsf{bsn},\ gsk\rangle \in$ DomainKeys.
  - If bsn $= \perp$ or no $gsk$ was found, generate a fresh key $gsk \leftarrow Kgen(1^\lambda)$.
  - Check CheckGskHonest($gsk$)=1, and store $\langle \mathsf{tpm_i},\ \mathsf{bsn},\ gsk\rangle$ in DomainKeys.
  - Generate the signature $\sigma \leftarrow sig(gsk,\ \mu,\ \mathsf{bsn})$.
  - Check ver($\sigma,\ \mu,$ bsn)=1, and check identify($\sigma,\ \mu,$ bsn, $gsk$)=1.
  - Check that there is no TPM other than $\mathsf{tpm_i}$ with key $gsk'$ registered in Members or DomainKeys such that identify($\sigma,\ \mu,$ bsn, $gsk'$)=1.
- If $\mathsf{tpm_i}$ is honest, then store $\langle \sigma,\ \mu,\ \mathsf{tpm_i},\ \mathsf{bsn}\rangle$ in Signed and output (SIGNATURE, sid, ssid, $\sigma$) to $\mathsf{host_j}$.

**VERIFY**: On input (VERIFY, sid, $\mu$, bsn, $\sigma$, $RL$), from a party $V$ to check whether a given signature $\sigma$ is a valid signature on a message $\mu$ with respect to a basename bsn and the revocation list $RL$, the ideal functionality does the following:

- Extract all pairs $(gsk_i,\ \mathsf{tpm_i})$ from the DomainKeys and Members, for which identify($\sigma,\ \mu,$ bsn, $gsk$)=1. Set $b = 0$ if any of the following holds:
  - More than one key $gsk_i$ was found.
  - $I$ is honest and no pair $(gsk_i,\ \mathsf{tpm_i})$ was found.
  - An honest $\mathsf{tpm_i}$ was found, but no entry $\langle \star,\ \mu,\ \mathsf{tpm_i},\ \mathsf{bsn}\rangle$ was found in Signed.
  - There is a key $gsk' \in RL$, such that identify($\sigma,\ \mu,$ bsn, $gsk'$)=1 and no pair $(gsk,\ \mathsf{tpm_i})$ for an honest $\mathsf{tpm_i}$ was found.
- If $b \neq 0$, set $b \leftarrow$ver($\sigma,\ \mu,$ bsn). Add $\langle \sigma,\ \mu,\ \mathsf{bsn},\ RL,\ b\rangle$ to VerResults, and output (VERIFIED, sid, $b$) to $V$.

**LINK**: On input (LINK, sid, $\sigma_1,\ \mu_1,\ \sigma_2,\ \mu_2,$ bsn), with bsn $\neq \perp$, from a party $V$ to check if the two signatures stem from the same signer or not. The ideal functionality deals with the request as follows:

- If at least one of the signatures ($\sigma_1,\ \mu_1,$ bsn) or ($\sigma_2,\ \mu_2,$ bsn) is not valid (verified via the VERIFY interface with $RL \neq \emptyset$), output $\perp$.
- For each $gsk_i$ in Members and DomainKeys, compute $b_i \leftarrow$ identify($\sigma_1,\ \mu_1,$ bsn, $gsk_i$) and $b_i'$= identify($\sigma_2,\ \mu_2,$ bsn, $gsk_i$) then set:
  - $f \leftarrow 0$ if $b_i \neq b_i'$ for some $i$, or $f \leftarrow 1$ if $b_i = b_i' = 1$ for some $i$.
- If $f$ is not defined, set $f \leftarrow$link($\sigma_1,\ \mu_1,\ \sigma_2,\ \mu_2,$ bsn), then output (LINK, sid, $f$) to $V$.

### 7.1.2 DAA Instantiation in the UC Model

**Setup**: An issuer $I$ generates his key pairs $x, y \leftarrow \mathbb{Z}_q$, sets $X = g_2^x$ and $Y = g_2^y$. $I$ proves that his key is well formed and register his key $(X, Y, \pi_{ipk})$ at $F_{ca}$.

**Join**: $I$ upon receiving a join request session from $\mathsf{host_j}$, chooses a fresh nonce $n$ and sends it to $\mathsf{host_j}$ who forwards it to the TPM $\mathsf{tpm_i}$. The TPM then chooses his secret key $gsk \leftarrow \mathbb{Z}_q$,

sets $Q = g_1^{gsk}$, and computes $\pi_1$, a proof of construction of $Q$. $\mathsf{tpm_i}$ sends $(Q, \pi_1)$ to the issuer via $\mathsf{host_j}$. The issuer then verifies $\pi_1$ and that the platform is eligible to join, then generates the credential as follows: The issuer chooses $r \leftarrow \mathbb{Z}_q$, sets $a = g_1^r$, $b = a^y$, $c = a^x Q^{rxy}$ and $d = Q^{ry}$, $I$ also generates a proof $\pi_2$ on his credential constructions. $I$ sends $(a, b, c, d, \pi_2)$ to $\mathsf{host_j}$. The host verifies $\pi_2$, checks that $a \neq 1$, $e(a, Y) = e(b, g_2)$, $e(c, g_2) = e(ad, X)$, then forwards $(b, d, \pi_2)$ to $\mathsf{tpm_i}$. The TPM verifies $\pi_2$ and stores $(gsk, b, d)$ in its records.

**Sign**: To sign a message $\mu$ with respect to a basename $\mathsf{bsn}$, the host re-randomizes his credential by choosing a random $s \leftarrow \mathbb{Z}_q$, and sets $(a', b', c', d') \leftarrow (a^s, b^s, c^s, d^s)$. $\mathsf{host_j}$ then sends $(\mu, \mathsf{bsn}, s)$ to the TPM $\mathsf{tpm_i}$ who checks that $b' = b^s$ and $d' = d^s$, sets $\mathsf{nym} = H_1(\mathsf{bsn})^{gsk}$, and calculates a $SPK$ on $(\mu, \mathsf{bsn})$ as

$$SPK\{gsk : \mathsf{nym} = H_1(\mathsf{bsn})^{gsk}, d' = b'^{gsk}\}(\mathsf{bsn}, \mu)$$

The TPM sends $(\mathsf{nym}, \pi)$ to the host who ouputs the signature $\sigma \leftarrow (a', b', c', d', \mathsf{nym}, \pi)$.

**Verify**: The verifier upon receiving a signature $\sigma = (a', b', c', d', \mathsf{nym}, \pi)$ on a message $\mu$ w.r.t. a bsename $\mathsf{bsn}$, it verifies $\pi$ w.r.t. $(\mu, \mathsf{bsn})$ and $\mathsf{nym}$. Checks that $a' \neq 1$, $e(a', Y) = e(b', g_2)$, $e(c', g_2) = e(a'd', X)$. For all $gsk_i \in RL$, the verifier checks that $b'^{gsk_i} \neq d'$, it sets $f = 1$ if all the checks passed and $f = 0$ otherwise.

**Link**: Upon receiving $\sigma_1 = (a'_1, b'_1, c'_1, d'_1, \mathsf{nym}_1, \pi_1)$ and $\sigma_2 = (a'_2, b'_2, c'_2, d'_2, \mathsf{nym}_2, \pi_2)$ on a message $\mu$ w.r.t. a basename $\mathsf{bsn}$, if both signatures are valid and $\mathsf{nym}_1 = \mathsf{nym}_2$ output 1, otherwise 0.

## 7.2 Security Proof of the DAA in the UC Model

In this section, we provide a sketch of the security proof of how DAA provides the core properties of **user-controlled anonymity, user-controlled linkability, non-frameability, correctness**. During the proof, we present a sequence of games based on Camenish et al. in [15], and show that there exists no environment $\varepsilon$ that can distinguish the real world protocol $\Pi$ with an adversary $\mathcal{A}$, from the ideal world $F_{daa}^{I}$ with a simulator $\mathcal{S}$. Starting with the real world protocol game, we change the protocol (game-by-game) in a computationally indistinguishable way, finally ending with the ideal world protocol. We will explain the sequence of games as follows:

**Game 1**: This is the real world protocol.

**Game 2**: An entity $C$ is introduced, $C$ receives all inputs from the honest parties and simulates the real world protocol for them. This is equivalent to Game 1.

**Game 3**: We now split $C$ into two parts, $F$ and $\mathcal{S}$, where $F$ behaves as an ideal functionality, it receives all the inputs and forwards them to $\mathcal{S}$, who simulates the real world protocol for honest parties, and sends the outputs to $F$. $F$ then forwards the outputs to $\varepsilon$. This game is simply Game 2 but with different structure, so Game 3=Game 2.

**Game 4 (ASSURED DAA Setup)**: $F$ now behaves differently in the setup interface, it stores the algorithms for the issuer $I$, $F$ also does checks and ensures that the structure of $sid$ is correct for an honest $I$, and aborts if not. In case $I$ is corrupt, $\mathcal{S}$ extracts the secret key for $I$ and proceeds

in the setup interface on behalf of $I$. Clearly $\varepsilon$ will notice no change, so Game 3=Game 4.

**Game 5 (ASSURED DAA Verification)**: $F$ now performs the verification and linking checks instead of forwarding them to $\mathcal{S}$. There are no protocol messages and the outputs are exactly as in the real world protocol. However, the only difference is that the verification algorithm that $F$ uses doesn't contain a revocation check, so $F$ can perform this check separately so the outcomes are equal, Game 4=Game 5.

**Game 6 (ASSURED DAA Key Membership)**: The join interface of $F$ is now changed, $F$ stores in it's records the members that joined. If $I$ is honest, $F$ stores the secret key $tsk$, extracted from $\mathcal{S}$, for corrupt TPM's. $\mathcal{S}$ always has enough information to simulate the real world protocol except when the issuer is the only honest party. In this case, $\mathcal{S}$ doesn't know who initiated the join, so can't make a join query with $F$ on the host's behalf. Thus, to deal with this case, $F$ can safely choose any corrupt host and put it into Member List $ML$, the identities of hosts are only used to create signatures for platforms with an honest TPM or honest host, so fully corrupted platforms don't matter. In the only case, when the TPM is already registered in Members List $ML$, $F$ may abort the protocol, but $I$ has already tested this case before continuing with the query JOINPROCEED, hence $F$ will not abort. Thus in all cases, $F$ and $\mathcal{S}$ can interact to simulate the real world protocol, and Game 6=Game 5.

**Game 7 (ASSURED DAA Anonymity)** (Simulating TPM wihout knowing the secret): In this game, $F$ creates anonymous signatures for honest platforms by running the algorithms defined in the setup interface. Let us start by defining Game $7.k.k'$, in this game $F$ handles the first $k'$ signing inputs of $\mathsf{tpm_k}$, subsequent inputs are then forwarded to $\mathcal{S}$. For $i < k$, $F$ handles all the signing queries with $\mathsf{tpm}_i$ using algorithms. For $i > k$, $F$ forwards all signing queries with $\mathsf{tpm}_i$ to $\mathcal{S}$ who creates signatures as before. Now from the definition of Game $7.k.k'$, we note that Game 7.0.0=Game 6. For increasing $k'$, Game $7.k.k'$ will be at some stage equal to Game $7.k + 1.0$, this is because there can only be a polynomial number of signing queries to be processed. Therefore, for large enough $k$ and $k'$, $F$ handles all the signing queries of all TPM's, and Game 7 is indistinguishable from Game $7.k.k'$.

We want to prove now that Game $7.k.k' + 1$ is indistinguishable from Game $7.k.k'$. Suppose an environment can distinguish a signature by an honest party with the gsk it joined with from a signature by the same party but with a different gsk. Then we can use that environment to break a DDH instance $\alpha, \beta, \gamma$ by simulating the join and the first signature using the unknown $\log_{g_1}(\alpha)$ as gsk, and for the second signature we use the unknown $\log_\beta(\gamma)$ as gsk. In the reduction we have to be able to simulate the TPM without knowing gsk, but merely based on $\alpha = g_1^{gsk}$. A TPM uses gsk to set $Q \leftarrow g_1^{gsk}$ in the join protocol, to do proofs $\pi_1$ in joining and $\pi$ in signing, and to compute pseudonyms. In simulation, we set $Q \leftarrow \alpha$ and we simulate all proofs $\pi_1$ and $\pi$. For pseudonyms, the power over the random oracle is used: S chooses $H_1(bsn) = g_1^r$ for r $\leftarrow \mathbb{Z}_q$, and sets $nym \leftarrow \alpha^r = H_1(bsn)^{gsk}$ without knowing gsk.

**Game 8**: $F$ now no longer informs $\mathcal{S}$ about the message and the basename that are being signed. If the whole platform is honest, then $\mathcal{S}$ can learn nothing about the message $\mu$ and the basename $bsn$. Instead, $\mathcal{S}$ knows only the leakage $l(\mu, bsn)$. To simulate the real world, $\mathcal{S}$ chooses a pair $(\mu',\ bsn')$ such that $l(\mu', bsn')$=$l(\mu, bsn)$. Therefore Game 8=Game 7.

**Game 9 (ASSURED DAA Platform membership)**: If $I$ is honest, then $F$ now only allows the platform that joined to sign. An honest host will always check whether it joined with a TPM in the

real world protocol, so no difference for honest hosts. Also an honest TPM only signs when it has joined with the host before. In the case that an honest $\mathsf{tpm_i}$ performs a join protocol with a corrupt host $\mathsf{Host_j}$ and honest issuer, the simulator will make a join query with $F$, to ensure that $\mathsf{tpm_i}$ and $\mathsf{Host_j}$ are in $ML$. Therefore Game 9=Game 8.

**Game 10 (ASSURED DAA Traceability)**:When storing a new $tsk$, $F$ checks CheckTtdHonest($tsk$)=1 or CheckTtdCorrupt($tsk$)=1. If the TPM is corrupt, F checks that CheckGskCorrupt($gsk$) = 1 for the gsk that the simulator extracted from $\pi_1$. This check prevents the adversary from choosing different keys. There exists only a single $gsk$ for every valid signature where identify($\sigma, m, bsn, gsk$) = 1, and thus this check will never fail. For keys of honest TPMs, $F$ verifies that CheckGskHonest($gsk$) = 1 whenever it receives or generates a new $gsk$. With these checks we avoid the registration of keys for which matching signatures already exist. Since keys for honest TPMs are chosen uniformly at random from an exponentially large group and every signature has exactly one matching key, the chance that a signature under that key already exists is negligible. Hence Game 10=Game 9.

**Game 11 (ASSURED DAA Correctness)**: In this game $F$ checks that honestly generated signatures are always valid. This is true as sig algorithm always produces signatures passing through verification checks, also those signatures satisfy $identify(tsk, \sigma, \mu, bsn) = 1$ which is checked via $nym$. $F$ also makes sure, using it's internal records $ML$ and DomainKeys that honest users are not sharing the same secret key $tsk$. Hence Game 11=Game 10.

**Game 12 (ASSURED DAA Non-Frameability)**: Add check to ensure that there are no multiple $tsk$ values matching to one signature.$F$ also checks with the help of its internal key records Members and DomainKeys that no one else already has a key which would match this newly generated signature. If this fails, we can solve the DL problem: We simulate a TPM using the unknown discrete logarithm of the DL instance as gsk like in the DDH reduction before. If a matching gsk is found, then we have a solution to the DL problem. Thus Game 12=Game 11.

**Game 13 (ASSURED DAA Credential Validity)**: To prevent accepting signatures that were issued by use of join credentials not issued by honest issuer, $F$ adds a further check Check (x). This is due to the unforgeability of our signature scheme adopted by the issuer to sign the TPM's public key, the hardness is based on the Ring SIS Search Problem, so we get Game 13=Game 12.

**Game 14 (ASSURED DAA Unforgeability)**: Check (xi) is added to $F$, this would prevent anyone forging signatures using honest TPM's $tsk$ and credential. In fact, if a valid signature is given on a message, that the TPM never signed, the proof could not have been simulated.

**Game 15 (ASSURED DAA Revocation)**: Check (xii) is added to $F$, this ensures that honest TPMs are not being revoked. If an honest TPM is simulated, if a proper key $RL$ is found, it must be the secret key of the target instance. This is again equivalent to solving the discrete logarithm of the problem instance.

**Game 16 (ASSURED DAA Linkability)**: All the remaining checks of the ideal functionality $F^l_{DAA}$ that are related to link queries are now included. Using the fact that if a $tsk$ matches to one signature and not the other, Game 16 is indistinguishable from Game 15, and $F$ now includes all the functionalities of $F^l_{DAA}$. This concludes the proof. □

## 7.3 Security Analysis of Revocation Scheme (based on DAA)

As described in Chapter 6, the intuition behind the revocation scheme in ASSURED, is to be able to de-activate any **credentials** and **short-term signature keys**, of an edge device, created during their enrollment [23] in the overall IoT ecosystem: essentially, revoking the use of any *Attestation Key (AK)* and/or *pseudonyms*, created during the $DAA\ JOIN$ phase (Section 3.3.3) in case where the device participation and data contributions must adhere to any pre-defined privacy policies (crf. "*Secure Collaboration of Platforms-of-Platforms* for Enhanced Public Safety" [24]). For instance, as aforementioned, separate data bundles - produced by the same device - **need to be sent in an authenticated but unlinkable manner**: no data operations can be linked back to the origin device ID, hence, the use of a different pseudonym per data bundle (unconditional anonymity).

*Note that in all the following security properties/proofs there is the inherent assumption that the revocation message, when received by a device, is forwarded correctly to the TPM-based Wallet. The attacker model also allows a device to block messages intended for the TPM-based Wallet, including revocation requests. In order for revocation to take effect in this case, the TPM-based Wallet needs to detect that this has occurred. This can be achieved by a heartbeat mechanism [84], such that the TPM-based Wallet periodically expects either a revocation message or a heartbeat (which may be a revocation intended for some other device, or else a timed message). Revocation messages and heartbeats include information about the period they are intended for, thus, a heartbeat for one period cannot be used at a different time.*

**Property 1** (Attestation Key Restriction)**.** *The use of an Attestation Key (or other signing key) can only be allowed when the device is not compromised and, thus, revoked.*

This property is achieved through the **TPM-based Wallet** that is responsible for the AK creation and management. Based on the trust model defined for the secure key management in ASSURED [21], the use of an AK is binded/protected to the correct state of a device:*only if the current device state matches to the digest list (Section 5.3), circulated during the enrollment of the device, depicting the correct, expected state of a device, will the TPM-based Wallet allow the use of the AK for signing any subsequent attestation processes and traces*. This is enforced through the creation of appropriate **policy digests** that will protect the AK within the wallet. Thus, in the case of revocation, due to possible device misbehavior, this *device rogue state* will violate the policy which, in turn, will "lock" the usage of the AK. The symbolic analysis of the creation and certification of the AK as part of a remote attestation process has been described in [43].

**Property 2** (Revocation Assurance)**.** *The platform cannot, under any circumstances, execute (cryptographic or other) operations using a revoked short-term anonymous signing key (e.g., pseudonym).*

**Setup:** A rogue device host wishes to execute a (cryptographic) operation with revoked - but correct - pseudonym $p$. Correct implies that it satisfies the policy-requirements meaning that it has been linked to an appropriate revocation index (Section 6.2.3) and has been registered with the RA, who verified its correctness prior activation. Revoked means that at least one of the linked indices (*revocation bits*) has been set by the RA due to a detected device misbehavior.

**Goal:** The goal of the rogue device host is for the *session digest* inside the TPM-based Wallet to match that of the pseudonyms policy digest since this will enable the attacker to bypass the policy and keep on using the pseudonym for signing (or other cryptographic) operations. It essentially tries to masquerade the setting of a revocation bit by matching what is the expected policy when

no revocation bit has been activated. Recall that the policy digest is the following, where $H$ is the chosen hashing-function, $0x00$ represents a zero-hash.

$$H(0x00 \,||\, \texttt{TPM\_CC\_PolicyNV}) \,||\, H(data \,||\, offset \,||\, operation) \,||\, IndexName)$$

**Execution:** It should be clear that changing the first or second parameter can never provide the correct digest. Therefore, a compromised host has two options: a) Change the index name (pointing to the placeholder of the specific revocation bit) to trick the operation to succeed, or b) Change the operation to match the content of the Index. In the first approach, the `IndexName` parameter would be different in comparison with what has been set during the creation of the pseudonym (during the $DAA\ JOIN$ phase). This would make the digest different, and the TPM-based Wallet would reject the operation. In the second approach, the host could try to change the `args` (the inner hash) of the revocation command provided to the wallet in an attempt to reflect a different operation, revocation of different bits, or with a different offset so that the revocation of a different pseudonym takes place and the attacker can still use the original revoked signing key. All of which would also be reflected in the final hash, and would be again rejected.

**Conclusion:** In all possible attack scenarios, the device's TPM-based Wallet would reject the host's operation on the revoked pseudonym because of a wrong session digest (manipulation), or the failure of a math-comparison operation succeed (pseudonym is revoked). Recall, of course, the assumption that the calculation of all revocation indices and pseudonym-policies was performed correctly by the device host. In order to weaken these trust assumptions (Section 6.2.3.6), we need to include additional validation of the processes that are executed in the host and might try to manipulate the parameters given to the TPM-based Wallet during the revocation process [64].

**Property 3** (Immutability)**.** *The platform cannot unrevoke (or recreate) a short-term signing key (pseudonym) that has been deemed as revocable by a valid Revocation Authority.*

**Setup:** A compromised host has gone successfully through all the early phases of the creation and activation of short-term anonymous signing keys including Initialization of the *Ephemeral Index, Initialization of the Revocation Index, and Activation of the Revocation Index*.

**Goal:** The rogue host now wants to alter the contents of the revocation index, without satisfying the correct policy, so as to not enable the revocation of a specific key even if it has been validated by the respective RA.

**Execution:** We have two scenarios for consideration here: a) There is only one short-term singing key created mapped to one revocation index, and b) there are multiple indices mapped to the hierarchy of signing keys (pseudonyms) created (as mentioned before, this is the more usual case where a device uses a different pseudonym for signing each data bundle, thus, achieving unconditional anonymity).

In the first scenario, the rogue host can try to write to the revocation index (trying to change its value back to '$0$' and a non-revocable state) but this will fail due to these writing operations been protected by the calculated pseudonym policy. This policy states that any update policy digest, signed by the Authorization Key, is a valid new policy operation. The host might, therefore, try to sign a new policy with the authorization key so as to allow the unrevoke of the target pseudonym. However, this will also fail, as the overarching policy for using the authorization key is to prove to the underlying TPM-based Wallet that the device can authenticate to the Authorization Counter Index (ACI) prior to changing the revocation policy digest. This ACI contains an **authorization count** and an **authorization limit** - since the limit and count values are equal from the previous

operations, when setting the initial revocation policies during the creation and activation of the revocation bits, the host cannot no longer use the authorization key. She must, therefore, strive to change the authorization limit. However, for this to take place, the attacker needs to be able to update the ACI (Section 6.2.3.5); but this is also protected through another *ephemeral key* (primary key in the NULL hierarchy) that is created at the beginning of the revocation setup phase only for enabling the correct use of the ACI (Section 6.2.3.1). Since the device, according to the protocol, has executed a reboot, then the NULL-hierarchy seed has changed, and therefore the Ephemeral Key can never be re-created (immutability property of the TPM-based Wallet [21]). *Note, however, that ff the attacker is present during the revocation setup phase, she can write a different limit and create a personal policy, thus, allowing the creation of a new policy for breaking the pseudonym immutability property*.

In the secondary scenario, where we have multiple revocation indexes (pertaining to multiple pseudonyms), the use of the ephemeral key is replaced by the authorization key. That basically means that the authorization key has an extra authorization round (higher limit by one additional operation) from the ACI. A rouge host could use this to try to create and sign a new policy digest (as in the first scenario). However, we rely on the correct execution of these calculation phases as described in Section 6.2.3.6. If this is not the case then it should be the Security Context Broker (as a Trusted Third Party) to verify the correct calculation of the policies and issue appropriate certificates.

**Conclusion:** As long as the revocation setup phases are executed correctly, or verified by a Trusted Third Party, a rogue host cannot change the contents of a revocation index and, thus, manipulate the revocable state of a short-term signing key.

**Property 4** (Assurance of Revocation Requests)**.** *The platform should act only on authenticated revocation requests coming from a valid Revocation Authority.*

A device should only accept genuine revocation requests so as to ensure that attackers cannot arbitrarily revoke the credentials of other legitimate devices. This property is achieved by including the RA's signature on any revocation message ($cpHash$ for a specific pseudonym to be revoked), so that it will not be accepted by the TPM-based Wallet as a valid revocation unless the signature is present. Since the revocation message also includes the $cpHash$ associated to the pseudonym to be revoked, it cannot be reused by an attacker to revoke any other pseudonym.

**Property 5** (Assurance of Revocation Confirmation)**.** *Any device or other entity can verify the correct execution of a revocation operation.*

A key requirement of the revocation mechanism is to provide strong guarantees that when an RA has initiated and run the protocol to completion, then the associated TPM-based Wallet must have been involved in the protocol instance and correctly received the revocation request. It is the TPM-based Wallet who is responsible for de-activating the pseudonym certificates and no longer using them. In particular, the RA should not reach the point of believing that the revocation has taken place when in fact the TPM-based Wallet is unaware of it. This assurance is provided by the TPM-based Wallet signing the confirmation against the pseudonym public key pkps whose revocation is requested, and the RA can verify the TPM-based Wallet's signature on that. No other party can create this signature, and TPM-based wallet will only create this confirmation when $pk_{ps}$ is being revoked. Hence, no revocation confirmation can be used by an attacker to spoof a confirmation of any other revocation request.

## 7.4 Security Analysis of Configuration Integrity Verification

As described in Section 5.3, the goal of the Configuration Integrity Verification (CIV) scheme is to enable the verification of the correct configuration - in terms of loaded binaries (as *operational tasks*) - of an edge device prior to be allowed to join a network. More specifically, as part of the overall ASSURED attestation toolkit, the main goal is to create trust-aware service graph chains through the provision of **zero-touch configuration functionalities**: *platforms, wishing to join a network cluster, adhere to the compiled attestation policies (enforced through the ASSURED Blockchain infrastructure) by providing verifiable evidence on their configuration integrity and correctness.* In other words, the framework should provide guarantees that a node will be able to join a network (and participate in the underlying services) ***if and only if*** it can prove to the gateway that it is at a "correct state" - without, however, the gateway needing to know the node's state beforehand. This allows ASSURED to support the secure enrollment and integration of heterogeneous devices and platforms equipped with different computing resources and operating systems.

Of course, such a service can extend beyond the device enrollment phase: *At any point in time, a device may be required to attest to its correct configuration*. Consider, for instance, the case of **secure software update** where it is imperative for a device (that performed the update) to provide verifiable evidence not only on the correctness of the update output but also that this update did not alter the trust model for the overall device. As described in D3.1 [21], the following are the properties of interest [33, 63]:

**Property 6** (Configuration Correctness)**.** *A device's load-time and run-time configurations must have adhered to the latest attestation policy compiled by the Policy Recommendation Engine and authorized by Security Context Broker (SCB), prior to be recorded on the ledgers, in order to be verified as being correct by any other devices.*

This property is achieved as follows. Let an adversary $\mathcal{A}\mathrm{dv}$ extend the PCRs (see Figure 5.8) with measurements of her choice during a measurement update, and $\alpha$ be the configuration that SCB has requested to be measured. If $\alpha$ was altered without first recording its digest, $d \leftarrow \mathsf{H}(\alpha)$, where H is a collision resistant hash function, $\mathcal{A}\mathrm{dv}$ cannot win unless she picks a random digest $d'$, where $d' = d$. If $\alpha$ is unchanged, $\mathcal{A}\mathrm{dv}$ computes $d \leftarrow \mathsf{H}(\alpha)$ and supplies $d$. However, since $\alpha$ is correct, Property 6 is not violated. Now, assume that $\mathcal{A}\mathrm{dv}$ altered $\alpha$, but her chosen $d'$ is correct. Her next challenge is to guess the attestation agent's ($\mathcal{AA}\mathrm{gt}$) secret ($\mathtt{hk}$) to solve for $\mathsf{HMAC}(\mathtt{hk}, d')$. Unless she solves this challenge, she cannot extend the correct measurement, and verification of the session's audit digest will fail on SCB. Thus, circumventing the measurement process's integrity is infeasible, and thus the property holds. Further, by also including configuration metadata information (e.g., creation and modification timestamps, or `i_generation` and `i_version` for Linux kernel's mounted with *inode* versioning support) when $\mathcal{AA}\mathrm{gt}$ measures (calculates a digest over) the configurations, $\mathcal{A}\mathrm{dv}$ cannot unnoticeably alter and restore configurations between updates. However, although not covered by the property, alterations to the configurations currently remain undetected until the next measurement. We propose two directions to mitigate this Time-Of-Check to Time-Of-Use (TOCTOU) problem.

**Reactive (lazy) TOCTOU-resistance** The first approach is to require a device's attestation agent $\mathcal{AA}\mathrm{gt}$ to vouch for the configuration's correctness at the *time of processing the attestation request* by either: (*i*) comparing the metadata (e.g., the `i_generation` and `i_version` fields), or (*ii*) re-measuring the configuration. However, for this to be useful, considering that $\mathcal{A}\mathrm{dv}$ can block access to $\mathcal{AA}\mathrm{gt}$, we must extend the existing attestation policies (Section 5.3.2.3) to require proof that $\mathcal{AA}\mathrm{gt}$ handled the attestation. We can achieve this with

`TPM2_PolicyAuthValue` and require that $\mathsf{hk}_{(SCB,\mathcal{AA}\mathrm{gt})}$ be supplied (along with the necessary `TPM2_PolicyNV` and `TPM2_PolicyPCR` commands) for `TPM2_PolicyAuthorize` to succeed (see Figure 5.9). Note, however, since the command `TPM2_PolicyAuthValue` does not support limiting when authorization should expire, $\mathcal{AA}\mathrm{gt}$ must close the policy session's handle once it has signed the V's challenge. If $\mathcal{AA}\mathrm{gt}$ had an asymmetric key pair, we could have instead used `TPM2_PolicySigned`, which allows specifying when authorization to the AK expires, like a "dead man's switch".

**Proactive TOCTOU-resistance** Another approach is to extend $\mathcal{AA}\mathrm{gt}$ software to continuously monitor all objects in $FQPN \in REQ_{update}$ between updates. If the configuration changes, $\mathcal{AA}\mathrm{gt}$ effectively neuters the device's AK by extending the *active* PCRs. However, to achieve this efficiently is non-trivial. The most notable framework is the conjunction of the Integrity Measurement Architecture (IMA) [76] and Extended Verification Module (EVM), which for the Linux-based kernels, provide fine-grained mechanisms to measure and detect file alterations. IMA essentially extends measured boot into the OS, where, depending on a measurement policy (MP), files and binaries (objects) are measured and recorded in a measurement log (ML) and a TPM register. Depending on the enforced MP, IMA proceeds to continuously remeasure objects as they are accessed or changed during run-time. However, since IMA lacks support to change the MP during run-time, it is unfit in our case. Another increasingly popular method is the use of extended Berkeley Packet Filters (eBPF). With eBPF, extensions can be applied to the OS kernel during run-time, enabling (privileged) software to hook and filter system calls dynamically. Employing the bpftrace [74] tool or BPF Compiler Collection (BCC) toolkit, we can instrument $\mathcal{AA}\mathrm{gt}$ to attach hooks (or *probes*) on file-related system calls and match calls targeting the configurations. For example, to detect writes and deletions we can attach `sys_enter_write` and `vfs_unlink` probes, and to catch calls that open configuration files in modes other than *read-only*, we can leverage `sys_enter_openat`. Note, however, that additional probes are required in practice since files can also be written in other ways (e.g., using `mmap`). Nonetheless, utilizing eBPF, we can effectively and preemptively mitigate the TOCTOU problem.

**Property 7** (Secure Enrollment)**.** *A device's enrollment involves the Privacy CA certifying the device in creating an acceptable Attestation Key (AK), which is certified to be used under the correct policy for signing produced traces.*

To ensure that the Privacy CA certifies the correct creation of all AKs, they must be created to only abide by policies outputted by the Policy Recommednation Engine and signed by the Privacy CA (acting as the Issuer). Since an AK must be certified by the device's EK (using `CertifyCreation`) to be accepted by the Privacy CA, where EK, for all devices, is a credentialed non-duplicable EK (*restricted* signing key) that can only sign TPM-generated data, $\mathcal{A}\mathrm{dv}$ can neither fool the Privacy CA to accept a self-signed creation certificate nor have the EK sign a $\mathcal{A}\mathrm{dv}$-forged certificate. Also, if any details in the AK's certificate (e.g., its attributes, name, or authorization policy) are incorrect, the Privacy CA rejects it. Thus, $\mathcal{A}\mathrm{dv}$ cannot threaten the AK creation process's integrity. Note that *forward acceptance* (Property 8) is ensured during AK creation by requiring that the authorization policy be a flexible policy. Thus, Property 7 $\implies$ Property 8.

**Property 8** (Forward Acceptance)**.** *To prevent excessive AK recreation and redistribution, all AKs are created such that they can be continuously re-purposed (i.e., in which policy they attest) as determined and authorized through any newly generated attestation policies as smart contracts.*

**Property 9** (Freshness)**.** *To ensure non-ambiguous verification, a device can have at most one policy that unlocks its AK.*

Given a configuration update $h_{update}$ for a device, $\mathcal{X}$, SCB uses Algorithm 4 (Figure 5.8) with $\mathcal{X}$'s *current* mock structures, $mPCR, mNVPCR$, to compose a policy which accounts for the update. The algorithm performs the following actions: (*i*) accumulate $h_{update}$ into the appropriate mock PCR (lines 1 to 7); (*ii*) initialize a policy $h_{pol}$ (line 8); (*iii*) extend $h_{pol}$ with a simulation, where, for each mocked NV PCR $i$, `TPM2_PolicyNV` is executed with $i$'s current measurement (lines 8 to 11); (*iv*) if nonempty, extend $h_{pol}$ with a simulation, where all PCRs in $mPCR$ are selected and their accumulated digest is supplied to `PolicyPCR` (lines 12 to 18); (*v*) sign H($h_{pol}$). The signature and $h_{pol}$ are then sent to $\mathcal{X}$, where $\mathcal{A}$dv also has access to it. Given the authorized $h_{pol}$, AK is unlocked using `PolicyAuthorize` if, after executing the *exact same* sequence of commands, the TPM's internally accumulated digest $h'_{pol}$ is equal to the authorized digest: $h'_{pol} = h_{pol}$.

When, at a later time, $\mathcal{X}$ must account for another update, $h'_{update}$, its *current* mock structures $mPCR', mNVPCR'$ are again used to authorize a new policy digest $h''_{pol}$. However, if $\{indicies(mPCR') \cap indicies(mPCR)\} \cup \{indicies(mNVPCR)' \cap indicies(mNVPCR)\} = \emptyset$, then $h_{pol}$ and $h''_{pol}$ share no elements (PCRs), which means that both policies can simultaneously unlock $\mathcal{X}$'s AK. Thus, when another device, $\mathcal{Y}$, wants to verify $\mathcal{X}$'s correctness, it is undefined *which* policy it fulfills. It is therefore necessary that policies are either (*i*) created with *at least one* element in common with the preceding policy and that this element be extended to neuter the preceding policy, or (*ii*) followed by another command which extends one PCR of the preceding policy.

**Property 10** (Zero-Knowledge CIV)**.** *To keep configurations confidential, any device should only require another device's AK's public part to verify its configuration correctness.*

When a Gateway, $\mathcal{X}$, receives a request to join the network from a device $\mathcal{Y}$, it first queries the Blockchain for downloading the appropriate policy to execute in order to verify the correct state of $\mathcal{Y}$. Through the downloaded policy, it receives $\{\mathsf{Sig}^{EK^{SCB}_{\mathsf{sk}}}_{AK^{\mathcal{Y}}_{\mathsf{pk}}}, AK^{\mathcal{Y}}_{\mathsf{pk}}\}$ (Figure 5.4). If Property 6 $\wedge$ Property 7 $\wedge$ Property 9 hold, then $\mathcal{Y}$ is correctly associated with $AK^{\mathcal{Y}}_{\mathsf{pk}}$. Thus, if $\mathcal{X}$ chooses a random number $n \leftarrow\!\!\$\, \{0,1\}^t$ and $\mathcal{Y}$ presents $\mathsf{Sig}^{AK^{\mathcal{Y}}_{\mathsf{sk}}}_n$, where $\mathsf{Enc}(\mathsf{Sig}^{AK^{\mathcal{Y}}_{\mathsf{sk}}}_n, AK^{\mathcal{Y}}_{\mathsf{pk}}) = n$, then $\mathcal{X}$ knows that $\mathcal{Y}$ was able to use $AK^{\mathcal{Y}}_{\mathsf{sk}}$ and thence fulfills SCB's requirements. Thus, Property 6 $\wedge$ Property 7 $\wedge$ Property 9 $\implies$ Property 10.

## 7.5 Security Analysis of Control-Flow Attestation

As described in Section 3.2, and elaborated in Section 5.1, ASSURED Control-Flow Attestation (CFA) relies mainly on the trained Deep Neural Network (DNN) for verifying the state of a Prover device $\chi$, during run-time. However, as the *level of accuracy* offered by the employed DNN towards verifying the run-time behavior of a complex program, in comparison to state-of-the-art solutions, is an open research question and will be documented in D6.2 [30] (based on the detailed evaluation and experimentation of the designed CFA in the context of the envisioned use cases), the focus in the reminder of this section is on the secure interaction of the ASSURED Attestation Agent with the remaining part of the system; i.e., **verify the authenticity of its inputs (as originate from the Tracer) and outputs when it comes to the correctness of the attestation result**.

**Property 11** (Integrity & Authentication of Attestation Inputs)**.** *An adversary $\mathcal{A}$ cannot manipulate the inputs for the attestation, i.e., the control flow traces, to pretend a benign state for a compromised device. Not she can impersonate a valid Tracer for producing falsified control-flow traces.*

As the Verifier $V$ is placed on a different device than the Prover $\chi$, it requires trusted inputs for the verification. Otherwise, an adversary $\mathcal{A}$ could provide manipulated inputs for pretending a benign state for a compromised device.To prevent this, the verification inputs, i.e., the traces $T$, are signed by the Prover's TPM-based Wallet: The collected traces $T$ extracted by the Tracer are passed to the TPM-based Wallet (as the underlying secure token) for been signed with the created and certified $AK$. Recall that the software stack for interacting with the TPM-based Wallet is not part of the overall trusted computing base. Thus, it might be the target of the adversary in order to compromise the authenticity and integrity of the traces. To achieve this, she can either intercept the valid traces outputted by the Tracer and replace them with some rogue ones or directly pass these falsified traces to the TPM. This, however, will not be allowed by the TPM itself since the use of the AK is protected by a $PolicySigned$ command as detailed in D3.1! [21]: *Only traces that have been signed with the Tracer's private key can be accepted by the TPM that will verify their signature prior to also signing them using its internal $AK$ protected also through the device's $EK$ (Figure 3.3 in D3.1! [21]).* In the end, **the Authorization Signature, the TPM Signature, the Traces and the TPM Public Key Data** is sent to the Verifier, who then verifies a) the policy, b) the authorization signature, c) the signature over the traces. Thus, any manipulation by the adversary will be detected. The Verifier needs to check the signature of the TPM Wallet, plus the correctness of the command executed, for validating the integrity of the traces $VERIFY(K_{W_{Pub}}, T)$, due to the hardware-based trust anchors.

**Property 12** (Freshness)**.** *An adversary $\mathcal{A}$ cannot use previously (intermediate) attestation results for faking the state of a prover.*

To ensure the freshness of the attested properties, two attack vectors need to be considered: attacks that target the **freshness of the traces** $T$ and attacks that target the **freshness of the attestation report** $R$**.**

**Freshness of Attestation Input.** To prevent attacks that reuse previously collected traces, every attestation task contains a nonce value $N$. The nonce value $N$ is determined based on the attestation policy $P$, depicted as a smart contract, and is calculated based on the hash of the last block header of the ledger. Therefore, the device once downloading the attestation policy to execute, it first interacts with the Blockchain Peer Node for remotely invoking the $getNonce$ method that returns a fresh nonce value $N$ [27]. Since $N$ differs for every attestation and the signature $SIGN_{K_{W_{Pr}}}(T)$ of the traces is verified as $VERIFY(K_{W_{Pub}}, T)$, the freshness of the traces is guaranteed. Furthermore, the nonce (or more precisely the block header that was used for calculating the nonce) is also registered on the ledger, as part of the attestation report, so as any other external verifying entity can audit the correctness of the nonce that was used during the attestation process. This enables us to also detect any cheating attempts from the Verifier itself; i.e., if it attempts to use another nonce than the one calculated in order to replay a previous recorded state for the Prover device.

**Freshness of Attestation Report** $R$**.** After the Verifier validates the integrity of the control-flow traces $T$, it creates an attestation report $R$, stating the trustworthiness of the Prover $\chi$ with respect to its **execution correctness**. An adversary $\mathcal{A}$ might record such an attestation report before compromising $\chi$ and using the captured attestation report for pretending a benign state of $\chi$. To guarantee the freshness of the attestation reports, every report contains another nonce $N$ (generated by the TPM-based Wallet of the Verifier) and a timestamp denoting the date and time, when the traces $T$ where recorded. Each of these individual properties allows to verify the freshness of the provided attestation report $R$ and detect, e.g., replay attacks. In combination with the signature $R$, obtained by using the Verifier's AK $K_{V_{AK}}$ for executing the signing as $SIGN_{K_{V_{AK}}}$

the receiver of $R$ can verify the freshness of $R$. This is also validated by the Blockchain Peer Node prior to recording the attestation result on the ledger. In case of a replay attack and due to lack of evidence of who launched the attack (Prover, Verifier or another insider attacker), the Blockchain Peer will trigger the Jury-based Attestation for further investigation.

## 7.6 Security Analysis of Jury-based Attestation

As described in Section 3.5, ASSURED Jury-based Attestation scheme acts as a second line of defense towards detecting possible misbehaviors during the execution of a (run-time) remote attestation process; either Configuration Integrity Verification or Control-flow Attestation. Consider, for instance, the case where the Verifier (e.g., IoT Gateway) has been compromised and, thus, records a falsified attestation report for a new device trying to enroll to the network. This essentially will result in a negative output while executing the "*Secure Device Enrollment*" policy. However, this sets the challenge ahead: *How do we make sure of the correctness of the Verifier that is responsible for attesting a newly joined device?* If the Verifier is compromised, when the attestation result is been broadcasted back to the Blockchain Peer Node, the Prover will overhear the message and report to the SCB of the wrong result. In this case, we need to have an additional step for identifying which of the participating entities is lying: **verify the evidence provided by both the Prover and Verifier so as to detect and revoke the misbehaving entity**.

On the one hand, the security of the attestation process per se used for the jury-based attestation relies on the correctness of the underlying attestation schemes, as described in the previous sections. Especially, the CFA that is usually leveraged as the verification scheme. Thus, the jury-based attestation inherits any security guarantees of the leveraged attestation blocks. On the other hand, the Byzantine agreement among the $j$ elected jurors is based on Practical Byzantine Fault Tolerance (PBFT) [17], which fails to guarantee honest outcomes if more than $\lfloor (j-1)/3 \rfloor$ jurors are adversarial.

The adversarial share of nodes the jury-based attestation can tolerate depends on the probabilities of electing compromised devices as jurors [2]. As the election is random, it may happen that enough adversarial nodes are elected for the Byzantine agreement to fail. More specifically, when enough adversarial jurors are elected to reach a quorum, they can enforce a malicious decision, i.e., violating PBFT's *safety* guarantee. Further, the consensus can also fail due to not reaching a quorum at all, i.e., violating the *liveness* guarantee. In the following we discuss the probabilities for different adversary shares as well as the chosen jury size $j$. While a larger $j$ reduces the chances of a failed election, PBFT also induces a message complexity of $O(j^2)$.

**Property 13** (Probabilistic Safety)**.** *Due to the random nature of the election and the safety threshold of the underlying BFT protocol, the safety for decisions among any elected jury is based on probabilities.*

If we have $n$ total nodes in our system, with $f$ of them being adversaries and elect $j$ jurors, the probability of electing at least $\lfloor (j-1)/3 \rfloor$ adversarial nodes is:

$$1 - \frac{\binom{j}{k+1}\binom{n-j}{f-k-1}}{\binom{n}{f}} \, {}_3F_2\left[\begin{matrix} 1,\ k+1-f,\ k+1-j \\ k+2,\ n+k+2-f-j \end{matrix} ; 1\right] \tag{7.1}$$

Where $k = \lfloor (j-1)/3 \rfloor$ and ${}_pF_q$ is the generalized hypergeometric function. Equation 7.1 is the cumulative distribution function of the hypergeometric distribution.

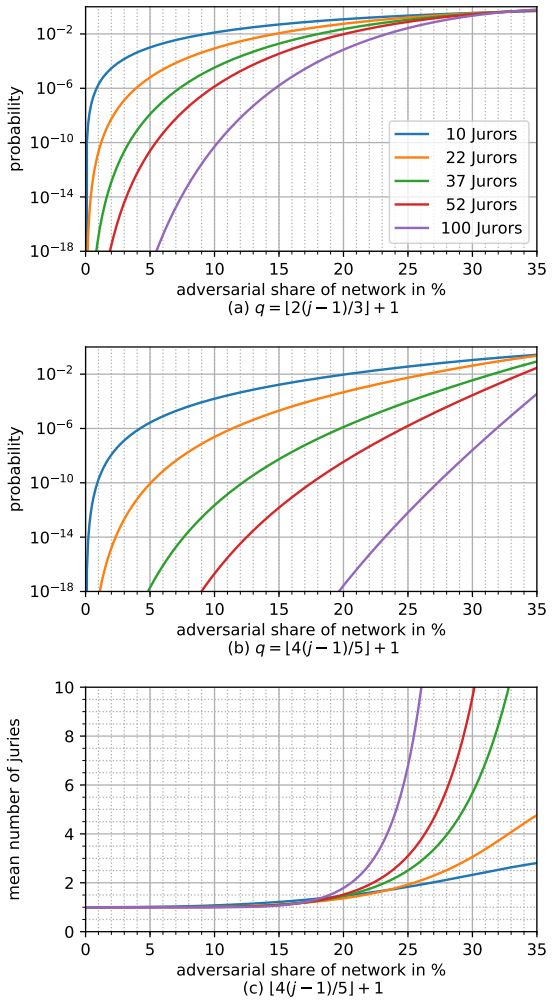



Figure 7.2: The probability of eventual safety violation of Byzantine agreement with a population size of $10\,000$ given a threshold of $q$ in (a) and (b), as well as the mean number of juries needed before agreement terminates, whether in success or *total* failure (c)

**Property 14** (Probabilistic Termination)**.** *Due to the possibility of a liveness fail of the underlying BFT protocol, there is a mean number of re-elections required to find a jury quorum for any decision in the network.*

While equation 7.1 models the probability for the Byzantine agreement to fail, we can rectify a liveness violation by re-election. In some applications, it may make sense to accept reduced fault-tolerance by increasing the quorum size required by PBFT from $\lfloor 2(j-1)/3 \rfloor + 1$ to some greater value $q$. Then, $n - q \leq \lfloor (j-1)/3 \rfloor$ faults are sufficient to cause a liveness violation, but a safety violation requires a greater number $2q - n$ of faults. If the protocol reaches an impasse, another consensus round, including a new jury, is started that may succeed. This can be modelled as a Markov chain: we begin in an initial "undecided" state and transition to a "success" state if no more than $n - q$ adversarial nodes are elected—guaranteeing agreement—and a "failure" state if at least $2q - n$ adversarial nodes are elected—allowing a safety violation. The failure state will eventually be reached with probability

$$\mathrm{P}[\text{Eventual Failure}] = \frac{\mathrm{P}[F \geq 2q - n]}{\mathrm{P}[F \geq 2q - j] + \mathrm{P}[F \leq j - q]} \tag{7.2}$$

and it will take on average $1/\mathrm{P}[j - q < F < 2q - j]$ elections to leave the “undecided” state.
Besides the threshold $q$, a primary factor affecting the probability of an eventual safety violation is the jury size $j$. The more jurors are elected per round, the lower the probability for the Byzantine agreement to fail.
Besides the threshold $q$, a primary factor affecting the probability of an eventual safety violation is the jury size $j$. The more jurors are elected per round, the lower the probability for the Byzantine agreement to fail. We illustrate the influence of the jury size $j$ and BFT threshold $q$ in Figure 7.2.

# Chapter 8

# Conclusion

This final section will act as a synopsis of this deliverable and summarize its findings. The scope of this deliverable was to provide the **mode of operation, work-flow and building blocks** of the newly introduced ASSURED trust extensions; as part of the first release of the ASSURED set of attestation schemes. This is considered as one of the main goals towards "**security and privacy by design**" solutions, including all methods, techniques, and tools that aim at enforcing security and privacy at software and system level from the conception and guaranteeing the validity of these properties. For privacy, ASSURED leverages advanced crypto primitives, namely Direct Anonymous Attestation (DAA), whereas for security and operational assurance, it enables the provision of Control-flow Attestation, Platform Integrity Verification and Swarm Attestation. it also employs Jury-based Attestation as a second line of defense in the case of contradicting statements between a Prover and Verifier devices engaging in a dynamic remote attestation process.

As part of the overall ASSURED attestation toolkit, the main goal is to create **trust-aware service graph chains** through the provision of **zero-touch configuration functionalities**: *platforms, wishing to join a network cluster, adhere to the compiled attestation policies (enforced through the ASSURED Blockchain infrastructure) by providing verifiable evidence on their configuration integrity and correctness.* In other words, the framework should provide guarantees that a node will be able to join a network (and participate in the underlying services) ***if and only if*** it can prove to the gateway that it is at a "correct state" - without, however, the gateway needing to know the node's state beforehand. This allows ASSURED to support the secure enrollment and integration of heterogeneous devices and platforms equipped with different computing resources and operating systems.

In this context, the ASSURED collective attestation algorithms provide a multi-level security verification mechanism for supporting trust aware service graph chains (based on the identified security attestation policies) on the integrity assurance and correctness of the comprised devices: **from the trusted launch and configuration to the run-time attestation of low-level configuration and behavioural properties**. Based on our analysis, we described how a device achieves privacy-preserving integrity correctness and how to utilize the attached TPM-based Wallet for binary data integrity as well as operational and threat intelligence sharing with the policy-compliant Blockchain infrastructure. Our early implementation and evaluation results demonstrate that trust enablers can satisfy the privacy, security, and efficiency requirements.

Our novel architecture yields many advantages over state-of-the-art solutions in terms of *security, privacy, and scalability*. Most notably one of the biggest advantages of such a decentralized approach is its **scalability**, as trust is shifted from the back end infrastructure to the Edge. Other benefits span from performance effectiveness to stronger privacy protection and revocation capabilities:

**Less Costs compared to PKI:** PKI's complex architecture is expensive to operate due to the following reasons: (i) **Organizational Separation** - PKI requires organizational separation to assure that PKI entities do not exchange information. This increases the operational costs to a level hard to manage. In ASSURED there is no need for a complex infrastructure to support the issuance and management of "*device state ok*" evidence. There are just two central entities required: *the Security Context Broker and the Blockchain Node*. There are no requirements on organizational separation because the trust is shifted on the edge device. So the costs for operating the infrastructure is very low, and (ii) **Operation Costs:** Typical PKI processes involve human personnel. This creates a permanent Operating Expenditure (OPEX). **Since ASSURED distributes the key creation and management to the devices, this will reduce operational requirements and thus OPEX for the infrastructure operator.**

**Stronger Privacy Protection:** In the current PKI landscape with multiple authorities, it is yet not clearly defined who will operate the identity and credential provision and how trust relationships will be established. It is not precluded that multiple authorities can be operated by one organization. This is not necessarily a problem as long as there are appropriate policies in place that prevent, for example, one person from being able to access information at more than one component of the PKI. The employment of a PCA (or multiple virtual instances of it) that issues all pseudonyms to the requesting devices, is an indicative example of the raising privacy concerns with a direct impact on the anonymity of the platforms. Therefore, **in PKI solutions, there are strong assumptions on the trustworthiness of the PCA so that the pseudonyms produced offer strong privacy protection to the required group**. However, in such a setting privacy assurance depends on how many devices are in the group. Since the PCA has access to all provided pseudonyms, in the case of a sparse network deployment, what is the impact on the underlying anonymity set? It has been shown that when a change of pseudonyms is triggered by a relatively small number of devices (i.e., $< 20$ devices in a mix-zone), the PCA can with a certain probability (around $33\%$) link pseudonymous location samples to each other (even when constructed under different pseudonyms) [85]. **In DAA-based architectures, there is no central entity that knows and monitors all issued pseudonyms. This allows the re-usage of the pseudonyms much more often, without any impact on the privacy of the devices. By having better pseudonym re-usage we can reduce verification times even more.**

**Efficient revocation:** The revocation service in our model provides strong guarantees of successful completion when a misbehaviour has been identified and reported correctly using existing protocols. This is mainly due to the presence of the TPM-based Wallet who is responsible for executing the revocation command, thus, not allowing to be circumvented by a (compromised) platform. Secondly, through the use of DAA deterministic signatures and link tokens, revocation under changing pseudonyms is still possible and the SCB can verify revocation messages without compromising the device's privacy. Overall, such a DAA-based solution avoids several of the shortcomings of the revocation solutions in PKI systems [41, 46, 70] and namely: a) it does not require pseudonym resolution in order to trace a pseudonym back to the misbehaving device's long-term identifier, b) it does not require the use and distribution of CRLs, and c) the devices do not need to periodically connect to the SCB for "pulling" the latest version of the revocation blacklist. So, our solution minimizes bandwidth and connectivity requirements since it is based on the broadcast of a short revocation message containing the pseudonym of the misbehaving device. Regarding the **no-use of CRLs**, the proposed revocation mechanism triggers the TPM-based Wallet to delete all of its secrets, thus, not allowing any subsequent (authorized) communication from the misbehaving platform. This is done by executing one TPM command that deletes the root key from the TPM-based Wallet, so it's much more efficient. We avoid all the complexity of managing and distributing CRLs. Finally, **revocation is immediate**. Once a device is revoked,

the revocation takes effect immediately.

We also formalized the notion of secure remote attestation towards trust aware service graph chains and presented security analyses of all our schemes against the key security properties that were defined in D3.1 [21]. We presented our first step towards the security modelling of the entire ASSURED ecosystem leveraging a TPM as the underlying root-of-trust coupled with the verification of its security properties. This is considered as a stepping stone for enabling the formal verification of all ASSURED attestation mechanisms based on the introduction of "*trusted platform command abstractions*". Towards this direction, we compiled a: (i) newly introduced verification methodology, based on a "bottom up" approach, in which the focus is on modelling the core attestation functionalities towards building chains of trust, and (ii) formalization of idealized TPM functionalities along with their security properties and a realistic adversarial model. *This break-down of TPM ideal functionalities and services allows for a more effective verification process towards building a global picture of the entire TPM platform security modelling as a Root-Of-Trust.*

# 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] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems. In *NDSS*, 2019.

[2] Tigist Abera, Ferdinand Brasser, Lachlan Gunn, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. Granddetauto: Detecting malicious nodes in large-scale autonomous networks. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 220–234, 2021.

[3] Tigist Abera et al. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proceedings of the 2016 ACM SIGSAC CCS Conf.*, pages 743–754.

[4] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. Sana: secure and scalable aggregate network attestation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 731–742, 2016.

[5] Moreno Ambrosin, Mauro Conti, Riccardo Lazzeretti, Md Masoom Rabbani, and Silvio Ranise. Pads: Practical attestation for highly dynamic swarm topologies. In *2018 International Workshop on Secure Internet of Things (SIoT)*, pages 18–27. IEEE, 2018.

[6] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. Simple: A remote attestation approach for resource-constrained iot devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

[7] Mahmoud Ammar, Mahdi Washha, and Bruno Crispo. Wise: Lightweight intelligent swarm attestation scheme for iot (the verifier's perspective). In *2018 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–8. IEEE, 2018.

[8] Mahmoud Ammar, Mahdi Washha, Gowri Sankar Ramabhadran, and Bruno Crispo. slimiot: Scalable lightweight attestation protocol for the internet of things. In *2018 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–8. IEEE, 2018.

[9] Opinion 03/2017 on Processing personal data in the context of Cooperative Intelligent Transport Systems (C-ITS). Document, October 2017.

[10] Nadarajah 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*, pages 964–975, 2015.

[11] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40, 2011.

[12] Ernest F. Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *ACM Conference on Computer and Communications Security, CCS*, 2004.

[13] Ernie Brickell, Liqun Chen, and Jiangtao Li. A new direct anonymous attestation scheme from bilinear maps. In *International Conference on Trusted Computing*, pages 166–178. Springer, 2008.

[14] Ernie Brickell, Liqun Chen, and Jiangtao Li. Simplified security notions of direct anonymous attestation and a concrete scheme from pairings. *International journal of information security*, 8(5):315–330, 2009.

[15] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. In *Public-Key Cryptography – PKC 2016*, volume 9615 of *LNCS*, pages 234–264. Springer, 2016.

[16] Xavier Carpent, Karim ElDefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Lightweight swarm attestation: a tale of two lisa-s. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 86–100, 2017.

[17] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI, 1999.

[18] Liqun Chen, Dan Page, and Nigel P Smart. On the design and implementation of an efficient daa scheme. In *International Conference on Smart Card Research and Advanced Applications*, pages 223–237. Springer, 2010.

[19] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[20] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, page 38–49, New York, NY, USA, 2016. Association for Computing Machinery.

[21] The ASSURED Consortium. Assured attestation model & specification. Deliverable D3.1, November 2021.

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

[23] The ASSURED Consortium. Assured blockchain architecture. Deliverable D1.4, November 2021.

[24] The ASSURED Consortium. Assured reference architecture. Deliverable D1.2, May 2021.

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

[26] The ASSURED Consortium. Operational sos process models & specification properties. Deliverable D1.3, 2021.

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

[28] The ASSURED Consortium. Risk assessment methodology & threat modelling. Deliverable D2.1, November 2021.

[29] The ASSURED Consortium. Assured secure and scalable aggregate network attestation. Deliverable D3.6, February 2022.

[30] The ASSURED Consortium. First demonstrators implementation report. Deliverable D6.2, 2022.

[31] Mauro Conti, Edlira Dushku, and Luigi V Mancini. Radis: Remote attestation of distributed iot services. In *2019 Sixth International Conference on Software Defined Systems (SDS)*, pages 25–32. IEEE, 2019.

[32] Heini Bergsson Debes and Thanassis Giannetsos. Segregating keys from noncense: Timely exfil of ephemeral keys from embedded systems. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 92–101, 2021.

[33] Heini Bergsson Debes, Thanassis Giannetsos, and Ioannis Krontiris. BLINDTRUST: oblivious remote attestation for secure service function chains. *CoRR*, abs/2107.05054, 2021.

[34] Rafaël del Pino, Vadim Lyubashevsky, and Gregor Seiler. Lattice-based group signatures and zero-knowledge proofs of automorphism stability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 574–591. ACM, 2018.

[35] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[36] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[37] Edlira Dushku, Md Masoom Rabbani, Mauro Conti, Luigi V Mancini, and Silvio Ranise. Sara: Secure asynchronous remote attestation for iot systems. *IEEE Transactions on Information Forensics and Security*, 15:3123–3136, 2020.

[38] Benjamin Edelman. Adverse Selection in Online 'Trust' Certifications and Search Results. In *Electronic Commerce Research and Applications 10*, pages 17–25, 2011.

[39] Nada El Kassem, Liqun Chen, Rachid El Bansarkhani, Ali El Kaafarani, Jan Camenisch, Patrick Hough, Paulo Martins, and Leonel Sousa. More efficient, provably-secure direct anonymous attestation from lattices. *Future Generation Computer Systems*, 99:425–458, 2019.

[40] ETSI. Trust and Privacy Management, 2012. http://www.etsi.org/deliver/etsi_ts/102900_102999/102941/01.01.01_60/ts_102941v010101p.pdf [Online; accessed 26-August-2017].

[41] David Förster, Hans Löhr, Jan Zibuschka, and Frank Kargl. REWIRE – Revocation Without Resolution: A Privacy-Friendly Revocation Mechanism for Vehicular Ad-Hoc Networks. In *Trust and Trustworthy Computing*, 2015.

[42] 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.

[43] Georgios Fotiadis, Jose Moreira-Sanchez, Thanassis Giannetsos, Liqun Chen, Peter B. Ronne, Mark Ryan, and Peter Y.A. Ryan. Root-of-trust abstractions for symbolic analysis: Application to attestation protocols. In *STM 2021: Security and Trust Management*, September 2021.

[44] Thanassis Giannetsos and Tassos Dimitriou. Spy-sense: Spyware tool for executing stealthy exploits against sensor networks. In *Proceedings of the 2Nd ACM Workshop on Hot Topics on Wireless Network Security and Privacy*, HotWiSec '13, pages 7–12, 2013.

[45] Thanassis Giannetsos and Ioannis Krontiris. Securing V2X Communications for the Future: Can PKI Systems Offer the Answer? In *14th Int. ARES Conf*, 2019.

[46] Stylianos Gisdakis, Marcello Lagana, Thanassis Giannetsos, and Panos Papadimitratos. SEROSA: service oriented security architecture for vehicular communications. In *VNC*, pages 111–118. IEEE, 2013.

[47] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 1989.

[48] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, 2014.

[49] Ragnar Mikael Halldórsson, Edlira Dushku, and Nicola Dragoni. Arcadis: Asynchronous remote control-flow attestation of distributed iot services. *IEEE Access*, pages 1–15, 2021.

[50] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585, 2012.

[51] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, P. Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[52] Jianxing Hu, Dongdong Huo, Meilin Wang, Yazhe Wang, Yan Zhang, and Yu Li. A probability prediction based mutable control-flow attestation scheme on embedded platforms. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 530–537. IEEE, 2019.

[53] Dongdong Huo, Yu Wang, Chao Liu, Mingxuan Li, Yazhe Wang, and Zhen Xu. Lape: A lightweight attestation of program execution scheme for bare-metal systems. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE*

*18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 78–86. IEEE, 2020.

[54] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. Us-aid: Unattended scalable attestation of iot devices. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30. IEEE, 2018.

[55] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. Healed: Healing & attestation for low-end embedded devices. In *International Conference on Financial Cryptography and Data Security*, pages 627–645. Springer, 2019.

[56] 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 & Privacy in Wireless and Mobile Networks*, pages 171–182, 2016.

[57] Hyperledger Sawtooth documentation on Proof-of-Elapsed-Time. https://sawtooth.hyperledger.org/docs/core/nightly/0-8/introduction.html, 2020.

[58] Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, and Stefan Katzenbeisser. Scapi: a scalable attestation protocol to detect software and physical attacks. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 75–86, 2017.

[59] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. Salad: Secure and lightweight attestation of highly dynamic and disruptive networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 329–342, 2018.

[60] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. A practical attestation protocol for autonomous embedded systems. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 263–278. IEEE, 2019.

[61] Boyu Kuang, Anmin Fu, Shui Yu, Guomin Yang, Mang Su, and Yuqing Zhang. Esdra: An efficient and secure distributed remote attestation scheme for iot swarms. *IEEE Internet of Things Journal*, 6(5):8372–8383, 2019.

[62] Boyu Kuang, Anmin Fu, Lu Zhou, Willy Susilo, and Yuqing Zhang. DO-RA: data-oriented runtime attestation for IoT devices. *Computers & Security*, 97:101945, 2020.

[63] Benjamin Larsen, Heini Bergsson Debes, and Thanassis Giannetsos. Cloudvaults: Integrating trust extensions into system integrity verification for cloud-based environments. In *Computer Security*, pages 197–220, Cham, 2020. Springer International Publishing.

[64] 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, page 48–59, New York, NY, USA, 2021.

[65] San Ling, Khoa Nguyen, and Huaxiong Wang. Group signatures from lattices: simpler, tighter, shorter, ring-based. In *IACR International Workshop on Public Key Cryptography*, pages 427–449. Springer, 2015.

[66] Jingbin Liu, Qin Yu, Wei Liu, Shijun Zhao, Dengguo Feng, and Weifeng Luo. Log-based control flow attestation for embedded devices. In *International Symposium on Cyberspace Safety and Security*, pages 117–132. Springer, 2019.

[67] M. E. Nowatkowski, J. E. Wolfgang, C. McManus, and H. L. Owen. The effects of limited lifetime pseudonyms on certificate revocation list size in VANETS. In *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pages 380–383, March 2010.

[68] R. Olfati-Saber and J.S. Shamma. Consensus filters for sensor networks and distributed sensor fusion. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 6698–6703, 2005.

[69] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615, 2012.

[70] J. Petit, F. Schaub, M. Feiri, and F. Kargl. Pseudonym schemes in vehicular networks: A survey. *IEEE Communications Surveys Tutorials*, 17(1):228–255, 2015.

[71] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX security symposium*, pages 179–194. San Diego, USA, 2004.

[72] Md Masoom Rabbani, Jo Vliegen, Jori Winderickx, Mauro Conti, and Nele Mentens. Shela: Scalable heterogeneous layered attestation. *IEEE Internet of Things Journal*, 6(6):10240–10250, 2019.

[73] Wei Ren, R.W. Beard, and E.M. Atkins. A survey of consensus problems in multi-agent coordination. In *Proceedings of the 2005, American Control Conference, 2005.*, pages 1859–1864 vol. 3, 2005.

[74] Alastair Robertson. iovisor/bpftrace: High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace.

[75] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), March 2012.

[76] Reiner Sailer et al. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security symposium*, pages 223–238, 2004.

[77] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[78] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 335–350, 2007.

[79] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

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

[81] Giannetsos Thanassis, Dimitriou Tassos, and Prasad Neeli R. Weaponizing wireless networks: An attack tool for launching attacks against sensor networks. In *Black Hat Europe 2010*, Barcelona, Spain, April 12-15, 2010.

[82] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. ScaRR: Scalable Runtime Remote Attestation for Complex Systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 121–134, 2019.

[83] Jorden Whitefield, Liqun Chen, Thanassis Giannetsos, Steve Schneider, and Helen Treharne. Privacy-enhanced capabilities for vanets using direct anonymous attestation. In *2017 IEEE Vehicular Networking Conference (VNC)*, pages 123–130, 2017.

[84] Jorden Whitefield, Liqun Chen, Thanassis Giannetsos, Steve A. Schneider, and Helen Treharne. Privacy-enhanced capabilities for vanets using direct anonymous attestation. In *VNC*, pages 123–130. IEEE, 2017.

[85] B. Wiedersheim, Z. Ma, F. Kargl, and P. Papadimitratos. Privacy in inter-vehicular networks: Why simple pseudonym change is not enough. In *Seventh International Conference on Wireless On-demand Network Systems and Services (WONS)*, pages 176–183, Feb 2010.

[86] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391. IEEE, 2017.

[87] Jiliang Zhang, Wuqiao Chen, and Yuqi Niu. Deepcheck: A non-intrusive control-flow integrity checking based on deep learning. *CoRR*, abs/1905.01858, 2019.

[88] Yumei Zhang, Xinzhi Liu, Cong Sun, Dongrui Zeng, Gang Tan, Xiao Kan, and Siqi Ma. Recfa: Resilient control-flow attestation. *arXiv preprint arXiv:2110.11603*, 2021.

# ANNEX

## 8.1 Notation used for ASSURED Attestation Schemes

Throughout the protocols described in Chapter 5, we consider the following notation:

| | |
|---|---|
| D | A device. |
| $\mathcal{TC}$ | Trusted Component (e.g., a SW or HW-TPM). |
| **SCB** | The Security Context Broker (trusted authority). |
| $\mathcal{AA}\mathsf{gt}^{\mathcal{X}}$ | Local attestation agent running on D $\mathcal{X}$. |
| $\mathcal{T}\mathsf{rce}(r)$ | Retrieve the binary contents of object identified by $r$ using the secure and immutable tracer $\mathcal{T}\mathsf{rce}$. |
| $\leftarrow, =$ | $\leftarrow$ denotes assignment and $=$ denotes comparison. |
| $h$ | Hash digest ($0 \dots 0$ is used to denote a zero-digest). |
| H | A secure and collision-resistant hash function. |
| $\mathsf{hk}_{(A,B)}$ | Symmetric hash key known only by $A$ and $B$. |
| $\mathsf{HMAC}(\mathsf{hk}, i)$ | hk-keyed Message Authentication Code over $i$. |
| $\mathsf{Eval}(expr)$ | Evaluation function for arbitrary expressions $expr$. |
| $\mathsf{Vf}(expr)$ | Verification, which interrupts if $\mathsf{Eval}(expr) = 0$. |
| $\mathsf{Sign}(m, \mathsf{k})$ | Computes a signature over $m$ using k. |
| $\mathsf{Sig}^{\mathsf{k}}_{\phi}$ | Signature over $\phi$ using key k. |
| $\mathcal{H}$ | TPM handle, where $\mathcal{H} \in \mathbb{N}$. |
| $\mathsf{TPL}(\phi)$ | Template for object $\phi$ (including its attributes). |
| $\mathcal{B}$ | Boolean variable: $\mathcal{B} \in \mathbb{B} = \{0, 1\} = \{\mathsf{false}, \mathsf{true}\}$. |
| $\mathsf{name}(\phi)$ | $\phi$'s name. For keys and NV indices, it is a digest over the public area, including attributes and policy. |
| $CC_{\mathsf{cmd}}$ | cmd's TPM Command Code. |
| $RC(\mathsf{Eval}(\mathsf{cmd}))$ | TPM Response Code after executing cmd. |
| $mPCR^{\mathsf{D}}$ | Set of mock PCR tuples: $\{\langle idx_0, h_0\rangle, \dots, \langle idx_n, h_n, \rangle\}$ associated with D, where $idx_i \in \mathbb{N}_0$. |
| $mNVPCR^{\mathsf{D}}$ | Set of mock NV PCR tuples: $\{\langle \mathcal{H}_0, h_0, \mathsf{name}(\mathcal{H}_0)\ \rangle, \dots, \langle \mathcal{H}_n, h_n, \mathsf{name}(\mathcal{H}_n)\rangle\}$ associated with D. |
| $PCRS$ | Set of PCR selectors: $\{i : i \in \mathbb{N}_0\}$. |
| $NVPCRS$ | Set of NV PCR selector tuples: $\{\langle \mathcal{H}_0, h_0\rangle, \dots, \langle\ \mathcal{H}_n, h_n\rangle\}$. |
| $PPS$ | A TPM's *secret* Platform Primary Seed. |
| $\mathsf{proof}(\phi)$ | A TPM's *secret* value associated $\phi$'s hierarchy. |
| $SK$ | Restricted storage (decryption) key. |
| $EK^{\mathcal{O}}_p$ | $\mathcal{O}$'s endorsement (restricted signing) key pair: $\langle\ EK_{\mathsf{pk}}, EK_{\mathsf{sk}}\rangle$, where $EK_{\mathsf{sk}}$ is encrypted, denoted $\mathsf{sealed}(EK_{\mathsf{sk}})$, while outside the TPM. Optionally, $p$ is used to refer to a specific part of the EK. |
| $AK^{\mathcal{O}}_p$ | $\mathcal{O}$'s attestation (unrestricted signing) key pair: $\langle AK_{\mathsf{pk}}, AK_{\mathsf{sk}}\rangle$, where $AK_{\mathsf{sk}}$ is encrypted, denoted $\mathsf{sealed}(AK_{\mathsf{sk}})$, while outside the TPM. Optionally, $p$ is used to refer to a specific part of the AK. |

## 8.2 Revocation Sequence Diagrams

Figure 8.1: Activate Revocation Index