CONNECT

CCAM TRUST & RESILIENCE

D4.2:Virtualization- and Edge-based Security and Trust Extensions

Project number:	101069688
Project acronym:	**CONNECT**
Project title:	Continuous and Efficient Cooperative Trust Management for Resilient CCAM
Project Start Date:	1st September, 2022
Duration:	36 months
Programme:	HORIZON-CL5-2021-D6-01-04
Deliverable Type:	R – Report
Reference Number:	D6-01-04 / D4.2 / 1.02 April 4, 2024
Workpackage:	WP 4
Due Date:	February 29, 2024
Actual Submission Date:	March 11, 2024
Responsible Organisation:	INTEL
Editor:	Matthias Schunter
Dissemination Level:	PU – Public
Revision:	1.02 April 4, 2024
Abstract:	Deliverable D4.2 documents the high-level design of the complete *CONNECT* security architecture as developed by Workpackage WP4. Compared to D4.1 we refine and add users stories and designs for the Multi-Access Edge Cloud (MEC). We also add detailed cryptographic protocols that implement the trust assessment of the services of *CONNECT*. Finally, we detail the concepts and interactions for secure migration and upgrade of TEE-protected workloads. This is the technical foundation for Secure Task Offloading. The key questions that this deliverable answers are "How can the Multi-access Edge Cloud (MEC) be architected to provide robust security and privacy guarantees?", "How can cryptography help to remotely assess trust of services while maintaining sound privacy guarantees?", and "How can vehicle and edge collaborate while maintaining security?".
Keywords:	Trusted Execution Environment, TEE, Multi-Access Edge Cloud, MEC, TEEGuard, CCAM, Security Architecture

Funded by the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or CINEA. Neither the European Union nor the granting authority can be held responsible for them.

Editor
Matthias Schunter(INTEL)

Contributors (ordered according to beneficiary numbers)

Anna Angelogianni, Nikos Fotos, Stefanos Vasileiadis, Thanassis Giannetsos,
Vasilis Kalos (UBITECH)
Panagiotis Pantazopoulos, Pavlos Basaras (ICCS)
Dimitrios Stavrakakis, Dmitrii Kuvaiskii, Matthias Schunter, Sergej Schumilo (INTEL)
Christopher Newton (SURREY)

Executive Summary

The current deliverable highlights the advances regarding the **security and trust extensions** proposed by *CONNECT*, aimed at facilitating the collaborative trust assessment vision; hence, achieving the overall goals of the project. Notably, D4.2 builds upon its predecessor to enhance the functionalities of the overarching **CONNECT Trusted Execution Architecture**, extending its applicability to cover the Multi-Access Edge Computing (MEC) side. MEC plays a crucial role for extending the stand-alone vehicle domain to safe and security solutions distributed from the Vehicle (far-edge) to Edge and Cloud facilities as an enabler of highly automated driving functions. Nevertheless, the addition of MEC in the CCAM continuum may introduce new attack vectors; hence, advanced protection mechanisms are needed.

This also materialises the vision of **task offloading** entangling the operational planes of the entire CCAM continuum. The concept of task offloading holds a significant role in the CONNECT vision. It involves the shift of tasks from the vehicle(s) to the infrastructure where the computational resources are typically richer. Subsequently, the result of the task execution may need to be returned to the original vehicle (and/or neighbouring vehicles), or, trigger other actions (such as the communication to a cloud CCAM server). Besides adequate orchestration capabilities for managing the communication of a high influx of data, that might be needed for supporting the task offloading process, this also requires a high degree of network function security: **Tasks should be offloaded to target nodes that can exhibit the required level of trust for protecting its internal operations.** Such security assurances can then directly translate into trust to the performed (offloaded) computations so that they can be consumed by the Vehicle.

In order to extend this protection to cover the MEC infrastructure where CCAM services are deployed, *CONNECT* builds on top of a trust anchor as established for the far edge. This extension facilitates the secure deployment and execution of services at the MEC level, leveraging a Secure Element. For the purposes of *CONNECT* demonstrators, the adopted technology is Gramine, which is based on Intel SGX, while for the launching of secure enclave, enclave-cc technology is utilised. *It shall be underlined, though, that* CONNECT *maintains a high degree of interoperability and agility with any type of Root-of-Trust that demonstrates the baseline of Secure Storage, Measurement and Reporting capabilities [12].*

Following this expansion to the MEC, the functional specifications, as it pertains to the Trusted Execution Architecture, are enriched to support trust assessment and trust quantification in the context of a more perplexed environment, constituted by services deployed both at the far edge and the edge side. The deliverable outlines the lifecycle of containerised services, providing in-depth information on secure launching, deployment, upgrade and migration processes, further capturing the internal operations of the Trusted Computing Base.

Furthermore, this deliverable elaborates on the cryptographic primitives, adopted by *CONNECT* and designed to specifically accommodate the needs of the CCAM landscape at the far edge side. These needs are consistent with the existence of different types of ECUs possessing varying cryptographic capabilities and trustworthiness evidence stemming from multiple trust sources. Additionally, it addresses privacy-preserving concerns when sharing this evidence with other vehicles or the MEC. The respective protocols covering the MEC side will be elaborated in D4.3 [18].

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Overview

In the first round of activities revolving around the design of "*secure chip-to-cloud*" solutions, enabling the trust assessment and quantification of CCAM ecosystems (documented in D4.1 [10]), we focused primarily on the vehicles (i.e. the far-edge). There we presented an initial description of the *CONNECT Trusted Computing Base (TCB)*, and the *CONNECT* Trusted Execution Architecture along with the necessary building blocks and their functionalities towards the secure lifecycle management of all in-vehicle (HW and SW) elements comprising an automotive service. At the core of this architecture, is the establishment of a chain-of-trust throughout the entire service stack of the host vehicle: from the device hardware, to the application and execution environment been instantiated in the Vehicle Computer. A trust pillar in this context is the anchoring of all services to a Root-of-Trust (instantiated in resource-capable in-vehicle elements, e.g., Gramine TEE) capable of expressing trust in a verifiable manner through the provision of **security/trustworthiness claims**. These claims include assertions on the integrity (and other trust properties of interest including resilience, robustness, safety, etc., as defined in D3.1 [9]) of the target device and its correct configuration and execution state to be used as a trust source for inferring (and reasoning on) the Actual Trust Level (ATL) of any in-vehicle node and data item.

Building on this initial trust execution architecture, the objective of the present deliverable is to describe how the *CONNECT TCB* extends to also capture the requirements of the *Multi-access Edge Computing (MEC)* layer of the CCAM ecosystem, by demonstrating the necessary refinements that have to be added to the *CONNECT TCB*. As mentioned in D2.1 [12], **the MEC introduces several advantages by providing vehicles with seamless access to significant computational capabilities**, addressing cases where the vehicle's resources are insufficient. This enhancement of the operational landscape translates into deploying services at both sides of the spectrum (i.e., far-edge and edge) and/or **offloading/migrating resource-intensive tasks from the vehicle to the MEC** so as to benefit from the rich (edge-deployed) computational resources. The motivation to employ task-offloading operations is the increased in-vehicle computation needs and the ever-increasing complexity/computational requirements of the CCAM applications. In view of the higher automation level of connected vehicles (mainly with the inclusion of advanced ML-based, cooperative perception and positioning capabilities [37]), further computational needs are posed. Even overcoming the proliferation (rate) of capable in-vehicle devices/hardware [31]. *Nevertheless, it should be clarified that this updated operational model, incorporating the MEC, may further introduce new attack vectors. Such attacks vectors are elaborated in the 5GAA paper [2] where the CONNECT consortium provided detailed definitions of threat models as well as the security boundaries identified for the MEC system and the services that the MEC hosts for supporting the operation of connected vehicles.*

As a result, sufficient **runtime security controls are needed, to assess and establish trust, not only among vehicles but also to the virtualised infrastructure** where the services are deployed; i.e., where the CCAM- or *CONNECT*- related trust services are instantiated. The **integration of a *Trusted Execution Environment (TEE)* into the *MEC* enables all operations to be protected, including the continuous and evidence-based trust assessment of any CCAM (SW and HW) element.** This integration of trusted and confidential computing provides strong integrity guarantees on the infrastructure where services are running and containerised services have been deployed; hence, trust assessment is also extended to consider the trust state of elements deployed across the entire continuum - from the far edge (Vehicle) to the edge (*MEC*) and the Cloud. Notably, the CONNECT trust execution technology realised on the MEC environment is essentially the same to (subsequently) be used to safeguard the (CCAM) cloud-based services.

This document delves into the essential refinements to the *TCB* for also enabling these additional trust extensions to MEC- and Cloud- deployment environment: These enhancements facilitate the implementation of security controls, **ensuring the secure execution of applications through the trust assessment of virtualised infrastructure where application servers are deployed**. Emphasising the significance of vehicle-to-everything trust assessment, encompassing the evaluation of the virtualised infrastructure beyond the boundaries of a vehicle, as defined in D3.2 [15], remains a critical aspect.

The distinguishing factor in ensuring the integrity of services within a virtualised infrastructure, as opposed to the case when executed in a vehicle, lies in the ability to **verify the integrity of the entire stack of the virtualised infrastructure**. As detailed in Chapters 2 and 3, *CONNECT* primarily utilises Kubernetes as a prominent orchestration technology for demonstration purposes [13], while maintaining interoperability agnostic to the technology employed for service orchestration. Towards this direction, D4.2 showcases the **deployment of legacy containers using Kubernetes, subsequently converting them to confidential containers** (according to their security requirements) with a focus on ensuring the **integrity of the entire process and the underlying networking stack**. This also constitutes another core innovation of CONNECT as it extends the ETSI standardised Levels of Assurance (LoA) classification model [24], for virtualised infrastructures, to capture the intricacies of MEC environments for supporting the life-cycle of connected vehicles. This enriched classification allows for a more granular scale of trust that can be assessed based on the provided security claims by the underlying *CONNECT TCB*.

The *MEC* facilitates the deployment of various workloads, including CCAM services and other application workloads. These workloads can operate in a non-secure mode or be converted into secure enclaves based on their requirements. They include containerised services, some of which are instantiated on the *MEC*, while others are deployed on the vehicle. Additionally, auxiliary functions within the vehicle may process kinematic or perception data for these services. **To automate and streamline the deployment process**, as outlined in Chapters 2 and 3, **the deployment of a containerised service, whether on the *MEC*, Cloud or the Vehicle, follows a common approach. The key requirement** is that the infrastructure (i.e, the In-Vehicle computer, the *MEC* or even the Cloud) **must be equipped with a Root of Trust (RoT), enriched with the *CONNECT TCB*.** This operational model spans across the whole compute continuum, (from the far-edge to the edge and even up to cloud-based services), adopting a unified deployment process for transparency and continuum-wide automation.

It should be clarified that even though in the rest of the document we refer to specific technologies employed for the orchestration, launching and support of both pre-existing legacy as well as newly designed TEE-enhanced containers, that are based on a specific Root of Trust (RoT) (i.e.,

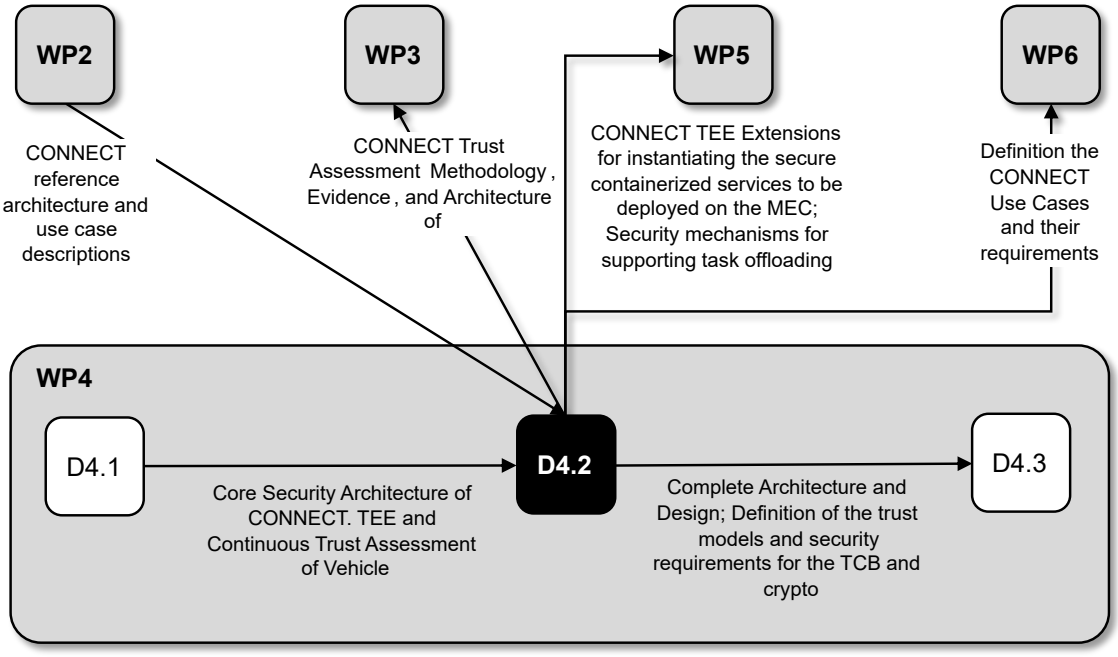

Figure 1.1: Relation of D4.2 with other Workpackages and Deliverables.

Intel SGX) and a TEE technology (i.e., Gramine), the **CONNECT *protocol designs prioritise technology neutrality***. *This ensures compatibility with other CoCo technologies and RoTs, as long as they follow the Root of Trust for Storage, Reporting, and Measurement principles [12].*

1.1 Relationship with other Workpackages & Deliverables

The documentation of the detailed overarching Trust Execution Architecture is a core building block for supporting the first-of-its-kind *CONNECT* Trust Assessment Framework [9, 15] capturing the trust model and complex trust relationships that need to be continuously assessed towards the materialisation of a trusted CCAM continuum. This enables the transformation of (zero-trust) CCAM service networks into “trustful service unions” featuring not only strong (end-to-end) security but also other core trust properties including integrity, resilience, robustness, safety, etc. This is the core focus of Deliverable D4.2 putting forth the detailed design of the first version of *CONNECT* trust extensions for enabling the safeguarding of all core in-vehicle components comprising a CCAM service function chain.

Figure 1.1 depicts the direct and indirect relationships of D4.2 to the other Tasks and Work Packages (WPs). WP2 provides the high-level requirements and overall architecture as inputs. As aforementioned, WP4 then refines and extends this high-level architecture with a focus on security. In D4.1 we concentrated upon Hardware-based Trusted Execution and continuous trust assessment for the vehicle. Now, in D4.2, we document the complete security architecture of *CONNECT*. Compared to D4.1, this adds users stories and designs for the MEC. It also adds the detailed cryptographic protocols that are required for trust assessment of the services of *CONNECT*. Finally, it details the concepts for secure migration and upgrade of TEE-protected workloads, which are the technical foundation of `Task Offloading`.

More specifically, D4.2 provides the detailed description of a new set of trusted computing capabilities towards the runtime integrity verification of a component (such as a *electronic control unit (ECU)*) through abstractions, called *policy-restricted attestation keys* (Section 5.4.2.2), that allow for the verification of the component's integrity correctness in a zero-knowledge manner; i.e., without disclosing any details on the actual configuration or operational profile of the target device. This functionality, together with those libraries offered for secure software upgrade and/or state migration (as reaction policies to the change of a node's trust level), comprise *CONNECT*'s *TCB* capabilities for supporting the operations of the Trust Assessment Framework: *All attestation attributes constitute one of the trustworthiness sources based on which the TAF (as the core WP3 artefact) calculates the Actual Trust Level (ATL) of a (SW and/or HW) element which leads to the trust state characterisation of the target node*. Thus, D4.2 provides one of the core building blocks for supporting **context-aware continuous trust assessment in (zero-trust) CCAM.**

Furthermore, this deliverable sets the scene for the other core innovation of *CONNECT* towards extending the standalone vehicle domain to safe and security solutions distributed from Vehicles to MEC and Cloud facilities. Towards this direction, D4.2 provides the final and refined description of *CONNECT*'s overarching Trusted Computing Base considering also the provision of the necessary mechanisms for ensuring the **secure life-cycle management of any CCAM (and beyond) service deployed on the MEC.** This does not only include the runtime attestation of the service itself (enacting upon the previously mentioned attestation procedures) but also captures the integrity and resilience of all the MEC infrastructural elements as well as the SW components running on them, and especially those facilitating the network orchestration plane. Attestation of a virtualised infrastructure element (i.e., measurement and verification) leads to the ability to establish information security assurance. **This, in turn, constitutes another core trust source based on which the vehicle can assess the level of trust it can put on the MEC to protect its information and functional assets.** All these capabilities set the scene for the detailed experimentation and evaluation activities of all these trust extensions in the context of the envisioned use cases on Intersection Movement Assistance, Collaborative Cruise Control and Slow-Moving Traffic Detection (WP6).

This also materialises the vision of **task offloading** entangling the operational planes of the entire CCAM continuum - from the Vehicle to the MEC and/or Cloud. The concept of task offloading (see D5.1 [11] for definition and D5.2 [13] for a detailed description of *CONNECT* networking and offloading capabilities) holds a significant role in the CONNECT vision. It involves the shift of tasks from the vehicle(s) to the infrastructure where the computational resources are typically richer. Subsequently, the result of the task execution may need to be returned to the original vehicle (and/or neighbouring vehicles), or, trigger other actions (such as the communication to a cloud CCAM server). Besides adequate orchestration capabilities for managing the communication of a high influx of data, that might be needed for supporting the task offloading process, this also requires a high degree of network function security: Tasks should be offloaded to target nodes that can exhibit the required level of trust for protecting its internal operations. Such security assurances can then directly translate into trust to the performed (offloaded) computations so that they can be consumed by the Vehicle. Such strict security assurances are based on the attestation mechanisms that are presented in this deliverable and will be further extended in D4.3 [18] for MEC deployment environments.

1.2 Scope and Purpose

As aforementioned, the focus of this deliverable is twofold: (i) **Extend the overarching *CONNECT* Trusted Execution Architecture to also account for the security and trust requirements of MEC deployments** - characterised by the presence of edge computing infrastructure hosting multiple MEC tenant application providers and application users. This culminates to the description of the final version of *CONNECT*'s *TCB* equipped with all trust extensions for allowing the management of security claims as verifiable assertions on the trust state of any (SW and/or HW) element deployed across the entire CCAM continuum, and (ii) Early design of the novel *CONNECT* trust enablers and crypto primitives for supporting the in-vehicle operations and trust calculations (based on the secure and privacy-preserving sharing of trust-related information and evidence). It further sheds light to the secure container life-cycle management at the far-edge, edge and cloud continuum, as well as the cryptographic protocols, tailored to accommodate the needs of the far-edge (i.e., vehicle) side. More specifically, the proposed designs enable the evidence-based trust assessment, leveraging novel attestation mechanisms, including a new variant of **anonymised threshold signatures** for the sharing of verifiable trustworthiness evidence in a zero-knowledge manner. This scheme provides privacy-preserving exchange of trustworthiness claims between the vehicle and other vehicles (in the vicinity) or the *MEC*. Furthermore, the Configuration Integrity Verification (CIV) scheme is elaborated, equipped with Key Restriction Usage Policies, that preserve integrity even in the in the case of a compromised (vehicle or MEC container) host. In the final Deliverable D4.3, the consortium will document the implementation and validation of these key components of the architecture and will proceed with the documentation of the equivalent components as part of the secure MEC deployments.

1.3 Deliverable Structure

The remainder of this deliverable is structured in the following chapters:

Chapter 2 documents the revised and refined *CONNECT* trusted execution architecture. We start by recapitulating the architecture of the core Trusted Computing Base (TCB) and building blocks from D4.1. We then describe our new security architecture for container-based cloud services on the *MEC*. Finally, we outline high-level information flows for the *MEC*.

In **Chapter 3** we document the new design for secure and TEE-enhanced containers. Our goal is to provide TEE-enhanced containers that can be seamlessly managed by Kubernetes while maintaining security throughout their life-cycle. A specific challenge is to ensure proper protection of security-critical state while a container is hibernated and then restarted at a later time.

Chapter 4 specifies revised and new "user stories" that specify key behaviours of our architecture. They document the requirements of different roles / user groups of the *CONNECT* Project. Each user story outlines well-defined usages by given user roles / groups together with their security and functional requirements. This serves as a high-level description of requirements that need to be satisfied by the services and architecture that is provided by *CONNECT*.

In **Chapter 5** we provide details on the cryptographic protocols that allow privacy-enhanced (remote) trust verification of services.

In **Chapter 6** we document design details on the *MEC*. This includes enhancements to the Gramine library OS and additional services and protocols that are provided by the edge.

In **Chapter 7** we conclude our findings and provide an outlook on the final deliverable D4.3 of WP4.

We conclude the document with two appendices that provide additional design details on Intel SGX (Appendix A.1) and the Gramine Library OS (Appendix A.2), a glossary, and a bibliography.

Chapter 2

Extending the *CONNECT* Trusted Execution Architecture to the MEC

This chapter presents the overarching *CONNECT* Trusted Execution Architecture including also the extensions added for capturing the required security and trust models that arise from the integration of MEC deployments. Therefore, in what follows we delve into the architectural details of *CONNECT*'s TCB and its *TEE Guard Security Extensions (TEE-GSE)*, focusing primarily on the MEC aspect. In this context, *Confidential Containers (CoCo)* (as the adopted security technology) offer the trust assurances, leveraging solutions such as the *TEE*, to **prevent unauthorised access or modification of applications and data while in use, thus, protecting against the tampering of the infrastructure where the container is instantiated**. This concept is specifically useful when it comes to security-related and safety-critical applications, such as the ones considered in *CONNECT*, being executed at the MEC-level. Furthermore, it entails **the runtime tracing capabilities for monitoring of a various system properties throughout the entire automotive service software stack**; i.e., from the host vehicle to the measurement of the system when instantiating service containers and network functions on the MEC, determining the integrity and origin of the deployed software. The provision of such granular tracing mechanisms enable the secure state migration of an application from one device to another, as well as the secure upgrade, as a reaction strategy to a change at the trust level of the stakeholder, as assessed by the *Trust Assessment Framework (TAF)*.

Verifying the integrity of the MEC infrastructure where services are deployed is important specifically in the case where the infrastructure may be operated by different MNOs, with varying levels of assurances. These assurances are provided during service execution, reflecting the infrastructure capabilities based on the implemented security controls. **The security controls determine the trust level of a particular MEC Infrastructure, defining the trust level of this domain (i.e., trust domain).** Hence, the extraction of the MEC trustworthiness evidence contributes to the assignment of a trust domain. *An interesting research question beyond the scope of* CONNECT *is the establishment of a trust relationship between two domains, possessing different trust levels. In the context of a vehicle, this challenge occurs when there is a need for seamless transition of a CCAM service provided by MEC provider A as the vehicle moves to MEC provider B, commonly known as handover. This raises questions regarding the seamless preservation and transfer of trust among MEC providers with different levels of trust.*

The following chapters will dive into the **security mechanisms offered by CONNECT and supported by the *TEE-GSE* to guarantee the secure management of containers** (i.e., including containers and confidential containers) **throughout their lifecycle**. This involves creating

fundamental cryptographic building blocks to protect the security and privacy of trustworthiness evidence, depending on the specific requirements. Regarding the privacy requirements, as applicable for the far-edge side, Chapter 5 of the present deliverable, details on the protocols created to meet the functional specifications, as outlined in D4.1[10], regarding the harmonisation of trustworthiness evidence when it exits the vehicle.

All these operations are supported by *CONNECT*'s two main trust anchors:

- ***CONNECT TCB*:** This constitutes the palette of security mechanisms and auxiliary processes that are required by all components offering the envisioned secure life-cycle management schemes. They are those hardware-, software- and firmware-based services, that are by default trusted, the combination of which is used towards the enforcement of a security policy. **In comparison to the architecture outlined in D4.1[10], the final view of the TCB incorporates additional components**, namely the Verifiable Policy Enforcer and the Migration Service. These components are defined in the following paragraphs.

  Part of *CONNECT*'s *TCB* (as will be detailed in Section 2.1) comprises the: (i) **tracing capabilities**, based on the newly developed TEE Device Interfaces (TDIs), with dedicated APIs to be exposed for the continuous monitoring and fetching of evidence and Trustworthiness Claims respectively, used in the context of a trust model for a specific set of trust-properties (i.e., integrity, resilience, etc.), (ii) **key management** module for setting up and governing the use of all application- and security-related keys that need to be setup during the secure deployment of a container and its service(s) (see Section 2.3.1), and (iii) **key usage restriction policy engine** for managing the construction/deployment and enforcement of the necessary policies, binding the use of keys to the expected state of the enclave, (iv) **verifiable policy enforcer (VPE)** for verifying that the latest-deployed the key usage restriction policy is associated to the correct (static) properties of the target application; thus, past policies associated with the same target application are being characterised as obsolete to avoid attacks based on past versions, and the (v) **migration service** for mediating the migration process of an enclave from one hot to another and validating the authenticity of the migrated data (see Section 2.3.3).

  The *TCB* must always behave as expected, otherwise, there is a risk of compromising the overall security posture of the target system, including the main trust assessment mechanism offered by *CONNECT*, which will lead to invalid trust calculation. It is assumed as trusted by default, with the support of the underlying *RoT*. As described in [10], **the end-most goal is to minimise the *TCB*** so as to be able to to meet the requirements while functioning in zero-trust environments. However, the incorporation of the MEC into the broader CCAM continuum necessitates a more dynamic and modular TCB. This means that the size of the TCB should be adjustable based on the service requirements and the underlying infrastructure. For instance, the instantiation of the VPE might be required in multiple instances when relevant to MEC operations, whereas a single instance may suffice for the far-edge. This dynamicity allows for a flexible adaptation of the TCB to varying operational contexts (i.e., different TCB specification per service). More detailed information about the TCB can be found in Section 2.1.

- ***CONNECT* TEE Guard Extensions (*TEE-GSE*):** The TEE-GSE provide isolation at an application-level, leveraging secure enclaves. In the MEC the TEE-GSE are deployed within the confidential containers, offering the necessary security guarantees; thus providing the functionality of secure life-cycle management processes. Following the notion of the confidential containers, the TEE-GSE are built on top of the *TCB* (each container will

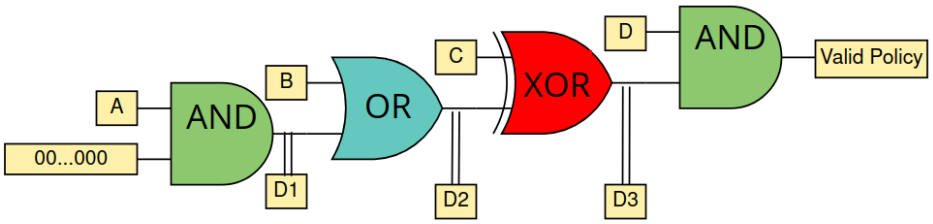

Figure 2.1: Key Restriction Usage Policy Engine.

be equipped with its own *CONNECT TCB*), enabling continuous interaction with the trust anchor, so as to then be able to provide the necessary evidence so that all requesting components will be able to perform the operations described in the section 2.4. More detailed information regarding the TEE-GSE can be found in Section 2.2.

Recall that while CONNECT *protocols have been instantiated over the Gramine technology, they remain agnostic to the type of secure element used as long as the baseline of characteristics for a Root-of-Trust are offered: Root of Trust for Storage, Reporting, and Measurement [12]. Likewise, even though for the purposes of our demonstration activities enclave-CC is leveraged, as it is based on the same type of TEE technology and leverages Gramine, the protocol designs for the MEC are agnostic to the employed technology and are also applicable to other CoCo technologies such as ENARX. In the same line of generality of* CONNECT *schemes, the same notion further applies for the adopted orchestration technologies. Even though Kubernetes is leveraged due to its prominence,* CONNECT *remains agnostic to the specific technology and may operate in tandem with any orchestration framework.*

2.1 *CONNECT* Trusted Computing Base & Building Blocks

The *CONNECT TCB* serves as the foundation for the execution of critical tasks as it pertains to security-related functions and trust assessment. Considering also the design choices (i.e., minimised overhead), the *TCB* comprises the minimal set of components whose correct operation is a prerequisite for supporting the security of the overall system. The following paragraphs delve into the exact building blocks that form the *CONNECT TCB* along with their functionalities. It is worth mentioning, as it was elaborated on D4.1[10], that the *CONNECT TCB* may refer both to the far-edge, and the edge, ranging from the *ECUs* all the way up to the services that are instantiated on the *MEC*. It is important to highlight that the type of *TCB* that is instantiated is resource-dependant. As a result, there is a distinction between different *ECUs* (i.e., *A-ECUs*, *S-ECUs*) and the *TCB* that resides on the *MEC*, based on their cryptographic capabilities. The following descriptions are mainly focusing on the *A-ECUs* and the *MEC*-based *TCB*, that have the capacity to support such operations. For the case of *S-ECU*s that lack the resources, the *TCB* is restricted solely to the secure storage of the employed cryptographic keys.

- **Key Management System:** It is an integral component of the *CONNECT TCB*, responsible for the secure generation, storage and management of cryptographic keys. In *CONNECT*

the security requirements are not restricted to the creation and secure storage of the secret cryptographic keys, but further cover the compilation of the chosen cryptographic algorithms during the execution of an attestation task, as requested by the Trust Assessment Framework (*TAF*).

- **Key Restriction Usage Policy Engine (KRPE):** Enables the vision and core innovation of *CONNECT* towards local attestation, where a Prover can attest to the integrity of its configuration and behavioural state to a Verifier, without the need to disclose any of its implementation details. This is facilitated through the use of **policy-restricted attestation keys** that can be used for producing signed attestation attributes only when a node's conformance can be verified by the local Attestation Agent. The Key Restriction Usage Policy Engine (KRPE), a proposed TCB component in *CONNECT*, supports a collection of logical equations constructed from these assertions. Figure 2.1 presents a logical equation, where D1, D2, D3 and Valid Policy are the hash digests of the inputs of the logical ports. The *Key Usage Restriction Policy Engine (KRPE)* processes these hash digests of inputs through logical ports to determine the validity of a live Key Restriction Usage Policy. To seamlessly integrate the KRPE into *CONNECT*'s TCB, it is conceived as a child process spawned by the device's Key Manager. This arrangement ensures that the KRPE operates entirely within the Trusted World (enclave), upholding communication integrity between the Key Manager, which requests key authorisation, and the KRPE itself. Furthermore, the *CONNECT* TCB needs to be configured in a way that strictly allows entity-creators or administrators to define the set of actions that can be performed before an action is labelled as "completed".

- **(Attestation) Tracer:** Another core component of *CONNECT*'s TCB is the Tracer (Figure 2.2). The Tracer comprises two parts; the first operating in the *untrusted/normal* world, inspecting safety-critical software components, while the other part runs in the *trusted* world, where the Tracer's secret key is stored. Regarding the part that is executed in the untrusted world, the Tracer is responsible for continuously fetching new traces by monitoring processes and routines that are executed within the *untrusted* world of each container/device. Its primary scope is the collection of essential information for attestation methods employed in *CONNECT*, for ensuring integrity. The tracer, in essence, is capturing hashes of configuration properties from safety-critical untrusted processes and routines.

  This monitoring in the untrusted world is not considered part of the TCB. Nevertheless, there is a part of the Tracer's execution that is executed in the trusted world, within the TCB. This part entails the cryptographic-related operations, and more specifically: i) the decoding of the raw security measurements, ii) the calculation of the real-time configuration hash and iii) the generation of the digital signature over the configuration hash based on the secret key. The collected traces are signed by the Tracer in the trusted world and are sent to the Key Manager to perform the required operations.

  CONNECT adopts and builds upon the classification, as defined by ETSI [24], to map the extracted traces to specific assurance levels for the virtualised MEC-based infrastructure, specifcally for the needs of the automotive sector. The ETSI-defined LoA uses numbering from 0-5, to represent a scale of relative trust, where a greater number denotes a higher level of trust. Based on this classification, the following scaling for *CONNECT* is defined:

  - **LoA 0**: denoting the complete absence of any form of integrity verification.
  - **LoA 1**: covering the local integrity verification (i.e., based on signatures) of the hardware and virtualization platform's (hypervisor) during boot and application loading. No

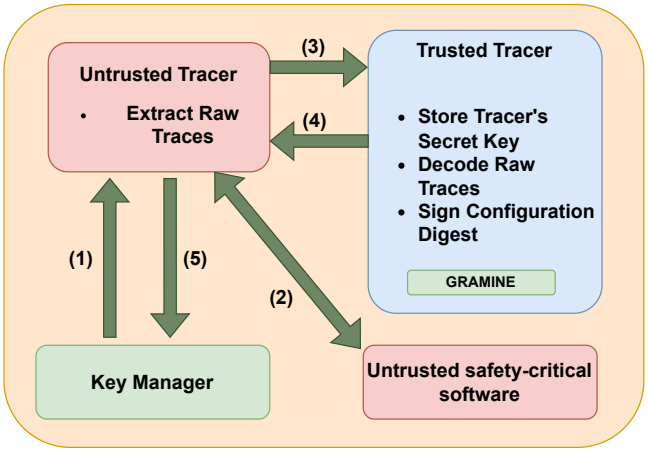

Figure 2.2: Attestation Tracer

proof of integrity is offered. Integrity status is derived from platform state after the end of boot and application load processes.

- **LoA 2**: Adding to LoA 1 the remote attestation of the hardware and virtualization platform integrity. Measurements of boot time and application load time are considered.
- **LoA 3**: Adding to LoA 2, LoA 3 includes the local verification of the Kubernetes orchestrator and API server, comprising the network plane, (i.e., based on signatures) as they are loaded on startup.
- **LoA 4**: Adding to LoA 3 the remote attestation of Kubernetes orchestrator and API server, comprising the network plane. Boot time measurements and application load time measurements should be used.
- **LoA 5**: Adding to LoA 4 the remote verification of the Kubernetes orchestrator and API server (comprising the network place) integrity state during run-time (i.e. post load time).

It has to be noted here, that the Tracer comes with a pre-shared key pair that acts as a Root-ID key of the Tracer. The public part of this Root-ID key is known by the Identity Authentication Management (IAM) component. During the Secure Configuration of all Tracer-enabled devices/components, the IAM sends the public key of the respective Tracer to the Key Manager of each device/component. This process establishes a shared key bound to the underlying hardware RoT, enhancing the security of communication and ensuring the integrity of the Tracer's attestation capabilities.

- **Attestation Agent:** It exposes the Trusted Execution Environment (TEE) Device Interfaces based on the TDISP protocol defined by the IETF standardization working group [25]. These interfaces are responsible for providing the run-time system measurements capturing the current device's configuration and operational state, as obtained from the Tracer, following the Trusted Device Interface for Security Protocols ensuring the integrity of the monitored traces even in the case of a compromised host. The Attestation Agent's role in providing authentic traces and ensuring the secure exchange of these measurements is fundamental to the overall security and trustworthiness of the system. As in CONNECT we are moving towards a **zero-trust** architecture, we want to minimise the trust assumptions to be made to the vehicle's or MEC's deployment environment for hosting *CONNECT TCB*. This essentially pertains to the validation of processes that are executed in the host

which constitutes the "*untrusted*" world of the system's operational environment. To this end, we differentiate between two types of interfaces exposed by the Attestation Agent: the ***Hardened-TDIs*** and ***Softened-TDIs***. Hardened TDIs refer to the composition of interfaces designed for enabling the secure interaction with *CONNECT*'s TCB. Essentially, this is for capturing the consumption of "TEE-assignable" resources meaning functionalities that have the required trust/security capabilities to not be manipulated even by a compromised host. These core functionalities relate to the *key management* (i.e., creation, storage and policy-restricted usage of all application- and security/attestation-related keys) and the monitoring of the integrity and authenticity of the Attestation Agent itself. On the other hand, Softened TDIs are also TEE-I/O interfaces (instantiated in the "*trusted*" world of the system) but for allowing the interaction with "Non-TEE-assignable" processes: These refer to processes that are executed in the host that do not need to possess the required security/trust capabilities offered directly by the TEE (so as to not impact their performance) but have a critical role in the overall system function. Such processes can be related to events for triggering operations including software upgrades, state migrations and/or continuous monitoring of any changes in the trust level of the target system. These untrusted processes mediate the communication of these events to the (trusted) AA for then securely executing the required functionality. Due to their updateable nature, we don't consider them as part of our Trusted Computing Base, thus, it is desirable to employ attestation mechanisms for verifying their integrity as part of the overall Attestation Agent software stack; hence, the introduction of the Verifiable Policy Enforcer (VPE) component (see later for detailed description). Designed protection mechanisms are detailed in Section 5.4

- **Verifiable Policy Enforcer (VPE):** Acts as the entity responsible for validating the current liveliness of the Key Restriction Policy Engine (KRPE) being enforced. Its primary function is to prevent potential attackers from having multiple obsolete policies active. For instance, if a past version of an application is associated with a specific policy with known vulnerabilities, the VPE ensures that this policy is identified and characterised as obsolete, preventing unauthorised or outdated policies from influencing the system's security posture and allowing an attacker to exploit them. Authorised by the *Identity and Authentication Management (IAM)*, the VPE monitors the correctness (i.e., integrity) of the software versions executed by the Tracer and the Attestation Agent (i.e., parts of *TCB*) within the enclave, and if the versions are successfully verified, then the VPE authorises the enforcement of a specific key restriction usage policy. This verification process involves the use of a key obtained (and sealed) from the IAM. If this key is correctly unwrapped and signed by the VPE, then this signifies that the policies are satisfied. More details on the flow of this newly designed protocol can be found in Section 5.4.2.2.

- **Migration Service:** This service offered by a component, instantiated as part of the *TCB* of the device, supports the secure transfer of the state of an application, been launched at an enclave, from a host A to a host B. The term hosts in this case refers to hardware-enabled trusted environments. This process may be instantiated due to the alteration on the trust level of a CCAM service or due to a specific event (i.e., task offloading).

  A specific case of need for migration of a service/task includes the task offloading (i.e., due to limited resources). In the task offloading case, a CCAM application or a CONNECT-related task (i.e., the TAF) may need to be offloaded from the Vehicle to the MEC. The MEC in this case is considered as a "trusted worker". More elaborated descriptions regarding the secure offloading will be discussed in D5.3 [17].

2.2 *CONNECT* TEE Guard Extensions & Building Blocks

The *TEE Guard Security Extensions* (*TEE-GSE*) are comprised by a set of *CONNECT* system components. These components play a crucial role in enabling a CCAM actor to provide runtime evidence of operational characteristics (i.e., configuration, control flow execution, etc.) in a verifiable manner, based on the use of advanced cryptographic primitives. In the initial deliverable (D4.1 [10]), the focus was directed towards defining the *TEE-GSE* specifically for the vehicle context. Nevertheless, it is crucial to emphasise that the trust assumptions in the vehicle setting differ from those pertinent to the MEC infrastructure (i.e., MEC-produced Trust Opinion and MEC Attestation Report).

Notably, while *CONNECT* relies on the use of standardised ETSI PKIs for safeguarding the identity of the vehicle, there are different privacy dimensions, one of which is the possibility of an attacker to fingerprint the vehicle (i.e., by spoofing attestation evidence). *CONNECT* has designed the cryptographic schemes to prevent such fingerprinting. These privacy requirements are not necessarily mirrored in the MEC environment (i.e., depending on the MNO). Therefore, the MEC-based *TEE-GSE* diverges from its counterpart in the vehicle context, as the harmonisation of collected evidence is not a pre-requisite. In the vehicle architecture, this harmonisation task is undertaken by the *Trustworthiness Claims Handler (TCH)* component, a role that is absent in the MEC *TEE-GSE*. Therefore, in the subsequent paragraphs, the MEC-based *TEE-GSE* is defined. Given the absence of the *TCH* role in the MEC, the responsibility for generating the *Verifiable Presentation (VP)* shifts to the *IAM*.

The attestation evidence collected on the MEC side serves a dual purpose, being utilized by both the standalone and federated *TAF*, instantiated within the MEC. These frameworks leverage the attestation evidence as a trustworthiness source during the execution of MEC services. The resulting Trust Opinion, reflecting the integrity status of the MEC infrastructure where specific services are executed, is communicated back to the vehicles. This information is transmitted through ETSI DENM messages, allowing the vehicles to consider the trust level associated with the MEC that generated the data.

- ***IAM***: The IAM component manages the V2X communication key, which consists of PKI-issued pseudonym credentials following ETSI standards. This key is utilized for secure communication with vehicles, the *TAF*, and the *Mis-behaviour Detector (MBD)* components. Additionally, the IAM is tasked with creating the *VP*, from the received *Verifiable Credential (VC)* from the MEC-based-*TAF* and -*Attestation and Integrity Verification (AIV)*, which contains information to be disseminated to the vehicles. This VP discloses attributes needed to the vehicles per the service.

- ***AIV***: The *AIV* manages the attestation process of all of the assets comprising a service graph, either been instantiated on the MEC or the vehicle. As mentioned in Section 2.1, in *CONNECT* we leverage the five LoA, as defined by ETSI. The process of evidence collection, which includes attestation evidence (i.e., traces) is instantiated either asynchronously or upon request. Note that there are a number of possible mechanisms for the interaction between the AIV and the TAF and these can be characterised as *pull* where the TAF requests the attestation evidence that it needs, or *push* where the TAF receives initial attestation evidence and is then updated if this evidence changes. Hence the communication can be both synchronous and asynchronous.

  In the first case a vehicle may request to receives the MEC updates as it pertains to modification on the trust level of a specific service. On the second case, upon request, and based

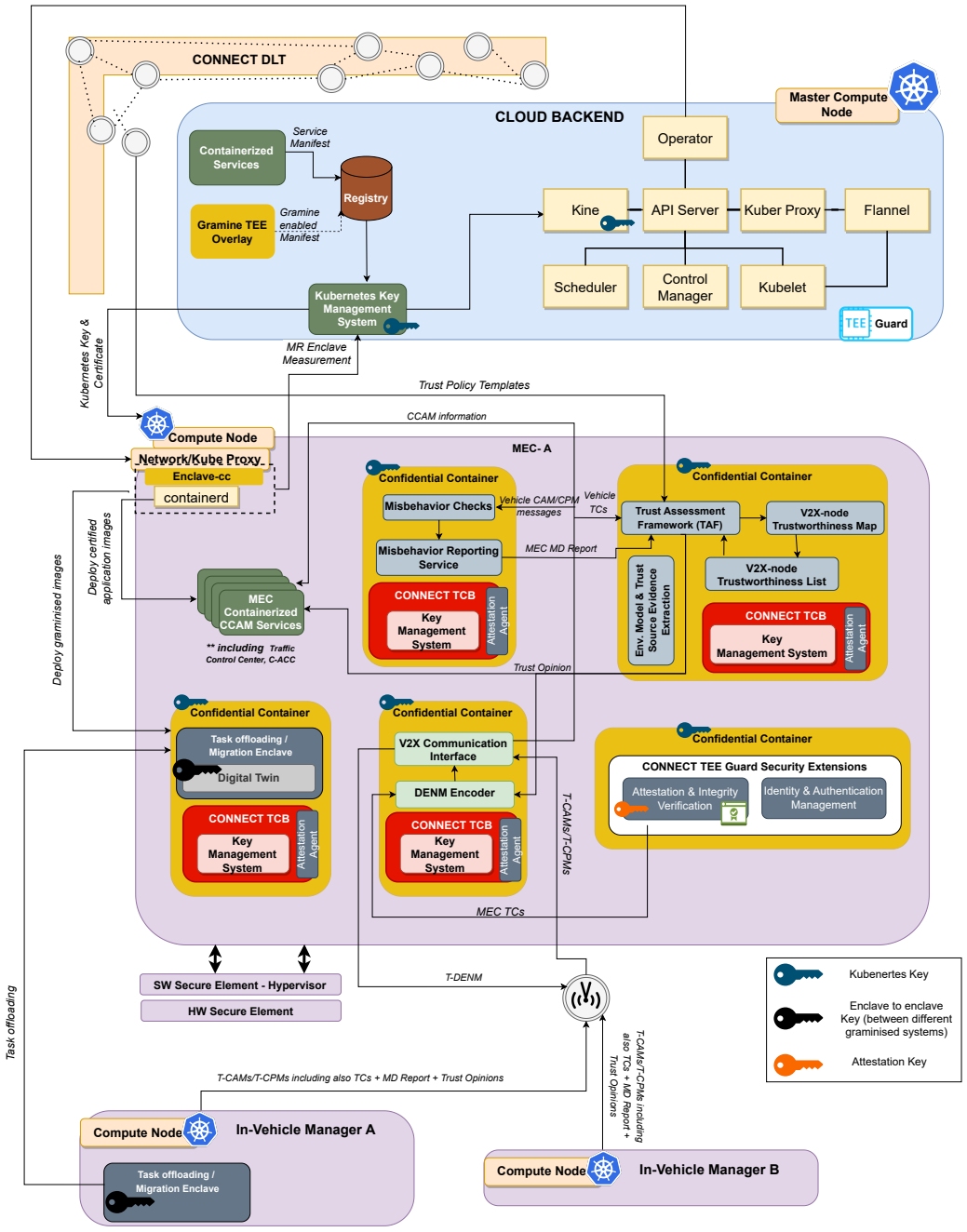

Figure 2.3: MEC High-Level Architecture

on the time constraints, a new or an older attestation report may be leveraged. This evidence is consumed both by the MEC-based *TAF* and the standalone *TAF* residing on the vehicle(s), that leverages the attestation report to calculate the *Actual Trust Level (ATL)*. It should be clarified that the evidence leveraged at the MEC-based *TAF* and the standalone *TAF* are different.

2.3 The *CONNECT* Trusted Execution Architecture

The overall Trusted Execution Architecture is depicted on Figure 2.3. At a high-level, the architecture is structured into four key levels: i) the vehicle, ii) the MEC, iii) the Cloud and iv) the DLT. The fundamental idea revolves around the secure sharing of trustworthiness evidence, that can be considered for assessing the trust level of any data or function during runtime. Based on this secure exchange of evidence, the communicating parties can establish a trust relationship. The *TAF* integrated into the MEC utilises these messages to create more accurate Trust Opinions, leveraging an enriched set of data stemming from multiple vehicles. These Trust Opinions can then be circulated to the vehicles, with the associated trustworthiness evidence on the integrity of the MEC-enabled services.

The MEC is deployed by the Master Compute Node (i.e., Kubernetes) that resides in the cloud side. The latter, has access to the Registry, containing images both for the CCAM services and for the Gramine-enabled *CONNECT* services, as well as the Key Management System, enabling the deployment of the correct image. Note that even if *CONNECT* adopts the Kubernetes orchestration technology for the purposes of the demonstration activities, as a framework it remains agnostic to the actual technology used. Details on the Secure Deployment process are described in Section 2.3.1. The verification takes place over the legacy infrastructure is deployed by the Compute Node to the MEC site. The legacy infrastructure builds the Gramine-enabled image and afterwards the Compute Node of the MEC site sends the MR Enclave Reference Value Measurement to the Master Node of the cloud to perform the verification.

These containers incorporate the enclaves for all services, including the *MBD*, the *TAF*, the *TEE-GSE*, the Digital Twin and the V2X communication interface. Lastly, the DLT is leveraged from the *MEC*'s *TAF* to acquire the Trust Policy Templates (i.e., guidelines for the management of dynamic Trust Models). More information regarding the description and usage of the dynamic Trust Models, acquired by the DLT are available on D3.1 [9] and D3.2 [15] The following paragraphs will delve into detail regarding the exact operations performed by each layer.

2.3.1 Launching Securely Verifiable Confidential Containers

First and foremost in order for the *CONNECT* MEC-based framework to function properly, the MEC containers should be securely launched (see Figure 2.4, which introduces a zoomed-in illustration of the Kubernetes-based security architecture). To achieve this, *CONNECT* leverages Kubernetes technology, specifically K3s. K3s is a fully conformant, production-ready Kubernetes distribution tailored for operational efficiency in unattended, resource-constrained, and remote environments. It ensures the reliable deployment and management of containers, supporting the needs of production workloads. In the present deliverable, the Kine, the Supervisor and the Tunnel Proxy are explained, that facilitate the launching of the MEC-based containers. More information regarding the Kubernetes internal blocks and their functionalities are available in D5.2 [13].

The Kine database system stores cluster state data, Kubernetes resources, worker node status, user roles, roles bindings, and container scheduling and orchestration information, enabling Kubernetes components to perform their functions and storing Kubernetes secrets for authentication. The Tunnel Proxy in K3s ensures secure and efficient communication between control and worker nodes, encrypting data transmission and protecting the cluster from external threats. Lastly, the Supervisor in K3s enhances communication between worker nodes and the server's control plane functions by acting as an intermediary firewall. It establishes a web-socket con-

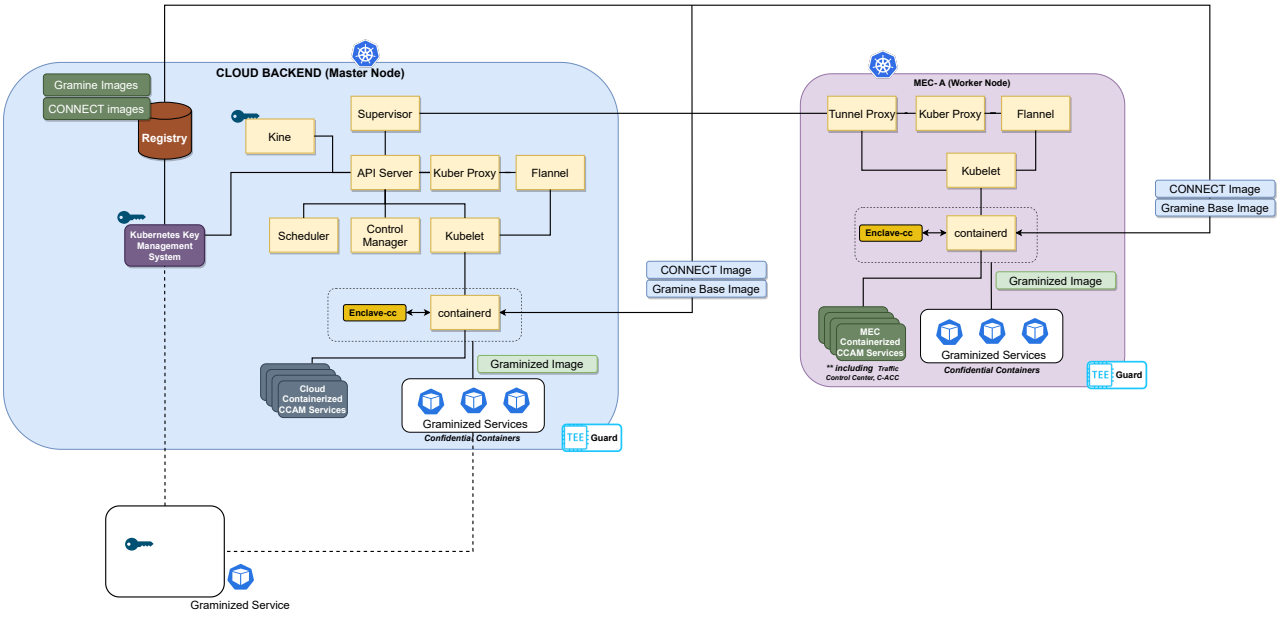

Figure 2.4: Kubernetes-based Container Architecture

nection for registration, connects to the supervisor and kube-apiserver via a load-balancer, and ensures resilient connections during server outages. These components are crucial for launching kubernetes containers.

As previously explained, *CONNECT* aims to follow the *CoCo* paradigm. Enclave-cc is leveraged to launch confidential containers. More details on this are presented in Chapter 6. All containers, including CCAM and *CONNECT* services, are initially deployed through a legacy container. The security added by leveraging enclave-cc and *CoCo* depends on the trust requirements. **These trust requirements are dictated by the equilibrium between safety (i.e., which may be affected by the overhead of such operations) and security; hence are dependant on the offered service and its operational profile. Thus, not all containers conform to identical principles.** *It should be mentioned that for the purposes of* CONNECT *demonstration activities, Kubernetes and enclave-cc is leveraged to launch confidential containers.*

Towards this direction, the Registry residing at the cloud-level comprises two types of docker-based image files (i.e., including their Manifest files), that introduce the: i) the *service manifest* files (i.e., CCAM service docker image) and ii) the *Confidential Computing (CC)-enabled service manifest* files. The first type comprises the standardised docker image information used for the deployment and orchestration of the service, specifying the size, bandwidth, network dependencies of the container. The second type, the CC-enabled image defines the trusted files and what runs isolated within the TEE. *Note that for* CONNECT *the employed TEE is Gramine, which is based on Intel SGX, hence, for the remainder of this document we will refer to the secure conversion of a legacy container with the term "*Gramine-enabled image*".*

Both types of image files, may be deployed at the MEC side. The Gramine-enabled image is built leveraging enclave-cc, which constructs the .sgx variant of the service manifest file including information on the security requirements of specific files, processes, system calls, etc. of the service and the level of isolation they need for their securte execution. **In essence, the enclave-cc builds an overlay of the Gramine-enabled image on top of a regular image file**. This Gramine-enabled image leverages the Intel SGX hardware in order to launch confidential containers, hence the manifest includes relevant information for the deployment over this trusted

hardware.

The Gramine-based Manifest file further includes the MR Enclave Reference Value Measurement, that is essence, a reference value, based on the bootup measurement of the enclave. Furthermore, in addition to the Gramine-based Manifest file, enclave-cc supplies the digest of the application intended for deployment within the container, referred to as the MR Enclave. This digest is signed to generate the .sig file, allowing anyone to verify its authenticity. Kubernetes acts the underlying technology that enables the deployment and orchestration of these manifests at the MEC side.

To ensure that the correct application is being launched, **verification through recalculation of the MR Enclave Reference Value Measurement** also takes place. This verification is performed by the Kubernetes Key Management System, residing at the cloud, which accesses the available information within the Registry. To perform this task, the Kubernetes Key Management System recreates the Manifest file for a given Gramine-based docker image, based on the MR Enclave measurement, and compares the two values; the one that it calculated with the one received by the registry. After the successful verification, the Kubernetes Key Management Service releases the Kubernetes secret key (i.e., used for the secure communication with other containers or with the Master Compute Node) and the certificate. This certificate includes the public part of the Kubernetes public key, to enable the authentication of the container's workload. Whenever a confidential container is launched, the Kubernetes secret key and the certificate need to be retrieved as they dictate the expected state of the container, which is needed for its verification.

2.3.2 Verification of CoCo Workload & Container Binding

Up until now, the secure deployment/launching of a container has been covered. When leveraging the term secure deployment, we also capture the final operation of the integrity check of the launched (confidential) container. This is done by extracting and sharing the MR Enclave Reference Value Measurement that holds the hash of the whitelist of binaries been instantiated as part of the Gramine-enabled container. The verification is performed by the Kubernetes Key Management Service as part of verifying the correctness of the container prior to releasing the kubernetes encryption key further enabling the Establishment of secure and authentication channels with other containerized services and/or the Master Compute Node.

However, while this operation ensures the integrity of the launched containerised service, it does not provide any evidence on the provenance of the container itself. This is required in complex and multi-tenant environments where multiple instances of the same application and/or security services might be instantiated and, thus, enhanced authentication capabilities are required for guaranteeing the communication with the intended service. For instance, consider the case where in a MEC deployment environment, operated by multiple MNOs, there are two *Attestation & Integrity Verification* (AIV) components instantiated in different containers - each for managing the construction of verifiable attestation attributes on the part of the MEC infrastructure managed by each MNO. When these trustworthiness/security claims are communicated to the vehicle towards assessing the trust level of the MEC itself, the vehicle needs to make sure that it receive claims - in a verifiable manner - from the appropriate AIV.

To achieve this, in *CONNECT* we proceed with binding the public part of the container's Kubernetes (identity) key, as part of its certificate issued by the Kubernetes Key Management Service, with the attestation secret/key based on which an attestation attribute is monitored and signed. This allows any external verifier (e.g., vehicle) to not only attest to the

correct state of the container (based on the attestation signature received) but also have the necessary guarantees of the authenticity of the container: The presented certificate is bound to the container's unique Kubernetes key, thus, no one else is able to use it or shoe this certificate without proof of possession of this unique identifier.

More specifically, *CONNECT* suggests associating the attestation key, utilised by the *AIV* for signing evidence, with the certificate. This association signifies that the AIV can construct the Trustworthiness Claims (TCs) only when such a certificate is present, facilitating verification through a key restriction usage policy. This, in turn, ensures that the enclave has been launched correctly. This certificate is also part of the *VC* to be constructed by the AIV further including the Level of Assurance (LoA), based on the attestation evidence being monitored. Consequently, both the certificate and the attestation key act as evidence that the confidential container has been launched correctly (i.e., boot-up integrity); hence, the application running in the enclave is the correct one. More details on this are described in Story-XXX. *Please note that the cryptographic schemes being described in Chapter 5 achieve a LoA 2. In the second version of the* CONNECT *framework the plan is to reach a LoA 4.*

2.3.3 Secure Migration and Task Offloading

In addition to the deployment of containerised services, *CONNECT* further considers the migration of an enclave and task offloading of a certain process from the vehicle computer to the MEC. Both procedures takes place based on the Trust Policies. An illustration of a policy that might prompt task migration is delayed response time. The offloading may include any application or even the *TAF*. In the latter case, the entire trust calculation may be offloaded to the Digital Twin TAF (DT-*TAF*) which resides at the MEC. Nevertheless for these calculations to be performed the vehicle should share its Trust Model along with the Trust Sources as extracted by its Trust Source Manager (i.e., *TAF* and *MBD*). Hence, the MEC acts as a trusted worker to facilitate the vision of both a Federated *TAF* and a DT-*TAF* (more information on this is available in D3.2 [15]).

Furthermore, *CONNECT* supports the secure migration of an entire enclave from a host A to a host B. To achieve integrity of the migratable state *CONNECT* leverages the MR Signer key. This key is a symmetric key, strictly bound to the enclave's underlying trusted hardware (i.e., Intel SGX for the *CONNECT* demonstration), which means that it can be generated strictly on the device that the enclave is running. The MR Signer key derives from the hash of the enclave public key. In parallel, for confidentiality the migration state components being instantiated into two devices interact to establish their own communication key. More information on the details of this scheme can be found in Chapter 6, while th respective user story is presented in Story-XXIX.

2.4 *CONNECT* MEC Information Flows

In the scenario of Intersection Moving Assistance (IMA) with a single vehicle, the vehicle actively gathers kinematic data from neighbouring vehicles and generates additional data through its own sensors. Utilising this data, the vehicle conducts a trust assessment and formulates its own trust opinions. At a specific moment, when the vehicle encounters another vehicle, it requests the trust level of the observed vehicle from the Multi-Access Edge Computing (MEC) infrastructure.

The CONNECT information flows, as it pertains to the MEC side, are depicted on Figure 2.5. The process is intiated by the MEC receiving a T-CAM/T-CPM message from the vehicle(s). Two

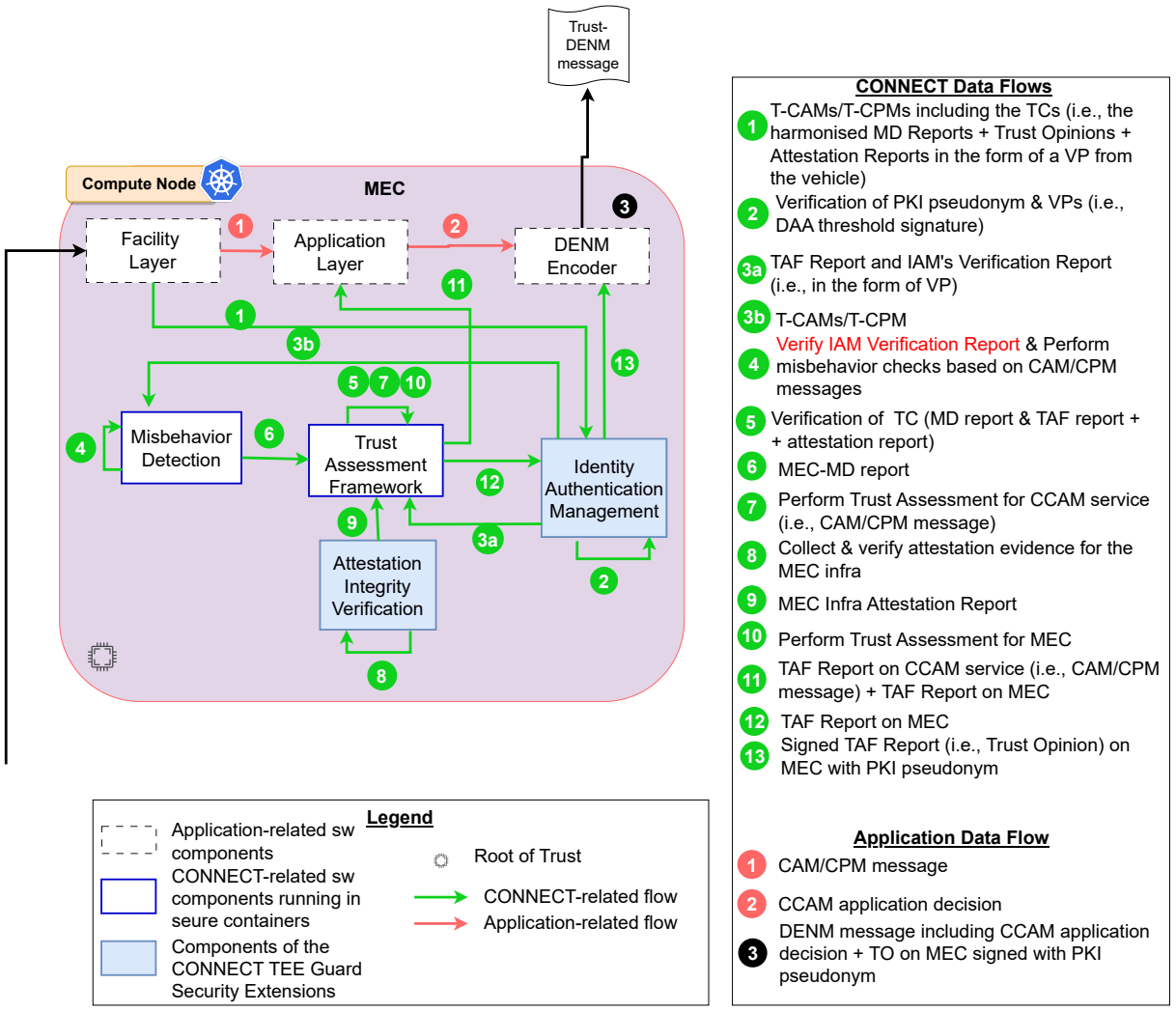

Figure 2.5: MEC Information Flows

separate flows are illustrated in the figure: the red flows depict the activities performed to support and provide (kinematic) data for the applications (i.e., the CCAM services), while those in green colour capture the interactions that take place between all *CONNECT*-related components for enabling both data- and node-centric trust assessments.

The flow begins when the MEC receives new T-CAM and T-CPM messages from the vehicle(s), including the Trustworthiness Claims (TCs) which encompass the Trust Opinion (TO) as calculated by the vehicle's *TAF*, the MD Report and the harmonised Attestation Reports, in the form of *VP*. This message is received by the Facility layer which forwards the T-CAM and T-CPM messages to the *IAM* **(step 1)**. Note that the T-CAM and T-CPM messages are also sent to the Application layer, where the CCAM service is executed (i.e., red flow **step 1**).

The *IAM* verifies the PKI pseudonym included in the *VP* as well as the aggregated threshold signature that is outputted by the vehicle's *TCH*, used to sign the Trustworthiness Claims (TCs), comprising Trust Opinions, Misbehaviour Reports, harmonised attestation assertions, following the scheme described in Chapter 5 **(step 2)**.

After the successful verification, the *IAM* sends the TAF Report (included in the *VP*) along with the *IAM* verification report to the MEC-based *TAF*, **(step 3a)** and the CAM/CPM message(s) to the MEC-based *MBD* **(step 3b)**. The latter, verifies the IAM report and performs its own misbe-

haviour checks on the received CAM/CPM messages **(step 4)**. The MEC-based *TAF* verifies the TCs (including the vehicle's MD, TAF and attestation reports) **(step 5)** and after the successful verification of the TC, it may continue with its own assessments.

Note that the TAF's scope in the MEC is dual, hence two assessments take place: i) a trust assessment for the CCAM service being executed at the vehicle, responsible for the collection and sharing of kinematic data and ii) a trust assessment for the MEC Infrastructure, where the cloud equivalent of the service is instantiated (i.e., traffic control). For the first, the TAF requires from the MD to send the MEC-based MD report, which is produced based on a variety of CAM/CPM messages derived from multiple vehicles **(step 6)**. The TAF combines this information along with its own Trust Model in order to produce a report for the CCAM service **(step 7)**. For the second type of assessment, the TAF requires the *AIV* report. As performed at the vehicle side, the AIV containerised service running on the MEC collects; thus verifies the attestation evidence about the target container, executing the CCAM service **(step 8)** and produces an attestation report, based on which a Trustworthiness Claim (TC) is created. This TC is sent to the TAF **(step 9)**, while it is circulated to the vehicle enabling to assess the MEC's level of trust when producing its own trust decision. As mentioned in the previous section, in order for the vehicle to be able to authenticate the correct reception of the evidence from a valid and authentic containerised service, it needs to be able to verify the identity of the workload. This is performed through the use of the certificate, bound to the attestation key, used by the *AIV* to sign the evidence, hence the evidence can be trusted by the vehicle.

Afterwards, the TAF verifies the attestation report and commences the trust assessment process for the MEC Infrastructure, producing a TAF Report **(step 10)**. The two TAF Reports (i.e., one for the CCAM service and the other one for the MEC Infrastructure) are sent to the Application Layer **(step 11)**. The Application Layer leverages this information to get the CCAM application decision (i.e., slow down the vehilce). Additionally, the TAF Report for the MEC Infrastructure is sent from the AIV to the IAM **(step 12)**. The latter signs the TAF Report on the MEC leveraging the PKI Pseudonym, and sends this information to the DENM Encoder, to be disseminated with the vehicles in the vicinity **(step 13)**.

Chapter 3

Secure Container Lifecycle Management on *Cloud/Edge/Vehicle* Continuum

3.1 Container Management in *CONNECT* Far-Edge & MEC

Chapter 2 has introduced the MEC-based *CONNECT* architecture, which incorporates both regular (i.e., legacy) containers, as well as confidential containers, that can be used to provide an enhanced environment, in terms of security, for the deployment and execution of MEC-based CCAM services. **CCAM applications can be supported either through legacy or confidential containers. The decision depends on the acceptable level of overhead in such safety-critical systems.** *Original Equipment manufacturer (OEM)s* or Infrastructure Administrators may opt for legacy containers when enhanced security is not critical, while confidential containers are chosen when advanced protection is paramount. Note that: (i) the *CONNECT* **components that comprise the *TCB* are executed within confidential containers**; while (ii) **to deploy a confidential containers the platform must provide a Trusted Execution Environment (*TEE*) equipped with a Root of Trust (*RoT*)**. Any infrastructure launching a confidential container needs a RoT to provide the necessary guarantees for the configurational and behavioural integrity of containerised services. This security measure aligns with *CONNECT*ś vision to provide advanced protection accompanied with verifiable evidence on the trust state of (HW and SW) elements across the entire CCAM continuum.

This container-based architecture mirrors, to some extent, the vehicle-based services described in Deliverable D4.1 [10]. Similar to the vehicle, components on the MEC are executed within containers deployed using a Kubernetes compute node. **In the vehicle this node is running within the In-Vehicle computer.** Managing containers is crucial for the *CONNECT* project to ensure smooth deployment and operation of containers across the Cloud/Edge/Vehicle continuum. This involves coordinating different signals to manage container lifecycle operations, such as deployment, startup, pausing, migration, and stopping. This uniform interface facilitates the management of containers, across the different layers of *CONNECT* (i.e., Cloud, Edge, and Vehicle). **The consistency in infrastructure throughout different layers simplifies the system architecture and guarantees uniformity in deploying and managing containers.**

As previously described, confidential containers are legacy containers that are enhanced with the necessary features to operate in a Trusted Execution Environment (i.e., Gramine-enabled containers in *CONNECT*). The detailed methods for accomplishing this goal are explained in the following sections, with the objective of achieving simplicity and scalability.

3.2 Docker-Style Container Management for TEE-protected Workloads

We now outline the interplay between a Docker-style container management infrastructure and TEEs. In Section 3.2.1 we outline how Container Management works for legacy services, without security. In Section 3.2.1.1 we then document the security objectives that we plan to achieve. In Section 3.2.2 we then document how a legacy container (without TEE-enablement) can be converted into a ”Graminized” container that includes the Gramine LibraryOS and TEE-based hardware protection. One tool for seamless migration is called enclave-cc. We outline the enclave-cc architecture in Section 3.2.3. We conclude the chapter in Section 3.2.4 by describing how the required state transition work once the container is TEE-enabled.

3.2.1 Legacy Containerized Service Instantiation

The process of instantiating containers in K3s involves multiple components interacting with each other, for the deployment of the service workloads at particular node(s) within the cluster, e.g., to the *CONNECT* MEC nodes. In what follows, the analysis focuses on the default process employed for the instantiation of containerised services either on the MEC or the Vehicle Computer. For instance, consider the envisioned use case of Collaborative Cruise Control where a CCAM service/application provider aims to deploy this service along its functions within a vehicle. In this scenario, the service will be instantiated in both spectrums of the continuum so as to also allow for the possible migration/offloading of specific functions to the MEC - either in the case of a possible compromise (change in the trust level of an ECU) or due to lack of resources of the execution of specific operations. The objective of this chapter is to detail the entire lifecycle management of a continerized service and highlight the core innovations brought by *CONNECT* as it pertains to the coupling of the two adopted technologies: these of **trusted and confidential computing**. As will become evident, *CONNECT* is the first of its kind to integrate the use of enclave-CC technology over the Gramine SGX-based TEE as the underlying Root-of-Trust. However, a more focused description of the employed Kubernetes orchestration technology can be found in D5.2 [13].

The process begins with a user/administrator (or an automated system) creating a deployment through the K3s API server (see Figure 2.4). This is typically done using kubectl, the Kubernetes command-line tool, to submit a YAML file (or a set of manifest files) that describes the desired state of container services, or via a Kubernetes package manager such as helm3, that helps streamline the deployment and management of Kubernetes applications.

The API Server submits the new deployment to Kine (i.e., the translation layer between Kubernetes and the underlying database, as described in Section 2.3.1 and D5.2 [13]), along with its status (i.e., new assignment, pending for deployment). The scheduler constantly watches the API Server for new work items (Pods that need to be assigned to a cluster node). Once a new assignment is found, the Scheduler queries the API Server to gather current data about the state of the cluster. This includes information about the resources available on each node, such as CPU, memory and storage, affinities, anti-affinities, taints, tolerations and other scheduling constraints that are defined in the Pod’s specification or the cluster’s policies. The Scheduler uses this information to evaluate which nodes have sufficient resources to run the pod. Once a node is selected, the Scheduler communicates back to the API Server to bind the pod to the chosen node, and API Server updates the pod’s information in Kine to reflect this binding.

Through the Tunnel Proxy, the Kubelet agents at each worker node are continuously watching the

API Server for pod bindings. Once a kubelet detects that a new pod assignment is destined for its node, it interacts with the container runtime engine (e.g., docker, containerd, cri-o) to prepare for container deployment. Particularly, Kubelet passes the container image, environment variables, volumes, and other necessary information (based on the k3s service manifest) to the container runtime, e.g., containerd. Subsequently, containerd pulls the container image from the image registry (or uses a local copy if available/instructed). After the image is pulled, containerd creates the container based on the specifications provided by the service manifest files.

Additionally, the kube-proxy and the network plugin (e.g., flannel) create the necessary virtual networks to establish communication between different Pods (potentially across hosts) or for Pod communication with services outside the cluster. Once the container is running, containerd continues to manage its lifecycle and reports the status back to the Kubelet. The Kubelet updates the status of the Pod in the API Server, which in turn updates the overall deployment status in Kine. This status information is also constantly monitored by the controller manager which performs automated health checks and adjustments when needed, to ensure that the desired state declared by the user in the deployment phase, is achieved and maintained.

Up till now, we have elaborated upon the launching of a docker legacy container. Nevertheless, as described in Section 2.3.1, the process adopted by CONNECT *further includes the verification of the container, through the recalculation of the MR Enclave Reference Value Measurement. This step is introduced by* CONNECT *to include a short of verification regardless of the container type (i.e., legacy or confidential container).*

3.2.1.1 Problem Statement: TEE-Enablement of Docker Containers

The goal of our research and development within the *CONNECT* project is to allow for seamless integration of confidential containers into a security-enhanced container management framework. Today, an application that was SGX-enabled by its developer is also capable of running within a container. Since the security guarantees of Intel SGX are self-contained, this approach will not affect the security (i.e., in terms of confidentiality/integrity) of the contained secure enclave. We structure our approach into two phases of the life-cycle of a container:

On-boarding During the first phase, the container and its included applications need to be security-enabled by its designer. Our goal is to automatically on-board most containers without the need of developer intervention.

Management While Intel SGX ensures security, the container-style management (starting, stopping, migrating) may render the state of the enclave inconsistent, which may reduce its availability. For instance, if an enclave is terminated at an inopportune moment, without having stored its state, cryptographic and other security protocols may encounter issues during the enclave's subsequent restart. To allow container-style state management, we will explain how a TEE can securely safe and restore its state in line with the desired life-cycle of the container.

The container management extension, Enclave-CC, aims to automatically convert any container image into a Gramine-enhanced container that is suitable for execution in an Intel SGX TEE. As aforementioned, one of the **core innovations of *CONNECT* is improving the integration of Gramine and enclave-CC technologies**. To achieve this integration, we plan to augment the existing Gramine baseline TEE image with improved integration hooks and test it against sample

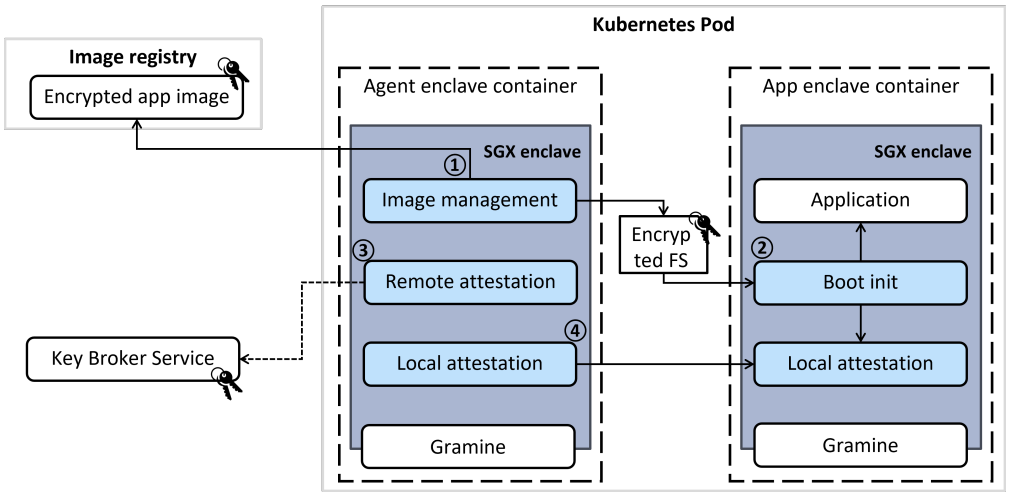

Figure 3.1: Integration points of Enclave-CC with Gramine.

Enclave-CC application images and flows. We have already enabled remote attestation flows based on the integrity checks (of the correct conversion of a legacy container to a Gramine-enabled container) performed by the *Key Broker Service (KBS)* as described in Sections 2.3.1 and!2.3.2 and depicted in Figure 3.1.

The initial integration point (that has been completed) revolves around image management **(Step 1)**. Specifically, when an Agent enclave (operating with Gramine) initiates the deployment of the application, Gramine must permit the Agent program to establish a secure SSL/TLS connection to the Image registry. This connection is leveraged to download the encrypted application image, decrypt it using the image encryption key, and subsequently re-encrypt it using an SGX-platform-specific sealing key. Gramine has all the required features to implement this integration point: Gramine supports SSL/TLS sessions over TCP/IP and has the ability to download files using "recv()" system calls. Gramine also provides utilities to decrypt and encrypt data using different keys; these utilities are based on mbedTLS library's functionality. Finally, Gramine exposes SGX-platform-specific sealing keys, so that the Agent program can use them for re-encryption.

In continuation of this process, the second step in Figure 3.1, accounts for the acceptance of the encrypted file system of the application image **(Step 2)**. In particular, when an Application enclave (that also runs with Gramine) is triggered by the Agent Enclave to start the application, Gramine must detect the encrypted file system of the application image on the host disk, copy it inside the SGX enclave, decrypt it using the SGX-platform-specific sealing key, reconstruct the file system hierarchy from this decrypted image and present it to the application (so that the application can see the files and operate on them). This second integration point *may* require new functionality in Gramine, because image file systems typically have properties which are not currently implemented: i) application image may have symbolic and hard links, ii) application image may specify file creation, access and modification times, iii) application image may have non-regular files such as named UNIX domain sockets. Therefore, this second integration point may require implementing some or all of the aforementioned functionalities in Gramine.

The final integration points concern the introduction of the enhanced attestation flows, mentioned in Chapter 2, for allowing both the **integrity check and credential binding of a confidential container** but also the **Enhanced Configuration Integrity Verification (CIV)** based on the newly designed protocol described in Section 5.4 **(Steps 3 and 4)**. For the former, an Agent enclave must connect to the KBS to obtain the image encryption key. The Agent must verify the trustworthiness of KBS, to gain trust in the key that it receives. Similarly, the KBS must verify the

trustworthiness of the Agent enclave, to gain trust in this Agent and to release the secret key to it. In other words, Gramine will be acting as the mediator for exposing the necessary interfaces to allow the integrity verification of the Application enclave.

For the latter **(Step 4)**, the two enclaves must verify each other's trustworthiness, to establish a shared SGX-platform-specific sealing key that encrypts the image's file system. In particular, the Application enclave must request this key (more specifically, the parameters to derive this key) from the Agent enclave; to this end the Application enclave must be sure that it communicates with the genuine Agent enclave that provides the correct encryption key. On the other hand, the Agent enclave must release this key only if it verifies that the Application enclave is genuine and is expecting the correct application image. Gramine has low-level attestation primitives for SGX local attestation, but unfortunately, it currently lacks a higher-level library to simplify the coding of "SGX local attestation over TLS" flows (this could be called *LA-TLS*). Therefore, the last integration point may require adding new functionality to Gramine – the LA-TLS library to establish TLS connections coupled with SGX local attestation.

3.2.2 On-boarding Containers while Enabling TEE-based Execution

The first challenge to address is on-boarding a workload to create a docker image that is executed within an Intel SGX Trusted Execution environment (see Story-XXV). We will outline two alternatives: The first alternative will "graminize" an image at design-time to create a container image that contains an application as well as Gramine. This image can then be stored in the image repository and instantiated like any other container. However, this approach has two limitations: (a) The designer needs to be aware that the container will be executed in a TEE and is required to prepare for it and (b) Each and every TEE-enabled image contains a copy of the Gramine operating system.

To overcome these limitations, the second approach adds Gramine at run-time. The core ideas is that the workload-image and the Gramine-image are kept separately. When a secure container is requested, both images are combined at run-time and then started within the TEE.

3.2.2.1 Background: Creating Images for Secure Containers at Design-Time (CASE A)

This approach requires a standard Docker image. By using a specific tool called `GSC`, this image can then be converted. To do so, the runs it through the GSC tool to obtain a "graminized" image. Let us consider a specific example: assuming that the original Docker container image has a name "helloworld":

```
$ gsc build helloworld helloworld.manifest
$ gsc sign-image helloworld private-key.pem
```

The resulting "graminized" Docker container image has the name "gsc-helloworld". This image can now be encrypted with a user-private key and uploaded to the Enclave-CC image registry. A corresponding encryption key should be securely uploaded to the Key Broker Service's Key Manager database. The important enclave measurements of the resulting image must also be submitted to the Key Broker Service, so that it will be able to verify the authenticity of the application image pulled by the Agent enclave. The Key Broker Service also needs a configuration to know how exactly to run this container image; the configuration is a JSON file that may look like this:

```
{
  "metadata": {
    "name": "helloworld-encrypted-container"
  },
  "image":{
    "image": "ghcr.io/confidential-containers/helloworld-container-enclave-cc"
  },
  "command": [
    "/usr/bin/gramine-sgx",
    "/usr/bin/hello_world"
  ],
  "log_path":"helloworld.log",
  "devices": [
    {"container_path": "/dev/sgx_enclave", "host_path": "/dev/sgx_enclave"}
  ]
}
```

Finally, the user must create a Pod definition to use this confidential container image in a Kubernetes deployment. The Pod definition would have the following configuration:

```
apiVersion: confidentialcontainers.org/v1
kind: CcRuntime
metadata:
  name: ccruntime-enclave-cc
  namespace: confidential-containers-system
spec:
  runtimeName: enclave-cc
  config:
    installType: bundle
    payloadImage: ghcr.io/confidential-containers/helloworld-container-enclave-cc
    installerVolumeMounts:
      - mountPath: /etc/containerd/
        name: containerd-conf
      - mountPath: /etc/enclave-cc/
        name: enclave-cc-conf
    installerVolumes:
    installCmd: ["/opt/enclave-cc-artifacts/enclave-cc-deploy.sh", "install"]
...
```

After these preparation steps, this Pod definition can be fed to the Kubernetes cluster, which will allocate a Kubernetes SGX node to run the Agent enclave container and the Application enclave container, and the user workload will start running inside the Application enclave.

3.2.2.2 Enclave-CC Goal: Run-time Conversion of a Container (CASE B)

Up until now, we have discussed the deployment of *TEE*s like Gramine-SGX on bare metal systems, in a manual fashion. In reality, most production environments use automated deployment tools, coupled with virtualisation and transparent scheduling and scaling of available machines. This is typically achieved by wrapping user workloads in containers such as Docker containers, and managing these containers via an orchestration system like Kubernetes [3].

To reap the benefits of trusted execution, the traditional containers and their orchestration model must be enlightened to work with *TEE*s, providing confidentiality of user workloads in the cluster of nodes. Such enlightened containers are called confidential containers (*CoCo*s).

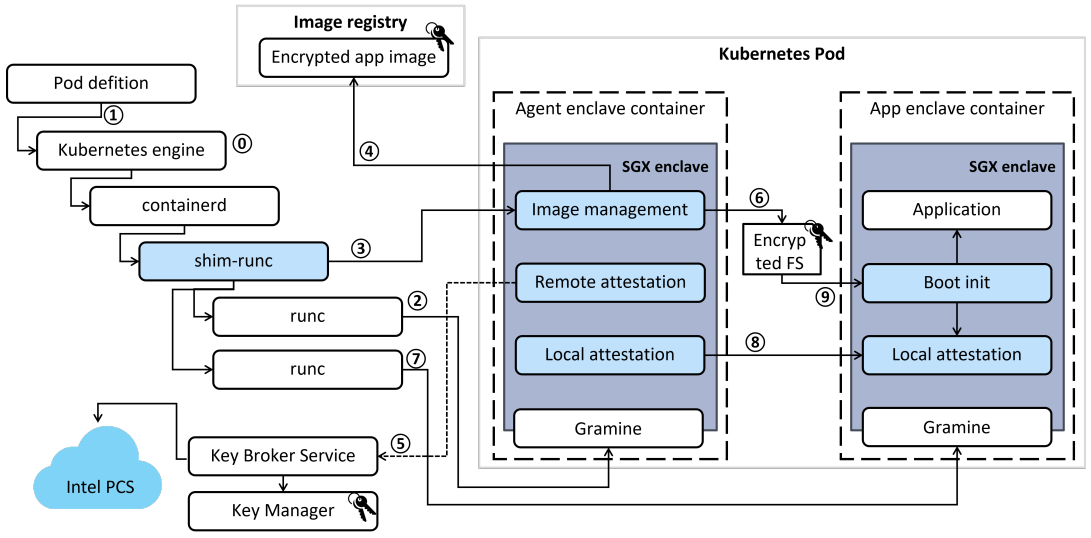

Figure 3.2: Enclave-CC architecture and flows.

Confidential Containers use hardware-based TEEs for resource isolation, data protection, and remote attestation [7]. They can protect sensitive data in use from privileged third parties. Confidential containers have the following characteristics:

- Provide confidentiality and integrity for data, especially for run-time data.
- Facilitate end-to-end security for the deployment and operation of sensitive workloads.
- Provide a way to prove that the environment in which confidential workloads are launched is authentic and trustworthy.
- Tenants maintain almost the same experience as using ordinary containers, but can deploy sensitive applications with greater confidence in their security.

Confidential container isolation can take two forms: i) process-based and ii) virtual machine (VM) based isolation. The *Enclave-CC* project provides a *process-based confidential container solution*, leveraging Intel SGX [5]. Enclave-CC does not measure and attest to the workload (user application); instead it uses a generic SGX enclave to isolate user's application from the rest of the system. Measurement and attestation of this empty generic enclave are used in building trust to run the application securely.

Enclave-CC provides all the elements for a complete container deployment flow, from creating a user container image with encryption/signature to uploading it to an image registry, to pulling the image, verifying its signature and decrypting it, unpacking it into the file system inside the SGX enclave, and finally starting the user application inside the confidential container. With this approach, all container processes are isolated using Intel SGX enclaves.

3.2.3 The Architecture of Enclave-CC

Figure 3.2 depicts the architecture and the general flows of Enclave-CC coupled with Gramine and Intel SGX. In the figure, the blue-painted boxes are the components that are developed in the

scope of the Enclave-CC project. Note that integration of Enclave-CC and Gramine is depicted in the figure, but it is currently an *ongoing effort*.

The main components in the Enclave-CC architecture are:

- **Agent enclave** – an Enclave-CC process that runs inside Gramine-SGX. It accepts requests from `shim-runc` and handles image management, SGX local and remote attestation, and encrypted file system management.
- **App enclave** – a user-application process that runs inside Gramine-SGX. It starts as a generic empty enclave and waits for instructions from the Agent enclave. When the Agent enclave has prepared all app-enclave data, the App enclave obtains the encrypted file system blob, decrypts it, and starts the execution of the client application.
- `shim-runc` – a standard shim component that sits between `containerd` and `runc`. It accepts requests from `containerd`, starts and pauses the Agent enclave container and asks the Agent enclave to perform image management actions. All containers started by `shim-runc` are instantiated by the traditional `runc` tool.
- **Image registry** – The Agent enclave container image and the encrypted App enclave container images are put in the image registry. Upon request from the Agent enclave container, the image registry releases the encrypted App enclave container image.
- **Key Broker Service (KBS)** – a service that verifies the trustworthiness of the Agent enclave, by performing SGX remote attestation with it and consulting the Intel PCS service to make sure the Agent enclave is a genuine SGX enclave executing the expected code. Key Broker Service also serves as a front-end to the Key Manager database that holds the encryption keys for encrypted application images.

The typical flow depicted in Figure 3.2 proceeds as follows. **(step 0)** The operator of the cluster installs the Enclave-CC run-time in the cluster. In particular, this installation copies `shim-runc` and `containerd` binaries into Kubernetes master nodes, adds image bundles of the Agent enclave container and the Boot (empty generic) enclave container to the image registry, and re-configures `containerd` to enable Enclave-CC run-time. After the Enclave-CC run-time is successfully installed by the operator, the application workloads can be deployed.

To deploy an application in the Kubernetes cluster, the remote user defines a Pod configuration to describe the workload and run-time requirements **(step 1)**. The user deploys this Pod definition into Kubernetes and the request is propagated to `containerd`. The `shim-runc` tool receives the container creation request from `containerd` and creates the Agent enclave container using `runc` **(step 2)**. Next the `shim-runc` tool requests the Agent enclave to pull an encrypted application container image **(step 3)**. The Agent enclave receives this image pull request and downloads the encrypted application image from the image registry **(step 4)**.

Then the Agent enclave must make sure that the encrypted image is known to the system; to this end the Agent enclave performs an SGX remote attestation with the trusted Key Broker Service, which verifies the identity and trustworthiness of the Agent enclave and confirms the correctness of the downloaded image, as well as sends the encryption key for this image **(step 5)**. After remote attestation, the Agent enclave decrypts the downloaded application image, re-encrypts the application image with a randomly-generated ephemeral key and constructs a new encrypted file system out of this image **(step 6)**. In parallel to this process, `shim-runc` creates

the Application enclave container and starts the Gramine instance, which performs the “empty generic enclave” boot-up **(step 7)**.

After the Application enclave is fully booted and initialised, it waits for the SGX local attestation request from the Agent enclave. During local attestation, the Agent enclave sends the application workload configuration, the path to the encrypted file system blob and the encryption key for this file system **(step 8)**. The Application enclave locates the encrypted file system, moves it inside the SGX enclave, decrypts it and installs it as the Gramine file system **(step 9)**. After this flow completes, the application can finally be run inside Gramine.

3.2.4 Life-Cycle Management of TEE-Enhanced Containers

In Section 3.2.2, we outlined how to create a TEE-enhanced container that is ready for deployment in a Kubernetes cloud infrastructure. We now describe considerations for the life-cycle management of a TEE-enhanced container (see Story-XXVII). The core requirement is that the container implements features to support the lifecycle of a Kubernetes Pod [39]. If we assume that the original container (without TEE-enhancements) implemented these requirements, then our goal is that the recently added TEE does not break this implementation. In general, the container life-cycle is structured into these phases (quoted from [39]):

Pending: The Pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run. This includes time a Pod spends waiting to be scheduled as well as the time spent downloading container images over the network.

Running: The Pod has been bound to a node, and all of the containers have been created. At least one container is still running, or is in the process of starting or restarting.

Succeeded: All containers in the Pod have terminated in success, and will not be restarted.

Failed: All containers in the Pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or was terminated by the system.

Unknown: For some reason the state of the Pod could not be obtained. This phase typically occurs due to an error in communicating with the node where the Pod should be running.”

By default, most containers are stateless while state is kept either outside the system (on a database server) or in a few well-defined containers that are part of a `StateFulSet`. The goal of this approach is that it supports scaling (any number of containers can be started for a given container image to scale the workload. Similarly, without state, a `failed` container can be destroyed and restarted to try to recover the workload. Since a TEE only adds protection (and not state), the same holds for TEE-enhanced containers. They can be killed and restarted and - whatever signals the container received - they will automatically also affect the TEE that is part of this application.

The challenging part is the life-cycle management of TEE-enhanced containers where the TEE contains state that must be preserved (see Story-XXVII). To address this need, the main requirement is that the TEE can save its state when receiving a `SIGINT` signal and recover its state when being restarted (see Story-XXIV). This is implemented in two phases. Depending on the programming language, this may require explicit support by the developer to preserve a consistent state.

3.2.4.1 Can Application State be Saved Automatically?

Our initial goal was to automatically snapshot and dump the state of a complete application. Based on our research [29] we concluded that agnostic snapshotting of any application may be possible in some simple cases (e.g. some single-threaded application written in `C`) but will not work in general. Example challenges we identified are:

- *How to reconstruct the opened file metadata: file position, file mappings into the process, etc.?*
- *How to re-establish pipes, especially encrypted pipes (if something was already sent on them, thus TLS state is not clean)?*
- *How to re-establish TCP sockets, especially if they send TLS packets (same problem as with pipes)?*
- *What to do with check-pointing multiple threads and processes?* While one thread is performing checkpoint, the other threads modify the state, leading to inconsistent state in the check-pointed multi-threaded blob.

3.2.4.2 Application-Centric Management of State

As a consequence, we will focus on application-centric management of state. This requires that applications need to be designed for TEE-based execution. This includes that an application must be able to snapshot itself and later restore all essential parts to continue its execution. This requires two extensions for each stateful application that we plan to execute within a TEE:

Saving TEE State: To enable a TEE to save its state, the manifest needs to include a file that is sealed to `MRenclave`. This ensures that only the given enclave can read this file once its integrity has been verified. In case the kubernetes infrastructure decides to shutdown a Pod and the contained Containers, it sends a `SIGTERM` signal and calls the hook PreStop. This signals to the TEE-enhanced application that termination is imminent and that its state must be saved. To do so, the application serialises and stores its state in the sealed file. After completing this dump of its state, it exits.

Restoring TEE State: When a TEE is started, the TEE examines the sealed file. If it detects a sealed state, it de-serialises and restores the state. Then, it commences its prior execution.

Chapter 4

Refined User Stories for Security-Critical Features of *CONNECT*

Chapter 4 illustrates the important behaviours captured by our architecture by using so-called ”user stories”. As it was the case in D4.1 [10], the goal is to document the requirements and functional specifications of different roles/user groups considered in the overall CCAM ecosystem and, therefore, in the *CONNECT* deployment environment. Each user story outlines well-defined usages by given user roles/groups together with their security and functional specifications. This serves as a high-level description of requirements that need to be satisfied by the services and architecture that is provided by *CONNECT*.

Changes compared to D4.1 [10]: To ensure that we document the complete set of requirements, we have kept (and expanded) all functional specifications that were documented in the previous version of this deliverable. *Recall that the focus of the first version of* CONNECT *Trusted Execution Architecture was mainly on those safeguards needed for capturing the security and trust requirements of the far-edge (i.e., Vehicle) while in the current version the necessary refinements are described to also capture the best practices needed for the secure deployment of CCAM services to the entire continuum - from the Vehicle to the MEC and Cloud facilities*. Therefore, we have made minor refinements and bug fixes for user stories documented in Sections 4.2-4.6 that focus on the: (i) secure establishment of an in-vehicle environment comprising ECUs to be equipped with the necessary crypto primitives for securely participating in the overall trust assessment process, and (ii) runtime attestation attributes extracted as trust sources to allow for the dynamic quantification of node- and data-centric trust. The remaining user stories starting with Section 4.7 include larger refinements or are newly added: In particular, the user stories capturing the behaviour of the MEC (Section 4.8) are a new addition focusing on those levels of assurance needed to be ensured towards establishing the trustworthiness of all MEC information and functional assets. A summary of the status of all user stories is listed in Table 4.1.

4.1 Introduction to the *CONNECT* User Stories

The user stories outlined here are designed to illustrate the *CONNECT* functionalities that will be used to provide trustworthiness evidence, both for the *TAF* and the *TCH* and how this evidence is used to provide *VCs* and *VPs* that will be sent outside of the vehicle while maintaining its privacy.

The user stories are divided into: (a) general user stories – those that are agnostic to the type

User Story	Status	User Story	Status	User Story	Status
Story-I		Story-XI		Story-XXI	
Story-II		Story-XII		Story-XXII	Revised
Story-III		Story-XIII		Story-XXIII	Revised
Story-IV		Story-XIV		Story-XXIV	New
Story-V	Revised	Story-XV		Story-XXV	New
Story-VI		Story-XVI		Story-XXVI	New
Story-VII	Revised	Story-XVII	Revised	Story-XXVII	New
Story-VIII		Story-XVIII		Story-XXVIII	New
Story-IX		Story-XIX		Story-XXIX	New
Story-X		Story-XX	Revised	Story-XXX	New

Table 4.1: Updates to the *CONNECT* set of User Stories compared to D4.1.

of hardware used, and (b) implementation based user stories – those that are specific to the hardware that *CONNECT* will use when implementing these systems, i.e. Intel-SGX and Gramine (starting with Story-XVI). Figure 4.1 shows a simplified diagram of the devices in the vehicle and the software components in the main Vehicle Computer. Note that in the underlying design of the *CONNECT* system each of the TEE-guard components - namely the *IAM*, the *AIV* and the *TCH* - will run in an isolated environment with its own *TCB*. However, for implementation using Gramine (described in [10]) we will have the entire TEE-guard running in a single secure container, having the *TCH* and *IAM* components running as children *Intel SGX is a TEE provided by Intel CPUs that allows to execute a user-space process within a hardware-protected execution environment that is called* enclaves *(Enclaves)* of the *AIV Enclave*, as depicted in Figure 4.2. For better clarity, we have opted to showcase the positioning and interactions between all components, comprising the *CONNECT TEE-GSE,* with a different level of abstraction: Figure 4.1 highlights the overarching architecture of the In-Vehicle Manager depicting the interactions between the CONNECT security components that take place over different phases of the entire lifecycle of the vehicle mapped to the user stories described in the following sections. Figure 4.2 captures a more detailed version of this in-vehicle security architecture showcasing also the exact instantiation of all components based on the use of the Grammine TEE technology.

We start with the general user stories. These are further sub-divided into: (a) those that are used to illustrate how a *CONNECT* vehicle is setup and configured, (b) those that show how, once the *CONNECT* vehicle's systems are up and running, evidence of the trustworthiness of data and applications is collected, assessed and communicated and (c) those that illustrate how, when the trust level in an *ECU* falls, critical applications running on that device can be migrated to another (more trustworthy) *ECU*.

General pre-requisite - Devices within the vehicle are all clearly identified and can be routinely addressed.

4.2 User Stories for Preparing the Vehicle

To enable the *CONNECT* security architecture and services, the vehicle needs to be set up during manufacturing and assembly. The following user stories describe key tasks that are required

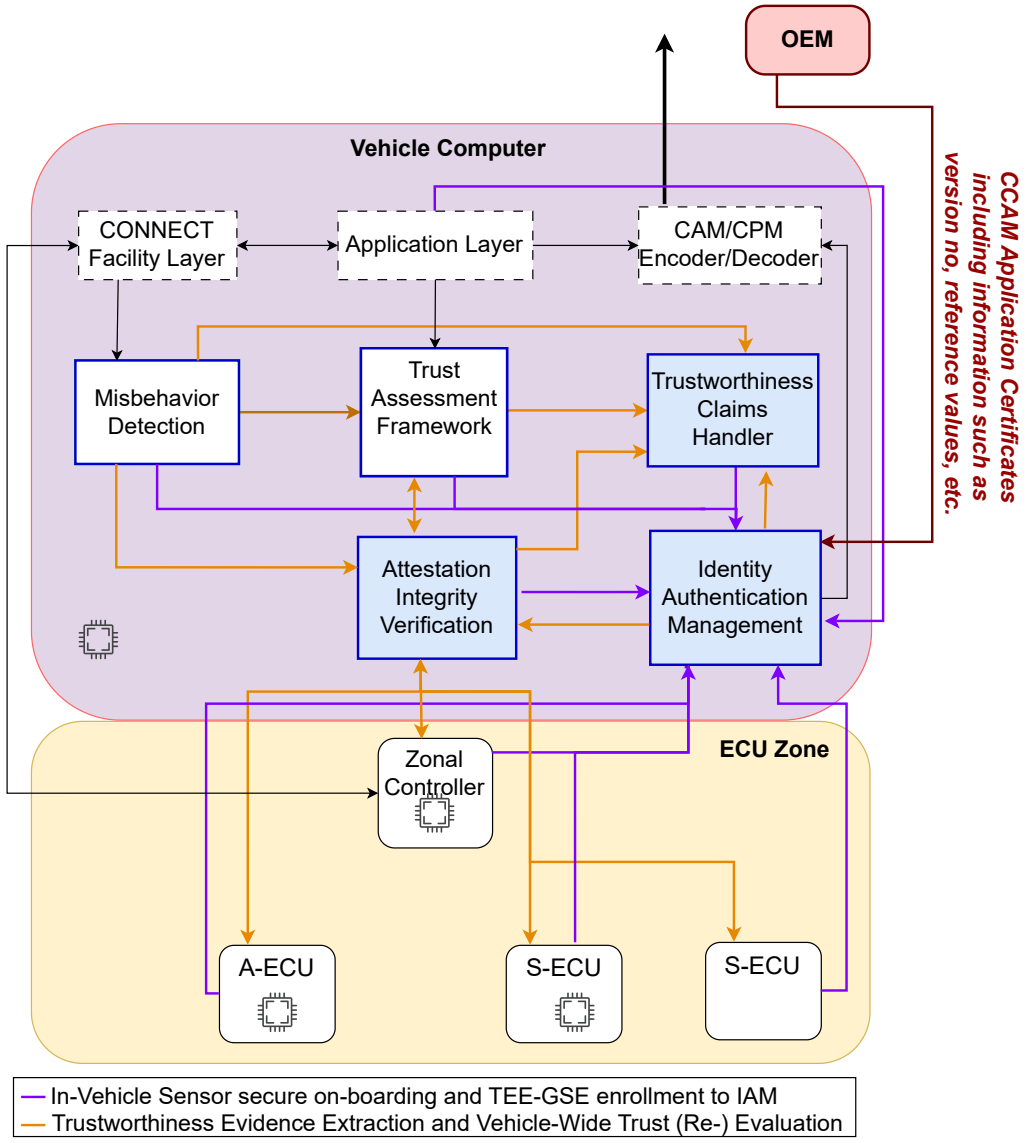

Figure 4.1: *CONNECT* In-Vehicle Logical Architecture capturing all functional specifications depicted through the described user stories.

during this phase. The outcome is a vehicle where all keys and software are installed and that is now ready for operation.

Story-I: Configure a device ready for installation into the vehicle.

Objective: To configure a device ready for installation into the vehicle.

Motivation: Before installation into the vehicle all devices need to be configured with the cryptographic keys that they need and the correct software installed. This will include application-specific keys as needed by the *Connected, Co-operative and Automated Mobility (CCAM)* applications and *CONNECT* keys used to provide evidence in a trustworthy manner. This is the first stage in this process and is carried out by the *Tier 1 Supplier*. More details of the protocols used are given in Section 5.3.

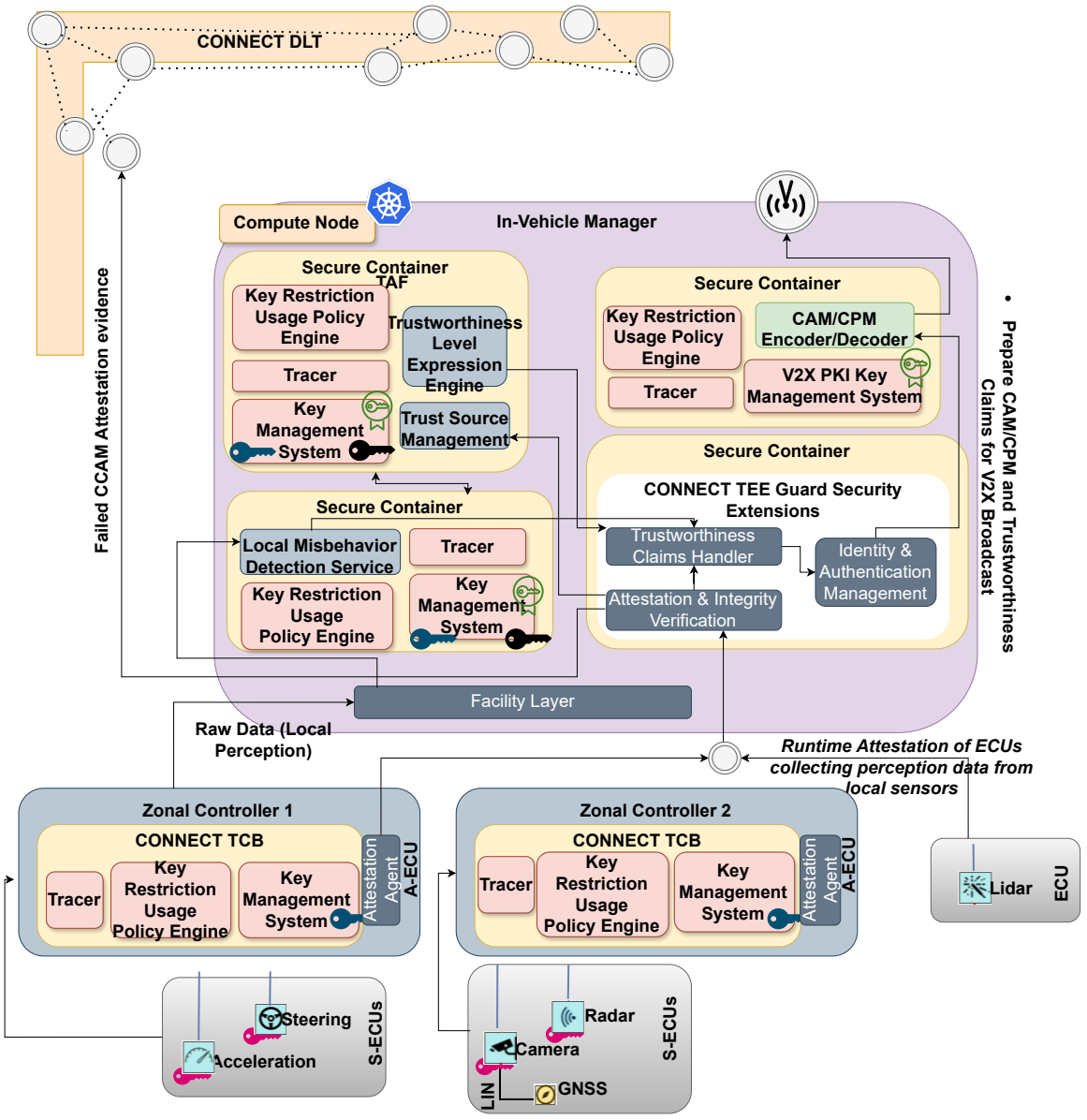

Figure 4.2: *CONNECT* CONNECT In-Vehicle Implementation Architecture depicting detailed positioning and interactions between the *CONNECT TEE-GSE* components. Detailed version of Figure 4.1 on page 33.

Requirements: The device should have a unique identity and an associated key pre-programmed by the (*Tier 2 Supplier*) device manufacturer. The identity and associated key pair are provided to the *OEM* when the device is supplied.

Story-II: Install and set-up the vehicle computer's IAM.

Objective: To install *IAM* software on the vehicle computer and install the vehicle computer's long-term master key (VKM).

Motivation: The IAM is one of the *TEE-GSE* components and runs protected by a *TEE*. It man-

ages the installation and update of the different software components running on the vehicle computer (as Gramine-enabled confidential containers) and stores the *VCs* containing their configuration and attestation information (as extracted by the *KBS*). It also manages the *CONNECT* range of keys which are used in the vehicle for attestation and to protect communication between its various components. These keys are derived from the vehicle's master key. Installation and setting up the IAM and provision of the VKM is carried out by the *OEM* as the first stage in configuring the vehicle computer and setting up its software.

Requirements: The vehicle computer (a powerful A-ECU) has already been configured by *Tier 1* (Story-I).

Story-III: Install the vehicle computer's software.

Objective: To install and configure the different software components that will run on the vehicle computer.

Motivation: There will be a number of software components running on the vehicle. These will include specific *CONNECT* containers (such as the *AIV* and *TAF*) and *CCAM* applications (such as *Co-operative Adaptive Cruise Control (C-ACC)* and *Intersection Movement Assistance (IMA)*) running outside of a TEE (see Figure 4.1). They will all need to be downloaded, verified and configured with the keys that they need (Story-IV). The configuration and attestation reference values will be provided by the OEM, or software supplier, in a *VC*. The *IAM* will store the *VC* and use the information that it contains to confirm that the software has not been modified and has the expected version number. Provided that these tests pass successfully, the *IAM* considers the software component as securely enrolled and exchange all application-specific and *CONNECT* keys.

Requirements: The IAM will need to be installed and configured beforehand. In addition, the IAM will control this process and also manage any updates as they are needed. (Story-II).

Story-IV: Configure the necessary keys for the different vehicle computer's software components.

Objective: To provide the vehicle computer's software components with the keys that they require.

Motivation: Each of the vehicle computer's software components needs to be able to protect its data and where necessary to sign that data to prove its provenance. So, for example, the *TCH* will need to be able to verify the signatures on the *VCs* that it receives from other components and to generate and sign *VCs*, or *VPs*, that it uses to send data to the other components. For the *TCH* some data will be sent internally and can be 'straighforwardly' signed (e.g., using ECDSA) while other *VCs* will need to be anonymously signed. In order to be able to anonymously sign the *VC* the *TCH* will need to obtain a credential from a Privacy CA for its key. This will involve establishing a secure channel to the Privacy CA and

running the protocol that is used to issue credentials. Similar requirements also apply to the other software components running in the vehicle, with the *TAF* and *MBD* also needing to sign the *VCs* anonymously.

Requirements: Initially, the software should have been downloaded and verified. Part of this configuration might be done as the software is installed (Story-III) while some may be delayed until the ECUs are also on-boarded and configured (those for securely communicating with the *ECUs*). Where necessary the software component should be able to establish a secure connection to a Privacy CA for the issue of credentials that allow the component to anonymously sign its *VCs*.

Story-V: Secure on-boarding of an *ECU* into the vehicle.

Objective: As an *OEM* I want to enable the authentication and secure on-boarding of an *ECU* into the vehicle and setup the necessary keys (both the *CONNECT* security-related keys and the application-related keys)

Motivation: All *ECUs* need to be configured with the software and cryptographic keys that they need (see [10]). The software to be installed will depend upon how the *ECU* is to be used while the protocols used for on-boarding will vary depending on the type of *ECU*. These protocols are described in Section 5.3.1 for *S-ECUs* and Section 5.3.2 for *A-ECUs*.

Requirements: The device has already been configured by *Tier 1* (Story-I) and the vehicle computer's *TEE-GSE* has already been configured (Story-III).

Story-VI: Equipping the *IAM* with pseudonyms.

Objective: To obtain a set of pseudonyms from the *Public Key Infrastructure (PKI)* and install them into the *IAM*.

Motivation: A mechanism for protecting the privacy of the vehicle has been standardised by *European Telecommunications Standards Institute (ETSI)*: messages that are sent outside of the vehicle (for example CPM messages) should be signed using pseudonym keys. These pseudonyms are obtained by connecting to the *PKI* and are then stored in the *IAM* for later use. Note: the *IAM* will ensure that the pseudonyms can only be used under the condition that the vehicle is attested to be in a good state – this will provide an efficient revocation mechanism for the pseudonym keys.

Requirements: The *IAM* has already been configured by the *OEM* (Story-II).

4.3 User Stories for Assessing Trustworthiness of Vehicle or Services

An important functionality of the *CONNECT* architecture is to establish a Trust relationship between different CCAM actors through verifying the Trustworthiness Claims included in the CAM/CPM messages. Apart from establishing Trust relationships between different vehicles, *CONNECT* scopes to allow authorised stakeholders to monitor the trustworthiness levels of Trust Relationships between devices in the in-vehicle topology, thought storing evidences in the DLT. The following user stories specify the desired functionalities to allow this trust assessment.

Story-VII: Obtaining and verifying trustworthiness (attestation) evidence from the Vehicle's devices.

Objective: For the *AIV* component to obtain and verify (attestation) evidence, that was collected from the execution of an attestation task dictated by a Request For Evidence (RFE) in order for the *AIV* to report to the *TAF*, the *TCH* and, if it is needed, the *Distributed Ledger Technology (DLT)* (see Story-IX), on the devices (zonal controllers and the *ECUs*) that are included in the attestation request. Storing data in the *DLT* occurs only in case of a failed attestation event.

Motivation: In order to create a Trust Opinion (TO) for either a data item or a collection of nodes of the in-vehicle topology, the *TAF* needs an attestation report over the attestation/trustworthiness evidence from the devices providing that data. Similarly the *TCH* consumes the attestation report comprising the verification status of various system properties, depicting the Trust level of the attested system. More specifically, the *TCH* engages its harmonisation mechanisms, in order to create a harmonisation/abstraction of the attributes of the engaged devices and eventually create a VP of such a harmonisation. The components that are involved need to acquire verifiable attestation evidences from the attestation agent of each component. In *CONNECT*, as it was thoroughly described in deliverables D2.1 [12] and D5.1 [11], a CCAM application/service requests the calculation of a Trust assessment by the Trust Assessment Framework over a specific service for which a Trust model is already being deployed. This functionality is crucial to *CONNECT*, as it scopes to provide Trust quantification for a vehicle, establishing Trust relationships and eventually CCAM-wide Trust quantification. These quantification is provided by establishing a Trust chain between *CONNECT* components, from the *TAF* to the *AIV* and eventually to the *TCH*. In order for this Trust chain to be feasible on a zero Trust model, the Trustworthiness/attestation evidence should be constructed in a verifiable manner by all engaged components, in order for the system to be able to assess the validity of the provided claims. To this end, as described in Chapter 5 section 5.4, *CONNECT*scopes to employ advanced cryptographic mechanisms to ensure the Trustworthiness of the attestation evidence collected for a particular Request For Evidence (RFE).

Requirements: For this user story to be feasible, all capable devices (*A-ECUs* and *S-ECUs*) must have successfully completed the Secure On Boarding (section 6.8.1), in order to set up their secret keys and the appropriate key restriction usage policies. Furthermore, the *AIV* has to get a list of devices to be attested. This information comes from the *TAF* and is

part of the RFE. Additionally, the *AIV* needs to have information about how the attestation evidence should be verified. Thus, the *AIV* must have acquired from the *IAM* a mapping of all the possible attested devices to their reference values. For that purpose the *AIV* must be able to securely connect to the *TCB* exposed interfaces of each device so as to be able to collect the appropriate type of attestation evidence required.

Story-VIII:Trusting Verifiable Presentations.

Objective: *VPs* that a vehicle receives (which contains the *VPs*, for harmonised attributes, the *TAF*'s report (ATL) and the *MBD* Misbehaviour Report) should contain the cryptographic guarantees of the necessary information depicting the level of trust of the vehicle sending the *VP*, in order for the third party receiving it (either another vehicle, or the *MEC*, or some other trusted third party) to be able to verify it.

Motivation: Depending on pre-defined policies, vehicles send CAM/CPM messages, that will also include a *VP* reporting on the trustworthiness of the vehicle sending the data. Such a construction is called T-CAM/T-CPM, as it's the concatenation of a plain CAM/CPM message and TCs. This *VP* will contain information extracted from the *VCs* provided by the *TAF* (for its trust opinion), the *MBD* (for its misbehaviour report) and the *TCH* (for the harmonised attributes). From these, the *VP*, selectively discloses the information needed for the T-CAM/T-CPM consumer to be able to create its local trust opinion on the data origin without any privacy implications/breaches. Thus, each VP will be signed anonymously including a linkability token, which eventually is going to be signed by the *IAM* using a pseudonym key. As aforementioned, *CONNECT* is built on top of a zero Trust model. That being said, each *CONNECT* component that contributes to the creation of the Verifiable Presentation is not deemed by default Trusted. Thus, *CONNECT* components that are hosted by resource-capable devices, have to activate the attestation enablers of their underlying TCB to check whether or not they are not behaving maliciously. This is possible through the newly developed key restriction usage policies, as described in section 3.2. As a result each component is providing the necessary verifiable evidence, created from the component's unique secret key. The calculated VPs are depicting the Trust level of the vehicle that sent them and will be used from another vehicle or the MEC to formulate their own Trust Opinion from the received VP. As the VPs are the representation of a vehicle in the formulation of a Trust graph, the verifier needs to have strong mathematical proofs (VP verification) in order to Trust the VP producer.

Requirements: Each component of the *TEE-GSE* that will need to provide an anonymous signature, such as the Trust Assessment Framework (*TAF*), the Trustworthiness Claims Handler (*TCH*) and the Misbehavior Detection component (*MBD*), will have to be successfully securely enrolled. More specifically, each of these components have to set up its key restriction usage policy correctly and establish a trusted and authenticated communication channel to a *VC* issuer, in order to obtain the necessary verifiable credential that represent their hardware built-in attributes. The *IAM* will need to have obtained a pseudonym key from the PKI (Story-VI).

Story-IX: A vehicle stores the trustworthiness evidence of a failed attestation task (executed as part of a Request for Evidence from the TAF) to the Distributed Ledger (DLT).

Objective: The Attestation Integrity Verification (*AIV*) component, should store failed attestation evidence onto the *DLT*. Storing failed attestation evidence in the distributed ledger enables the *OEM* or any other Security administrator to process them, in order to pinpoint the vulnerability that was exploited and as a result identify and resolve zero-day vulnerabilities. This enables the re-calculation of the Required Trust Level of affected Trust Relationships and to keep track of the history of the trust state of a device like a reputation system to be potentially used by the federated TAF.

Motivation: When an attestation task fails, there is strong indication of risk regarding the attested device. For example, the integrity of the device does not meet the expected requirements. In this case, the failed attestation evidence is stored onto the Blockchain infrastructure so that it can be accessed later by the OEM or regulatory authorities for analysis of the compromised device. Henceforth, such authorities can then take actions on the analyzed malicious behavior, by either patching the existing software to fix found vulnerabilities, to withdraw faulty hardware that leads to malicious behavior, or updating the RTL in collaboration with the Trust Management Framework running on the cloud. Apart from the actions mentioned above, reporting on failed attestation tasks is an integral part of revocation and migration mechanisms (see Story-XIV). Attestation evidence for a device that is assessed to have failed the attestation test(s) (static and, where appropriate runtime) will be stored off-line with a pointer stored on the DLT. The data to be stored will be encrypted using ABE to restrict access to authorised parties (those with the correct attributes), such as the OEM or regulatory authorities.

Requirements: The *AIV* needs to be configured to have access to the distributed ledger. More specifically the Attestation Integrity Verification component needs to be equipped with the appropriate *VCs*, in order to be granted access to the *DLT* through ABAC. Additionally, the *AIV* has to be configured to perform Attribute Based Encryption. To make this feasible in the context of *CONNECT* use cases, during the secure on-boarding phase the *IAM* issues a policy for each pre-defined Trust model, including the attributes under which ABE is going to be performed.

4.3.1 Protecting Privacy during Trust Evaluations

While evaluating the trustworthiness of a T-CAM/T-CPM provider is important, disclosing all details of a vehicle is privacy invasive since it allows the verifier to identify a specific vehicle. As described in deliverable D5.1 [11], apart from the well studied cryprographic properties such as anonymity, unlinkability, untraceability and unobservability that are achieved through the traditional PKI-issued anonymous credentials, we are interested in the appropriate level of obfuscations regarding the exchanged trust-related information. This enables the continuous Trust assessment, without exposing sensitive information about the vehicle's architecture that can possible lead to numerous attacks.

Story-X: As the TCH I want to self-issue a valid VP, comprising trustworthiness (attestation) evidence adequately *abstracted*, so as to allow vehicle-wide trust appraisals by any receiving entity.

Objective: The objective of this functional specification is for the Trustworthiness Claims Handler (*TCH*) to be able to provide a Verifiable Presentation (*VP*). This *VP* is the cryptographically enhanced harmonized attributes, *TAF* and *MBD* report. More specifically, from an RFE the *TCH* receives an attestation report, a *TAF* report and possibly the *MBD* report. These information is the baseline for providing evidence on the integrity of a Trust model. As an expansion to this, in *CONNECT* we want to shut down every privacy implication that may be raised. For that purpose we employ harmonization mechanisms, in order to obfuscate the attributes of the attested Trust model and anonymized *VC* for the *TAF* and *MBD* report. Employing such advanced mechanisms, *CONNECT* manages to provide evidence regarding the Trust level of a vehicle in a zero-knowledge manner, as the *TCH* is not disclosing any information that can lead to the fingerprinting of the vehicles architecture or identity.

Motivation: As described in the deliverable D5.1 [11], *CONNECT* aims to establish Trust relationships to a CCAM ecosystem, thought attestation mechanisms, for all the participating vehicles, while preserving all aspects of the vehicle's privacy. The *TCH* receives the attestation report constructed by the *AIV* as the result to a Request for Evidence (RFE), that was circulated by the *TAF* for triggering the collection of the necessary trustworthiness evidence. The attestation report is consisted from a Verifiable Credential signed by the underlying hardware based key of the *AIV* and the attributes and system measurements of all participated devices depicting. The attestation report is comprising a service graph chain investigating all Trust dimensions, for all the participating devices. These attributes goes through harmonization mechanisms to converge in to a more abstract depiction of the devices architecture. These harmonization mechanisms, will be focusing on grouping together the same Trust properties of the attested system, so that they can depict the same type of Trust related information but in a more abstract way. For instance, such mechanism could be a special type of group based signatures or threshold signature schemes. The approach that is going to be employed eventually, will be investigated in the future. It has to be noted here, that this harmonization has to be verified by an external entity prior to calculating its own Trust opinion. For this purpose the harmonized attributes are used to calculate a Verifiable Presentation, with the TCH's secret key, which will be anonymized so as to preserve the privacy of the vehicle. To this end, *CONNECT*introduces a novel DAA threshold signature scheme(see chapter 5 section 5.6), that leverages the cryptographic characteristics of both the thresholds signatures and the DAA, in order to avoid the any privacy implications regarding both the vehicle fingerprinting and identity.

Requirements: For the completeness of this functional specification, the *TCH* needs to know the attributes of the attested ECUs so it can perform a correct harmonization each time a VP needs to be constructed for broadcasting to the MEC-instantiated *TAF* and/or the neighbouring vehicles. Apart from the disclosed attributes, the *TCH* needs to have successfully completed Secure Enrollment, in order to have its key restriction usage policy set up correctly.

Story-XI: As the TAF I want to self-issue a valid trust opinion VC based on the relevant trust sources.

Objective: The objective of Story-XI is for the Trust Assessment Framework (TAF) to be able to provide anonymous and unforgeable signatures, over a calculated TAF report disclosing the ATL of a Trust model that was attested.

Motivation: As aforementioned, a CCAM application may request from the TAF to calculate a Trust Assessment Request over a pre-defined Trust model. With the freshly now acquired attestation report, the Trust Assessment Framework calculates a Trust Opinion for this particular model, in order to be sent to the *TCH* and eventually outside of the vehicle. For this purpose the Trust Opinion is signed with the secret key of the TAF and gets associated with a link token to eventually construct an anonymized Verifiable Credential. By constructing an anonymized Verifiable Credential, we enable both the verification of the relevant Trust Opinion, and authorized entities (the link token's issuer) to trace back to the signer's identity. Moreover, between the *TAF* and the *TCH* we don't have any privacy implications, but for a verification performed by an external entity, *CONNECT* needs to assure that no information about the identity of the *TAF* that provided the *VC*. For this purpose, we need the *VCs* to be anonymized, but with an accountability factor in case the *TAF* is acting maliciously and necessary actions need to be made.

Requirements: For this usage story to be feasible the Trust Assessment Framework (TAF) needs to know all the relevant trust sources. More specifically, the *TAF* receives a report from the *AIV* component depicting all dimensions of Trust defined in deliverable D3.1 [9].The Attestation report, includes a mapping of the trust attributes of the ECUs that provided the attestation evidence, for the calculation of this Trust Opinion, along with a signature that depicts that the *AIV* is indeed in a correct/Trusted state.Moreover, as the *TAF* needs to provide its own anonymised signature, it needs to acquire its link token in order to be able to associate it with each self issued Verifiable Credential. The link token is provided by a Trusted Third Party (TTP), with which the *TAF* can establish a trusted and authenticated channel.

Story-XII: As the MBD I want to self-issue a valid mis-behaviour report VC for the data that is being sent.

Objective: The objective of Story-XII is for the Misbehaviour Detection component to be able to provide anonymous and unforgeable signatures, over a misbehaviour report formed with the data collected from the facility layer of the vehicle.

Motivation: A mis-behaviour report from the *MBD* is one of the methods used to provide trustworthiness evidence over a vehicle. More specifically, based on pre-defined policies the facility layer will request from the Mis-behaviour Detection component to construct a Mis-behaviour Report from the evidence collected by the zonal controllers and their underlying ECUs. The constructed Mis-behaviour Report is sent to the *TCH* and is included to a T-CPM

message. Because the Mis-behaviour report is shared with other components/entities, it needs to be created in a verifiable manner. To be more precise, for the finalisation of the Misbehavior Detection report the MD calculates a digital signature over the mis-behaviour checks for a particular observation and then associates the signature with a link token in order to construct its verifiable credential. Moreover, as the Mis-behaviour Detection report is included in a T-CPM message, a lot of privacy implications are raised, as any external entity should not be able to extract any information about the identity of the signing *MBD*. To resolve this issue, the *VC* provided by the Mis-behaviour Detection component needs to be anonymised.

Requirements: For this user story to be feasible the Mis-behaviour Detection (*MBD*) component needs to have a mapping of all the devices keys, that corresponds to communication integrity and data integrity. More specifically, the Mis-behavior Detection component expects encrypted CAM/CPM messages, with the keys that *IAM* shared during the boot up phase of *MBD*, ensuring the integrity of the received data. Moreover the MD needs to provide its own anonymised signature. That being said it has to have successfully completed the secure enrollment phase, where a TTP has issues a link token for a specific *MBD*. The anonymous signature along with the link token, are used to construct the *MBD*'s Verifiable Credential.

Story-XIII: Verify a trustworthiness claims VP provided in a CAM/CPM message.

Objective: As the receiver of a CAM/CPM message containing a trustworthiness claims *VP*. I wish to verify the integrity of the evidence that I have received.

Motivation: Trustworthiness claims *VPs* are included in some, or all, of the CAM/CPM messages to provide trustworthiness claims, produced from the trustworthiness evidence collected by the *AIV*, *MBD*, IDS and *TAF*. Each of the aforementioned components are contributing using their own secret key to the Trustworthiness claims, solidifying the validity the data that are included. These claims are eventually signed by the *IAM*, using a *PKI* pseudonym and then is sent to all neighbouring vehicles and the *MEC*. The contribution of all the relative components has to be verified by the T-CAM/T-CPM message consumer, in order recreate the Trust chain that was instantiated in the vehicle of the T-CAM/T-CPM message provider enabling the calculation of the referral trust assessment (on the vehicle, or the *MEC*). It goes without saying, that without this verification procedure the *TAF* cannot assess that the T-CAM/T-CPM message provider has used the appropriate keys and was in a correct configuration or that it matched the requirements of any other aspect of security that was attested for this particular Trust assessment.

Requirements: The CAM/CPM message consumer needs to check the pseudonym used to sign the *VP*, so as to assure that it is indeed a *PKI*-issued pseudonym. Furthermore, upon the successful verification of the pseudonym signature, T-CAM/T-CPM message consumer needs to verify the Verifiable Presentation (*VP*) as well. This verification process is broken down in two phases. The first phase should be verifying the Verifiable Presentation (VP) as a whole. The second is initiated in case the first phase fails to be compiled successfully. In this phase the T-CAM/T-CPM consumer tries to verify the Verifiable Credential provided

by the *TAF* and *MBD*, in order to be able to trace back to the entity that failed to sign with its secret key. This way, due to the associated link tokens, authorised entities can extract the identity of the component that failed to create a valid contribution for the T-CAM/T-CPM message and act accordingly.

4.4 User Stories for Re-Establishing Trustworthiness

Due to bugs, it may happen that parts of a system are compromised. In this case, it is important to support recovery of unaffected system parts wherever possible. One tool for this recovery is the migration of a critical CCAM application to another *ECU*. The goal of this story is to salvage the protected state of the *TEE* (assuming it was not compromised) and re-establish a clone of this *TEE* on another *ECU*.

Story-XIV: Migration of a CCAM application from one ECU to another.

Objective: When the *IAM* is notified by the *TAF* of a change in the trust level of a device, hosting a CCAM service, that puts it below the RTL, he triggers the migration of the CCAM's service to an *ECU* with the appropriate RTL.

Motivation: As in *CONNECT* we are moving towards zero trust architecture, we need to ensure that even when a complete *ECU* no longer meets the required trust level (RTL), the system could recover its trustworthiness level (i.e., ATL). The goal is to then migrate a critical CCAM component/service from a degraded *ECU* to another *ECU* that is still trustworthy. As other ECUs capable of hosting the same application meet the Required Trust Level, we choose to migrate the compromised ECU's application to the one that fits best and still has sufficient trust. More specifically, the *TAF* informs the *IAM*, as he is the Root node of the Trust tree of each vehicle, that the ATL of a device hosting a CCAM service doesn't meet the RTL, in order for the *IAM* to take the necessary actions. That being said, the *IAM* has an interface dedicated for calculating the policy under which a CCAM service can be migrated to an other ECU. Such policy contains the original and target ECUs of the migration, the security requirements under which the migration needs to be instantiated (which cryptographic protocol is going to be enforced a lightweight Diffie-Hellman or an Attribute-Based Encryption scheme) and which parts of the application needs to be migrated.

Requirements: For this user story to be feasible, the *IAM* needs to know the Required Trust Level (RTL) of all the devices that hosts a CCAM service. Similarly the Trust Assessment Framework needs to be configured to send notifications regarding trust level changes to the *IAM*. Moreover, for the critical information of the CCAM application (i.e., integrity/communication keys), the *IAM* has a dedicated interface that either specifies a set of attributes that are shared between the list of ECUs that allow migration, so as to be encrypted with ABE or it initiates a Diffie-Hellman between the engaged ECUs. When choosing to encrypt the migratable data with ABE, the *IAM* needs to define dedicated attributes, which will allow the migration of this service to be compiled just once, thus, preventing replay and DoS attacks.

4.4.1 Binary Instrumentation & Device Data and Execution Flow Monitoring

Story-XV: As the AIV, I want to make sure on the freshness of the monitored trustworthiness evidence.

Objective: The Attestation Integrity Verification component should be able to prevent numerous types of attacks. One of these attacks, is called replay attack where an adversary sends to a verifier evidences from previous successful executions of attestation tasks.

Motivation: As described in the deliverable D2.1 the Trust Assessment Framework calculates a Trust Opinion over an attestation report provided by the *AIV*. This attestation report is based on the attestation/trustworthiness evidences that the *AIV* collected from all the devices corresponding to this specific Trust Assessment Request. For this purpose, in order to achieve a real time and accurate depiction of all aspects of Trust defined by the Trust Assessment Request, the *AIV* has to be strict in the calculation of the attestation report, ergo the verification of the attestation/trustworthiness evidence has to meet all the requirements defined in [10, Chapter 6]. One main requirement, defined by the IETF as well, is the freshness of attestation/trustworthiness evidence, in order for the verifier (i.e., the AIV) to be able to verify not only the correct creation of the digital signatures that he collected, but also under which session they were created.

Requirements: For this user story to be feasible, all ECUs ,both A-ECUs and S-ECUs (not the N-ECUs as they are not capable of providing attestation evidence at all), has been configured correctly. More specifically, all capable ECUs should have successfully completed the Secure on Boarding initiated by the Identity Authentication Management component, in order to set up their secret keys and the appropriate key restriction usage policies. Moreover, the AIV needs to be able to create and map nonce values for all attested devices for signing and verification purposes. It goes without saying that all the aforementioned must be instantiated or supported by the underlying Trusted Computing Base of each device.

4.5 User Stories for Workload Protection Using a Trusted Execution Environment

The following stories specify the desired protections provided by a TEE and the security-related services that are provided by a TEE.

Story-XVI: Protection of Workloads on ECUs.

Objective: As an *OEM*, I want to protect security-critical applications in the vehicle against unauthorised modification and information leakage by executing it inside a *TEE*. The following non-functional objectives should be guaranteed:

1. The integrity of the application and the integrity and confidentiality of its state must be protected.
2. TEE-protected applications must keep their code paths/logic integrity-protected and unmodified at all times.
3. TEE-protected applications must keep the data integrity-protected and confidential at all times.
4. Selected TEE-protected applications must transparently receive encryption keys and other secrets from remote applications/users, in order to e.g. decrypt input files and encrypt output files.

Motivation: To protect critical applications against a potentially untrusted or compromised software on the vehicle, we require hardware protection for critical workloads.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. The computer must be equipped with hardware-based Trusted Execution Environment (TEE) capabilities.
2. No requirements are imposed on any software outside the TEE (including but not limited to the container framework, the operating system, and the hypervisor.

Story-XVII: Integrity-verification of TEE Applications.

Objective: As an *OEM*, I want to ensure that a TEE only launches a given application if it was able to verify the integrity of the application.

The following non-functional objectives should be guaranteed:

1. The OEM must be enabled to authorize a given application and the TEE must be able to verify the integrity of this application.

NOTE: In this user-story, I mean the Gramine's `gramine-sgx-sign` tool.

Motivation: A security-critical application is deployed along a supply chain and can be modified in transit. To protect against this risk, we require end-to-end integrity guarantees for applications to be executed within a TEE. This is usually achieved using digital signatures.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. The computer must be equipped with hardware-based Trusted Execution Environment (TEE) capabilities.
2. The OEM must have access to a corresponding signing key and tools for signing a trusted application.

Story-XVIII: Attestation of Applications running in a TEE.

Objective: As a *OEM*, I want to be able to remotely validate the integrity of an application that is executed within a *TEE*.

The following non-functional objectives should be guaranteed:

1. The attestation service ensures that the OEM obtains a correct and fresh cryptographic checksum of the application.
2. The correctness and freshness of the checksum is guaranteed by the hardware and does not depend on any other software component.
3. As a *Application Developer* I want to rely on existing TEE-attestation solutions that seamlessly generate TEE-specific attestation evidence and verify this TEE-specific.
4. TEE-protected applications must create TEE-specific attestation evidences to prove themselves to other applications.
5. TEE-protected applications must verify the trustworthiness of other TEE-protected applications.

Motivation: Since the application consists of software that can be changed.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. The computer must be equipped with hardware-based Trusted Execution Environment (TEE) capabilities.

Story-XIX: Support for Development and Debug.

Objective: As an *Application Developer* of applications for a *TEE* (on the vehicle main computer, in an *ECU*, on the *MEC*, or on other platforms supporting a *TEE*), I want to develop and test the applications (such as TEE Guard, *AIV*, *TAF*) in a familiar non-TEE-based environment and then seamlessly deploy these applications as stated in Story-XX. The following non-functional objectives should be guaranteed:

1. For seamless migration, support for debug and test should not disrupt the normal development and test processes deployed today.
2. During debug, the security policy of the TEE is not enforced and thus applications should not yet contain critical secrets.
3. The story should also be enabled for TEE-applications packaged in containers.

Motivation: Deploying an application within a TEE reduces flexibility since, e.g. a manifest has to first be signed with a specific key. During development, user friendly deployment and test is important to maintain the productivity of the developer.

Requirements: This user story does not impose any requirements since Story-XIX should work on any development machine.

4.6 User Stories for Creating a Trusted Execution Environment

The following stories describe how a TEE is designed and how existing applications can be migrated into it.

Story-XX: Migrating an Application to Gramine with Gramine tools

Objective: The *Application Developer* converts an existing Linux-style application into an application bundle that can be executed within Gramine ("graminize an application" in the following). The following non-functional objectives should be guaranteed:

1. Security: After graminizing an application, the application can only be executed with the specified security guarantees enforced (i.e. confidentiality and integrity are usually protected using the *Intel Software Guard Extensions (Intel SGX) TEE*.
2. Ease of use: Graminizing an existing application should involve minimal effort. Porting the application to run inside *Gramine* SGX environment must involve minimum engineering effort. This requirement is satisfied by *Gramine* as follows: the application does not need to be re-written or re-compiled (an original binary can run inside *Gramine* SGX). The only effort required is writing a corresponding manifest file that configures the security posture of the application, the lists of files the application can access, and the properties of the SGX enclave in which it will run.
3. Configurable protection: Developers should be able to configure the protections that are enforced for a given application.
4. Ease of deployment: A graminized application should be easy to deploy and execute. Porting the application to run inside *Gramine* SGX environment must provide flexibility to fine-tune application and SGX-enclave parameters and to block or allow specific files to be accessed. This requirement is satisfied by *Gramine*'s manifest syntax, which has more than 50 different knobs and allows very fine-grained management of files.
5. Porting applications to run inside a TEE must involve minimum engineering effort.
6. Porting applications to run inside a TEE must provide flexibility to fine-tune application parameters and configurations, to block or allow specific files to be accessed, to block unused sub-systems, etc.

The main task of Graminising an existing application is to write a correct manifest file. After the creation of the manifest file, the application developer can invoke tools to generate Gramine- and SGX-specific files.

Motivation: By packaging applications within Gramine, they can be executed in an Intel SGX enclave and can thus benefit from the hardware protection of Intel SGX (or other backends).

Requirements: This user story does not impose any requirements since Story-XX should work on any development machine.

Story-XXI: Configuring the Security of a TEE

Objective: As an *OEM*, I want to define the security posture (aka security policies that are enforced by the TEE) for each individual application.

The following non-functional objectives should be guaranteed:

1. The OEM can specify specific files as read-only and integrity-protected, some files as transparently encrypted with specific keys, and some files as completely inaccessible.
2. The OEM can specify sub-systems that are required by each application. E.g., I want to disable spawning children if the application never uses this functionality, or to disable eventfd signalling if the application never uses this functionality.
3. The OEM can hard-code the command-line arguments and/or environment variables passed to each application, to reduce the number of possible control paths taken by the application.

Motivation: One important goal is to reduce the *TCB* of the application that is executed in a *TEE*. The goal is to only require trust into the TEE hardware, other TEE services, and specifc files and services with well-defined security guarantees. This is achieved by specifying the TCB in a so-called *Manifest File*.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. The computer must be equipped with hardware-based Trusted Execution Environment (TEE) capabilities.

4.7 Stories for Upgrading and Migrating Protected Workloads

We now describe capabilities to upgrade software or offload TEE workloads.

Story-XXII: Upgrading the TEE Software.

Objective: As the *CCAM* application, I want to migrate application state from one TEE on a given computer to another (potentially upgraded) TEE on the same computer.

The following non-functional objectives should be guaranteed:

1. The migration can migrate from one software version to a later version.
2. While upgrading one TEE instance, it must be ensured that only one new instance is started and no more than one instance is authorized as the master/reference at any point in time.
3. It must be ensured that the old TEE is blacklisted and will no longer be seen as the master/reference.

4. Both version must be authorized by the OEM.
5. During this migration, integrity of program and state and confidentiality of the state must be protected.

Motivation: To introduce new features or fix bugs, the software that is executed within a TEE sometimes needs to be updated. This story enables secure upgrade - from one authorized version to an authorized successor version. For this updgrade it is critical to ensure that software and state remain protected and can only be importanted into a TEE that is secure enough.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. The computer must be equipped with hardware-based Trusted Execution Environment (TEE) capabilities.
2. The OEM has authorized the upgrade.
3. When receiving a SIGINT or SIGTERM signal, the applications running in the TEE can save their state into a set of specified files. Note that for C, this can be achieved automatically, stateful software in languages like GOLANG or Python require explicit support for saving their state before exiting.
4. The applications are able to load their saved state from a set of files at startup.

Story-XXIII: Migrating the TEE Software.

Objective: As a *CCAM* application, I want to migrate a running TEE-protected application's state from one computer to a different computer while protecting the state during this transition.

The following non-functional objectives should be guaranteed:

1. The migration can migrate from one machine to another machine while protecting state and sofware.
2. Both version must be authorized by the OEM.
3. During this migration, integrity of program and state and confidentiality of the state must be protected.
4. Upon migrating from one TEE instance to another TEE instance, it must be ensured that only one new replica is started.
5. Upon migrating from one TEE instance to another TEE instance, it must be ensured that the old replica is terminated.

Motivation: One objective of *CONNECT* is to allow workload offloading from the vehicle to the *MEC*. This can include security-critical applications that are protected by a TEE. To allow this new feature, we plan to extend the Gramine Library OS to allow for protected migration of workload.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. The computer must be equipped with hardware-based Trusted Execution Environment (TEE) capabilities.
2. The application to be migrated is required to support saving its state into files at exit and restoring the state from files at start-time. For C, this can be automated. Other languages (GOLANG, Python) may require explicit support.

4.8 User Stories for Protection of Workloads on a Mobile Edge Cloud (MEC)

The following user stories specify the desired behaviours of TEEs on Mobile Edge Clouds (MEC). The MEC can host container workloads on a security-enhanced Docker infrastructure that is designed and implemented by *CONNECT*. Services hosted on the MEC can provide services that are compute- or communication-intensive. One example is services that coordinate collaboration between multiple vehicles such as automated management of an intersection. Another option where the MEC proves useful is offloading compute- or communication-intensive workloads from a vehicle by migrating them to the Edge infrastructure. One example for such a service is re-training of Machine Learning models based on vehicle data and remote data.

A core concept that we use are containers that are secured using a hardware TEE. Our goal is to allow users to design a normal Docker container image and then convert this image into a security-enhanced container image. When a container is created and started using this image, it is automatically executed in a TEE like Intel SGX. Still, by enhancing the original container, our goal is to enable seamless Docker-style management. I.e. the infrastructure should be enabled to create, start, stop, pause, and migrate a given security-enhanced container.

We now detail the individual user stories that outline the functionality required by the MEC.

Story-XXIV: Creating a docker image that implements event-triggered management of its own state.

Objective: As the *Application Developer*, I want to be able to design and implement containers that can (a) be managed by a Kubernetes cluster while (b) keeping a consistent state that can later be security-protected. The following non-functional objectives should be guaranteed:

1. When Kubernetes signals the container to shutdown (e.g. using SIGTERM), the container should save any critical state to a set of well-defined files and then exit gracefully.
2. When Kubernetes re-starts a container on a given pod that is part of a cluster, the container is required to restore its state and any critical network connections given the well-defined set of files containing a prior saved state. This may include checking and repairing the state in some cases (see example below).
3. When Kubernetes re-starts a container on a different pod/cluster, the container is required to restore its state and any critical network connections given the well-defined set of files containing a prior saved state.

4. When importing a state, the software verifies that the state has been exported/saved by a compatible software version. I.e. exporting and then importing the state maintains its consistency.
5. Optional: Rollback-prevention: Once the container has been upgraded to a given software version, it will no longer start with any outdated version.
6. Optional: Singleton guarantees: Only a given set of authorised instances are allowed to be active at any point in time.

Motivation: The (security-enhanced) container infrastructure that we envision will start, stop, pause, and migrate given containers. For stateless containers, this is easy since they can be killed and restarted without loosing any interesting state. However, many security-critical applications such as a keystore require state to be kept. Managing this state over the life-cycle of the container is important for the security of the contained applications.

Some programming languages (e.g. C) allow hibernation of a running application into a file. For other languages (e.g. Golang, Python) this is not guaranteed to work. As a consequence, we will offer developer developer support and guidance (for Golang, Python, C) that allows developers to design and implement container images that support saving and restoring its state explicitly.

Assume that a container contains a key store for cryptographic keys (that constitute the state). To be manageable, the container is required to process a SIGTERM signal at any time. Upon receiving this signal, it will ensure that the keystore on disk is up-to-date and will exit. While this sounds simple, there are corner cases that require transactional behaviour in a distributed system. E.g. if a new key is issued, the container may be killed at any time (e.g. due to a power failure). In this scenario, one still aims to achieve consistent overall behaviour [26]. This can mean that a new key is first saved to disk before signaling success to a key management server.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. The developer has specified the essential set of files that need to be saved and restored.
2. Upon SIGTERM the application saves its state into those files.
3. Upon restart, the application restores its state from those files.
4. Optionally, the application may use an external service to prevent cloning and rollback of the given service.
5. For rollback protection, an external monotonic counter service is required. The software inside the container can ensure that it only starts if it contains the latest software version. The design of anti-rollback will be outlined in 6.3.2.
6. For singleton instances, a external trusted service is required to ensure that only one authorised instance can be started at any point in time.

Story-XXV: Converting an existing Docker *image* into a protected Docker image.

Objective: As an *Application Developer*, I want to transform an existing container image into a new container image that can be used to create a container that is executed in a TEE-protected environment that ensures that container integrity and confidentiality is protected at run-time.

The following non-functional objectives should be guaranteed:

1. Porting a Docker container to run inside *Gramine* SGX environment must involve minimum engineering effort.
2. Deployment of the "graminized" Docker container must involve minimum engineering effort. This requirement is satisfied by the GSC's design – it does not modify any properties of the original Docker image. Thus, the GSC-generated Docker image may be deployed in the exact same way as the original Docker image.
3. The resulting image includes a manifest that defines the container security properties and required metadata for integrity and confidentiality protection.
4. The desired security is configurable.

This user story will be implemented using the GSC tool as follows: the original Docker image does not need to be re-written or re-built. The only effort required is writing a *minimal* manifest file that contains the desired properties of the SGX enclave. The GSC tool expands this minimal manifest file to a proper *Gramine* manifest, by querying the relevant information from the Docker image metadata (like the lists of files). This ensures that a proper manifest is created that ensures that the file containing state is encrypted/sealed.

One important point to consider during this conversion is "what key to use for sealing the state". There are two potential cases:

Container bound to CPU: If we plan to bind the container to a specific CPU, then we need to tell Gramine to auto-encrypt/seal the given file using the `MRenclave` key. This key is specific to the given SGX instance and does not allow migration.

Container migrateable: If we plan to allow migration of state between different machines (under the condition that the container is authorized by the same signer), then we need to tell Gramine to auto-encrypt/seal the given file using the `MRsigner` key.

Motivation: Vehicles may want to offload critical tasks (packaged as docker containers) onto MECs. In this scenario, the MEC may not be fully trusted. To enhance the security guarantees, TEE-protected execution should be enabled for the the given containers.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. We require that the container to be graminized handles its state properly. I.e. upon SIGTERM it will save its state into the specified set of files and upon start it restores its saved state (see Story-XXIV)

Story-XXVI: Creation and protection of containers on the MEC based on a graminized image.

Objective: As a MEC service provider, I want all hosted security-critical CCAM applications (running in the deployed containers) to be protected by hardware-based TEE, keeping the original CCAM service provider agnostic to the underlying container-protection mechanisms.

The following specific goals are relevant:

1. Automated tools (compatible with container management technology) must be available to support the secure *creation* of TEE-protected containers at the MEC in an efficient/seamless way (see section 3.2.2).
2. The integrity of the application and the integrity and confidentiality of its state to be protected.
3. MEC service *deployment* to be feasible through TEE-protected application containers in an orchestration environment such as a Kubernetes cluster.
4. Continuously monitoring of the integrity of the applications across the lifetime of the associated container.
5. Execution of protected application containers to entail no risk for leakage of sensitive private data

Motivation: Automotive (or other type of) applications running on the MEC, offering critical data to the end-user, need to be protected against a potentially untrusted or compromised MEC host environment.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. A software mechanism to build confidential containers manageable by orchestration environments (such as Kubernetes).
2. The MEC infrastructure (host) nodes to be equipped with hardware-based Trusted Execution Environment (TEE) capabilities.
3. No further requirements are imposed by the user story on any software outside the TEE (including but not limited to the container framework, the operating system, and the hypervisor).

Story-XXVII: Life-cycle management (LCM) of a confidential container at the MEC to be carried-out in an efficient way (i.e., similar to the legacy containers' LCM).

Objective: The LCM operations of a legacy as well as a confidential container to be carried-out at the MEC with equal programming effort and similar performance.

The following specific goals are relevant:

1. TEE-protected containers hosted at the MEC, to be managed seamlessly *during their lifecycle* by an application container management environment (e.g., Kubernetes cluster) – (see Section 3).

2. LCM operations (e.g., scaling in/out capabilities for the TEE-protected containers) should not negatively affect the quality of the provided service (QoS), nor influence the involved SLA.
3. Confidentiality along the LCM operations of TEE-protected containers should be ensured seamlessly (no extra effort compared to the legacy case).
4. LCM operations of (the resulting) confidential containers to entail no risk for leakage of sensitive private data.

Motivation: To practically fulfil the need for protecting (automotive) sensitive workloads, facilitated by confidential containers, the imposed programming/performance overhead during their LCM operations should be minimal; otherwise, confidential containers would be of limited applicability.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. Optional: Prevention of container cloning and rollback.
2. An application container needs to implement secure event-triggered management of its state.
3. A confidential (e.g., graminised) container needs to ensure that its state remains protected across all LCM operations.

Story-XXVIII: Upgrading a protected container on the MEC.

Objective: The *Cloud Administrator* is enabled by the *Application Developer* to securely upgrade the software inside a security-protected container. The following non-functional objectives should be guaranteed:

1. After upgrading, the container continues to operate and has imported the state of the prior software version.
2. Outside the enclave, the state remains encrypted and integrity-protected at all times.
3. Only software upgrades that are authorized by the signer can import a given state. Technically this means that the signer of the enclave (a) signed the new software version and (b) enabled the new software version to import state from a given version of another software.[1]
4. Optional: Only one instance of the container exists at any point in time.
5. Optional: Rollback-protection: After the upgrade, the system prevents launching older versions of the container.

The actual upgrade of a container includes the following steps:

1. The original container dumps its state and seals it to the `MRsigner` key. This exports includes the software and version othat is exporting the state.

[1] The sealing to `MRsigner` only ensures that the same authority signed both containers. Extra mechanisms need to ensure version compatibility.

2. The *Cloud Administrator* create a fresh container using an upgraded image.
3. The *MEC* start the fresh container - inputting the sealed state to allow the container to initialise itself.
4. Optional: The container verifies singleton requirements and exits while discarding its state if they are not satisfied.
5. Optional: The container verifies that the latest software version is present (to prevent rollbacks) and exits while discarding its state if an outdated software has been used.
6. The container checks the compatibility of the versions and then imports its state.

Motivation: The *Cloud Administrator* is responsible for managing capacity of the *MEC* and keeping the installed software up-to-date. As part of this responsibility, the admin may want to upgrade the software in a container while maintaining the security and consistency of the contained state.

Requirements: This user story imposes the following requirements from the underlying components and services:

1. We assume that the container can manage its own state (see Story-XXIV).
2. For rollback protection, an external monotonic counter service is required. The software inside the container can ensure that it only starts if it contains the latest software version. Anti-rollback will be detailed in Section 6.3.2.
3. For singleton instances, a external trusted service is required to ensure that only one authorized instance can be started at any point in time. Guaranteeing singleton execution will be detailed in 6.3.1.

Story-XXIX: Migrating seamlessly a confidential container in a different host, or offload a task from the vehicle to the MEC.

Objective: The migration process requires certain assurances as it pertains to secure state transfer from a host A to a host B. To achieve that, a fresh target container should be established, creating a new instance from the same image but without any existing state, while key agreement mechanisms should be defined among the two hosts, the source and the target system, to protect the communication. When the state has been successfully transferred from the target system, the old container should be terminated in the source system. The same mechanism is also used for task offloading between vehicle and graminised Digital Twin (DT)-*TAF*.

Motivation: Whenever an application/enclave running either on the vehicle or the MEC, has to be migrated to a different host (i.e., Vehicle to Vehicle, ECU to ECU and Vehicle to MEC) based on its policies (i.e., low response time), this process should happen in a seamless yet secure manner, while maintaining state of the container. In addition, some cases may not require the migration of the entire container but the offloading of a specific task from the vehicle to the MEC (i.e., a CCAM application or the Digital Twin *TAF*). The key used for protecting the authenticity of the state of the container is linked to the hardware.

To achieve this, *CONNECT* proposes the introduction of an auxiliary application, the Migration Service, which as defined in Chapter 2, is part of the *TCB*. This application is devoted

to the mediation of the migration process, in order to i) verify the authenticity of the state of the process to be migrated (i.e., leveraging the MR Signer key), and ii) assure the confidentiality of the communication between the host and the target enclave (i.e., utilising an enclave to enclave key).

Requirements: For this user story to be feasible there are certain steps that have to be accomplished. To begin with, dedicated (symmetric) keys are leveraged to ensure both the authenticity of the migratable state as well as the confidentiality and authentication on the communication exchange, between the host and the target enclave. The type of key, utilised to ensure the authenticity of the state of the migratable function, is called the MR Signer key and it is linked to the hardware (i.e., Intel SGX). Furthermore, preventing rollback mechanisms is crucial to avoid the migration of obsolete container versions. This is essential to ensure that only the most up-to-date and secure containers are migrated, addressing potential security risks associated with the use of outdated versions.

Apart though from the keys that are necessary to ensure protection, several steps are needed to facilitate this migration. These steps include the deployment of a trusted mediator (i.e., the Migration Service deployed within the *TCB*), which facilitates; thus provides guidance and orchestrates the secure migration process. Additionally, the target enclave should be equipped with a vanilla application, from the backend application system. Lastly, to cover the case of task offloading for trust calculation migration (i.e., Digital Twin *TAF*, including the Federated TAF as elaborated upon in D3.2 [15]), synchronisation of the trust model/state between two enclaves is needed.

Story-XXX: Validate the integrity of a Secure Container and Workload Identity of the deployed application.

Objective: All confidential containers should be verified in terms of integrity and workload identity for the deployed application. This task is performed by the consumer (i.e., vehicle), desiring to access a trustworthy service. Towards this direction, the public part of the Kubernetes key (i.e., as released by the Kubernetes Key Management System during the secure launching) should be bound to the certificate (also released by the Kubernetes Key Management System during the secure launching). This binding ensures the correctness of the deployed container. Furthermore, to enable the validation of identity on the running workload, the key of the enclave (i.e., for example the AIV, which exports trustworthiness evidence) is bound to the certificate.

Motivation: In scenarios involving multiple instances of the same CONNECT service being deployed in the same MEC infrastructure, it is crucial to verify that the received response originates from the intended container. This verification process ensures that the results align with the specific instance of the application intended to perform the task. CONNECT establishes a robust mechanism to validate the source of responses, maintaining the integrity and accuracy of outcomes in such scenarios. This mechanism provides protection against attacks such as replay attacks, which could be launched by a malicious enclave, intercepting traffic. To be more precise, on the MEC there can be instantiated multiple AIVs, deployed in separate containers in order to provide Trustworthiness evidence to the Vehicles TAF. To this end, it is crucial to provide an in-toto Attestation toolkit, so as to enable every working container, in this case an AIV container, to provide Verifiable Evidence both

over the Identity of the MEC, that is handling the offloading challenge, and over the identity of the container that carried out the workload.

Requirements: For this user story to be feasible the MEC ignites and has to successfully complete the enhanced Secure Launching of Confidential Containers, an extension of the Secure Launching of Confidential Containers (see Chapter 2 Section 2.3.1). More specifically, when launching a container that needs to provide such verifiable evidence, upon completing the standard algorithm for the Secure Launching of Confidential Containers, the container proceeds to complete the enhanced version pf the Secure Launch. To be more precise, when a container acquires its Kubernetes secret key, the container sends to the Kubernetes master compute node, through an encrypted and authenticated channel, the public part of the containers Attestation Key. The Kubernetes master compute node, creates a certificate over a mapping of the Kubernetes secret key and the Attestation Public key. This certificate is wrapped in the form of a Verifiable Credential so as to append more attributes if necessary and eventually the container can provide Verifiable Evidence that is bound to its identity. The Verifiable Evidence produced by the container is then signed with the PKI by the IAM, enabling MEC identification as well.

Chapter 5

The *CONNECT* Cryptographic Protocols for Enabling Dynamic Trust Assessment

Building on top of the previously described functional specifications required to enable *CONNECT* to provide CCAM-wide trust measures, we now give details of the first set of trust extensions that will be used to establish these **trust-centric automotive networks**. These trust extensions will enable the *CONNECT* TAF to provide dynamic trust assessment of all of the nodes (HW and SW) comprising the CCAM ecosystem. Recall that *CONNECT* extends the stand-alone vehicle domain to safe and security solutions distributed from Vehicle to MEC and Cloud facilities so as to support the envisioned automation of *connected vehicles*: This vision is enabled by the vehicle's communication with other entities formulating the Vehicle-to-everything (V2X) landscape. The resulting "*CCAM continuum*" paradigm seeks to seamlessly and securely combine the available hardware and software (from the in-vehicle sensors to the MEC virtualised infrastructure) to support the deployment and operation of certified (using ISO/SAE 21434 [1]) CCAM functions. Whereas the collective consideration (treatment) of the continuum resources presents opportunities for increased performance and a higher degree of automation, the individual Software (SW) and Hardware (HW) infrastructure may exhibit diverse yet dynamic trust states; and when those infrastructures are brought together to make-up the continuum strong security mechanisms need to also be deployed for asserting to the correct state of all these functional assets.

For this to be achieved, it will require secure life-cycle management of:

- the **identification of the type of trust evidence** to be collected.
- the **trust assessment of any deployed CCAM function** (based on the defined trust models capturing the relationships and inter dependencies of all internal components, as described in D3.2 [15]).
- the **creation of mechanisms for capturing the devices (HW and SW) trust levels and adapting to changes**, anchored to decentralised Roots-of-Trust, and subsequently elevated to the continuum level.
- the **devices in the system to be equipped with an underlying Trusted Computing Base** which will offer secure measurement and reporting of a devices' (integrity) state focused on the provision of the necessary trustworthiness evidence for that device.
- the ability to share trustworthiness evidence both: (i) with **any component belonging to the same domain** (i.e., in-vehicle E/E domain), without any privacy protection, and (ii) with

external TAF instances (of other vehicles or trust assessment services instantiated on the MEC) with the necessary level of abstraction and strong privacy safeguards so as to avoid the identification and linking of trust-related evidence to their (vehicle) source that can potentially lead to further implementation disclosure attacks [11].

More specifically, we focus on the design of a novel set of *CONNECT* crypto and attestation primitives, supported on the vehicle side, for ensuring the **trustworthiness of the data** they provide - both as it pertains to kinematic and perception data consumed by the deployed CCAM functions, but also to (attestation) evidence depicting the "*trust state*" of components in the system. *CONNECT* Trustworthiness Claims (TCs), provide evidence (encoded as Verifiable Credentials, as detailed in D5.1 [11]) of the devices' correctness [22]; *from their trusted launch and configuration to their runtime attestation of both behavioural and low-level execution properties that act as sources of trust to be used by the local TAF.* Going beyond the current state-of-the-art in remote attestation schemes, *CONNECT* builds a harmonised TCB that exposes a well-defined TEE Device Interface (TDI) linked to the runtime monitoring of an extended set of device characteristics that serve as evidence for the deployed trust models. This translates to the governance of the security elements (e.g., TEEs), instantiated in each capable node, for the monitoring of only that evidence needed for the evaluation of a specific trust property of interest. For instance, *CONNECT* Enhanced Configuration Integrity Verification (CIV) (Section 5.4) allows for the provision of strong security claims on the integrity of the target device while overcoming the barriers of **configuration privacy and scalability**. This is achieved through the introduction of trusted computing abstractions, called **policy-restricted attestation keys** [21, 20], that dynamically restrict the use of a devices' attestation key to policies depicting the expected state of the device; i.e., holding the reference value of the expected configuration measurement of the binary to be attested. This allows for the (remote) verification of the devices' runtime configuration conformance without revealing any (runtime) configuration information. Such a capability is further enhanced with additional privacy abstractions that allow the sharing of trustworthiness claims with "*external*" entities (i.e., outside, for instance, of the target vehicle) in a zero knowledge manner that can be used for verifying the trust state of the vehicle's operational landscape, without disclosing any sensitive information about the state of any (in-vehicle) sensor and ECU that might leak details valuable to an adversary in their attempt to compromise the system. This feature is based on a novel crypto scheme that protects **trustworthiness claims with zero-knowledge proofs (signatures)** for constructing presentations. This focuses on asserting to the trust attributes of a vehicle (e.g., integrity, resilience, etc.) modelled as abstractions. These abstractions are associated with adequate proof of possession of the corresponding attribute attestations, without revealing any details of the attribute values. Towards this direction, as will be detailed in Section 5.5, *CONNECT* leverages **Threshold Direct Anonymous Attestation (DAA)**.

A common denominator to all these crypto elements is the successful construction of the necessary keys and runtime security controls for safeguarding the integrity of the application and attestation processes themselves. This is achieved through *CONNECT*'s enhancements to the (OEM adopted) enrolment process of an ECU into the in-vehicle network, enabling the verification of the underlying RoT of the ECU and all other primitives. In turn, this facilitates the subsequent trust evaluation and quantification of all internal (SW and HW) components.

CONNECT User Stories	Relevant Protocol & Crypto Scheme
Story-I : Configure a device ready for installation into the vehicle. **Story-V**: Secure on-boarding of an *ECU* into the vehicle.	Sections 5.3.1 and 5.3.2, and in particular Figures 5.1 and 5.3, depict the process followed during the enrolment of an ECU into the in-vehicle network and the construction of all necessary crypto primitives (including both application-and security-related keys) needed to support the secure interactions of the target component with the other elements belonging in the same domain (e.g., same "zone" capturing a specific set of vehicle functionalities; e.g., breaking). All CONNECT security-related keys (policy-restricted attestation keys) are also created for allowing the secure and privacy-preserving sharing of trustworthiness evidence.
Story-VII: Obtaining and verifying trustworthiness (attestation) evidence from the vehicle's devices. **Story-XV:** As the AIV, I want to make sure of the freshness of the monitored trustworthiness evidence. **Story-XVIII:** Attestation of applications running in a TEE.	While S-ECUs will use the standard attestation mechanisms, A-ECUs (including zonal controllers) will use the *CONNECT* Enhanced Configuration Integrity Verification (CIV) scheme described in Section 5.4 for allowing the provision of verifiable attestation attributes on their correct configuration state. These attributes, constitute one of the core trust pillars/sources based on which the TAF operates. Guarantees on the "*trusted configuration state*" of a device are provided through proofs of conformance; i.e., signatures constructed through the use of attestation keys protected by appropriate **key restriction usage policies** predicating their use *if and only if* the host application is at a correct state. In both cases nonces will be used to ensure freshness.
Story-XI: As the TAF I want to self-issue a valid trust opinion VC based on the relevant trust sources. **Story-XII:** As the MD I want to self-issue a valid misbehaviour report VC for the data that is been sent.	Upon reception of the trustworthiness evidence, as trust sources to the TAF (in the context of *CONNECT*'s envisioned use cases, such evidence stem from the MD Service as well as the deployed attestation security controls), TAF should be able to construct a Verifiable Credential encoding the calculated trust opinion and Actual Trust Level (ATL for the target CCAM function) bound to conformance proofs that the TAF itself is at a correct state. This is achieved by equipping all *CONNECT* security components with policy-restricted attestation keys (Section 5.4.2.2) predicating their use for signing on the configuration correctness of the host application (i.e., TAF).
Story-X: As the TCH I want to self-issue a valid VC for attestation evidence from the vehicle's devices.	Sections 5.5 and 5.6 describe how the trustworthiness claims will be generated and anonymously signed. As described in Section 5.1, one of the core goals of *CONNECT* is to enable the node- and data-centric trust quantification (across the entire CCAM continuum) in a secure/verifiable and privacy-preserving manner. This entails the exchange of trustworthiness evidence (of a vehicle) in a manner that allows the characterisation of its trust state but without revealing any specific identity, configuration and/or implementation properties of the attested vehicle [16], so as to not increase an attacker's capabilities to track or exploit the operational profile of the vehicle.
Story-VIII: Trusting Verifiable Presentations. **Story-XIII:** Verify a trustworthiness claims VP provided in a CAM//CPM message.	The Verifiable Presentations are anonymised through the construction of *threshold signatures* (Section 5.6.1.2), and constitute the trust-related evidence collected in scope of a trust assessment process [11]. Once compiled by the TCH component, they are ready to be shared with external entities as evidence on the trust level of the entire vehicle, *hiding* any sensitive attestation attributes on the internal vehicle components, so as to avoid vehicle fingerprinting and other privacy breaches.

	These **anonymised trustworthiness claims**, upon reception, should be verified on both their authenticity and integrity ("*been constructed from vehicles that are equipped with a valid* CONNECT *TCB*") but also on the associated proofs of possession (Section 5.6.1.3); i.e., *transmitting vehicle should have access to the original credentials, as issued by the AIV, MD Service, and TAF, holding the non-disclosed attestation attributes for each verified vehicle component.*

Table 5.1: *CONNECT* Protocols & Crypto Primitives used to support the User Stories given in Chapter 4.

These enhancements can be divided into: i) those for **on-boarding the ECUs** (Sections 5.3.1 and 5.3.2) and ii) those for **attesting to the state of the ECUs during runtime**; thus, ensuring that these attestation results (Section 5.4) are correctly signed before they are sent outside of the vehicle (Sections 5.5). The on-boarding protocols are reasonably straightforward, while those for confirming and signing the attestation results are more innovative and therefore described in some detail. Protocols that are standardised, e.g. setting up a TLS connection, are omitted for simplicity of the protocol descriptions.

This set of protocols constitutes the first version of *CONNECT*'s TCB for enabling the **overarching federated trust assessment mechanism (as described in D3.2 [15]) to maintain a continuous up-to-date view of the trust state of each node across the CCAM continuum.** The provided trust extensions enact the functional specifications detailed in Chapter 4, in the form of user stories - focusing not only on node-centric trust but also data-centric trust capturing varying security and privacy requirements as posed by the target CCAM function. The detailed mapping can be seen in Table 5.1.

We have to highlight, that as mentioned in Chapter 2, these mechanisms currently focus on enabling the trust assessment process of a vehicle (node-centric trust) and its internal sensors and components monitoring kinematic data (data-centric trust). Essentially, they allow the secure and verifiable measuring and reporting of the correct configuration and behavioural state of all (in-vehicle) sensors and ECUs processing data as part of a function, thus, establishing a chain of trust for the entire vehicle anchored to the secure elements (TEEs) instantiated to those capable devices. This set of trust extensions will also provide the baseline for the trust assessment of the MEC infrastructure to be documented in D4.3 [18]: this does not only include infrastructural elements but also the CCAM services being instantiated on them including both the control- and network-plane of the service life-cycle management and orchestration layer. The internal crypto abstractions of all schemes are agnostic to the environment where they are instantiated. The core enrichment would revolve around the extensions of the TEE Device Interfaces to also monitor those MEC-specific characteristics based on which the overall Level of Assurance of the edge infrastructure can be quantified (based on the classification detailed in Section 2.1.)

5.1 Determinants Behind *CONNECT* Crypto Agility

Critical to the design of *CONNECT* trust assessment protocols and attestation mechanisms is the selection of the appropriate type of crypto primitives that satisfy the envisioned security and privacy requirements for the sharing of trustworthiness evidence, as "**trustworthiness claims**". Based on this, the trust state of an entity or data item will be calculated. As described in D3.1 [9]

(and further elaborated in D3.2 [15]), *CONNECT* supports three variants of the Trust Assessment Framework (TAF): (i) **Standalone TAF** for creating a local view on the trust level of either the host entity (vehicle or MEC) or a remote entity but solely relying on its own perception of its environment (build from locally extracted data and CAM/CPM messages received from neighbouring nodes); (ii) **Federated TAF** that enacts upon the joint calculation of a trust score of a (SW and/or HW) component between two or more TAFs. This trust quantification elevation relies on the secure collaboration of distinct TAFs (residing on the vehicle and MEC or the vehicle's *digital twin*) exchanging trust sources, as part of their linked trust models, towards the calculation and maintenance of trust reports for a wide set of nodes and data items; and (iii) **Digital Twin for Trust Assessment** which is an instantiation of the federated TAF mode of operation, enabling a vehicle to avail from the availability of the MEC in order to outsource its trust model and TAF state to its DT where reasoning on the trust sources can be executed on behalf of the original TAF agent residing in the vehicle.

In all cases, trust in data or entities is based on the sharing of trustworthiness evidence (characterised as trust sources) that capture the behavioural profile of the target node/component to be assessed. Such evidence might originate from any security control that has been deployed to the node - albeit, in *CONNECT* we rely on the **attestation attributes to be produced by the integrity verification capabilities**, offered by the underlying Trusted Computing Base (TCB), as well as the **detection of any misbehaviour** (encoded in **misbehaviour reports**) based on the plausability checks conducted by the employed Misbehaviour Detection (MD) framework [12]. Due to their nature, such trust sources contain sensitive information on the configuration and execution profile of the target node which renders existing verification and trust assessment schemes prone to **privacy breaches and implementation disclosure attacks** under "honest-but-curious" adversaries [20]. For instance, attestation mechanisms usually rely on a ***trusted verifier*** to maintain reference material, such as the target device's execution graph or configuration hash of its whitelist of installed binaries, and other acceptance criteria to decide on the legality of the attested node's properties, as recorded and reported by the proving node trust anchor. In *CONNECT* the role of the verifier is undertaken by the Attestation and Integrity Verification (AIV) component, as detailed in Section 2.1, which is responsible for governing the attestation process with the required in-vehicle components based on the Trust Assessment Request (TAR) initiated by the TAF. It monitors any changes in the trust level of a component (depicted through the continuously reported attestation attributes) and forwards the attestation report (with the associated evidence) to the TAF for the calculation of the Actual Trustworthiness Level (ATL). In the same context, the MD service (deployed as part of a vehicle's internal security policy management) has access to all extracted kinematic data and the originating in-vehicle sensors so as to reason on any signs of malicious or otherwise incorrect behaviour in order to inform the TAF.

One important guiding factor in the design of any trust assessment protocol (so as to also not hinder its adoption by the automotive community and standards) is that this **exchange of (sensitive) attributes related to a vehicle's trustworthiness should not create any privacy**[1] **(or other security-related) threats**: *An attacker should not gain any advantage in identifying a vehicle (or linking overheard T-CAM/T-CPM message transmissions to their source) nor extracting any aspects of the vehicle, that may compromise its privacy, including the brand of the vehicle and its internal architecture (vehicle fingerprinting)*. Especially the latter may lead to breaches in the unlinkability and untraceability of the vehicle and enable implementation disclosure attacks,

[1] There are no such privacy considerations for trust evidence related to the MEC virtualized infrastructure since sharing of such attestation attributes is envisioned in the Service Level Agreement (SLA) signed by all involved stakeholders; i.e., Mobile Network Operator (MNO), MEC tenant Application Provider, and MEC User.

since a malicious party may deduce information related to the internal architecture of the vehicle. For instance, consider the envisioned C-ACC use case [12] where data collected from cameras in a vehicle, or low-level sensors such as a LIDAR, needs to be sent to the Zonal Controllers of the vehicle, and afterwards to the C-ACC application on the in-Vehicle Computer. However, in addition to the information originating from the vehicle itself, the C-ACC application also requires information originating from the roadside infrastructure, information from the Global Navigation Satellite Service, as well as positioning and kinematic data from other vehicles. In the latter case, a vehicle may need to send information, such as steering data and location data, to other vehicles for the execution of a C-ACC service. When providing trustworthiness information, such messages could contain detailed information regarding the correct configuration of an ECU within a vehicle and this may lead to the inadvertent disclosure of data such as the vehicle's brand or internal architecture. This, in turn, can allow an adversary to identify and exploit possible vulnerabilities that may have been identified for systems featuring the specific characteristics (e.g., OS/FW version, kernel version, etc.) in order to penetrate the system. Therefore, we need to define **mechanisms that can allow the sharing of such type of (trust-related) messages but without requiring the disclosure of sensitive information.**

To remediate such privacy issues and simultaneously reduce the complexity of the verification process of the TAF, *CONNECT* employs **zero-knowledge proofs** (as part of its Enhanced Configuration Integrity Verification (CIV) scheme (Section 5.4)) for enabling the AIV to verify the trust state of a component without the need of getting access to the (raw) monitored traces capturing the target system's configuration and behavioural profile. This is achieved through the provision of **local attestation capabilities** (from the underlying *CONNECT* TCB and the Attestation Agent) offering three core innovations. First, *CONNECT* TCB provides the trusted computing abstraction, called **policy-restricted attestation key**, that dynamically restricts a node's attestation key (secured in each TEE) to policies enforced by authorized entities (i.e., OEMs through the trusted anchor of the Identity & Authentication Management (IAM) component as the secure component managing all identity attributes of a vehicle or MEC tenant application provider). By predicating its ability to use its attestation key for signing on its configuration correctness, we can verify a node's conformance using a simple challenge-response protocol that neither requires nor reveals any configuration information. The verification can be performed (by the AIV) through simply verifying the validity of the received signature. Secondly, *CONNECT* leverages Direct Anonymous Attestation (DAA) (Section 5.2.4) as a platform authentication mechanism capable of offering strong anonymity guarantees. More specifically, during the on-boarding of TEE-equipped ECUs in the vehicle (Sections 5.3.1 and 5.3.2), a DAA Key is created as a signing attestation key, bound to key restriction usage policies as instructed by the IAM. Such policies essentially represent the expected configuration state of the target device so as to be deemed trustworthy. This way, the device can only unseal its attestation key if its configuration has not changed. Thus, verifiers, who know a node's public key, can determine its integrity simply by requesting it to sign a challenge using its secret key. Furthermore, the fact that DAA is based on the use of Elliptic Curve Cryptography (ECC), built on pairing-friendly curves, allows for the signing of attestation data structures in a secure and privacy-preserving manner, thus, reducing the knowledge that external entities can learn about the system.

Third, to update a component's "*trusted configuration state*" during runtime (e.g., due to a SW upgrade for deploying new security patches), **IAM should be able to reactively authorise new policies binding/sealing the usage of the component's attestation key to a new configuration state.** However, this needs to be done in a way that can safeguard against possible manipulation from a compromised host: As part of the attestation process, the challenge that is sent from the AIV to all target devices will be forwarded to the Attestation Agent (running in the

"*trusted world*" of the device) through a process been instantiated in the "*untrusted*" (vehicle) host that may potentially try to manipulate the parameters given to the attached trusted execution environment. This essentially considers a Dolev-Yao adversarial model, which allows an adversary to monitor and modify all interactions between the host and the trusted component. **The critical operation to be verified is the management of the policies that are calculated outside the TEE and that these have been correctly calculated based on the latest SW version of the application binary to be attested.** While such a policy, as aforementioned, is instructed by the IAM, a compromised host could take a snapshot of a previously valid key restriction usage policy (e.g., prior to a SW upgrade) and, then, try to load it again to the trusted component. This will allow the adversary to bypass the attestation mechanism since they will be able to instruct the AA to create a signature (leveraging the DAA Key) based on a policy that matches a previous SW version that might be exposed to specific vulnerabilities. Hence, they will be able to rollback the device to a previous untrusted state without this being known to any verifier (AIV) that will assume that such attestation attributes/signatures have been created given that the correct key restriction usage policy has been enforced. To alleviate this challenge, *CONNECT* is the first of its kind to introduce the concept of Verifiable Policy Enforcement (VPE) (Section 5.4.2.2) for safeguarding the policy management with adequate proofs on the authenticity and integrity on any update operation that tries to alter the key restriction usage policy.

Another important factor, as outlined before, is that the data collected by the security controls (attestation enablers, misbehaviour detection), and used as trustworthiness evidence does not reveal any type of information that may have adverse effects on the privacy of the vehicle and/or user. One of the most predominant threats in this regard is vehicle fingerprinting, which refers to the identification of aspects of the vehicle that may compromise the underlying privacy requirements, and to enable a malicious party to perform implementation disclosure attacks. To compound this issue, it is important to **leverage appropriate signature schemes that can anonymise the trustworthiness claims** (constructed by the Trustworthiness Claims Handler (TCH) and encoded as Verifiable Credentials [11]) so as to protect against any **linking operations** that can be done by an external entity (when such claims are sent outside the vehicle - to either neighbouring vehicles or the MEC). This feature, in combination with the need to assume **near-zero trust** assumptions for the vehicle, requires additional validation of all those components that act as trust sources (including attestation enablers, misbehaviour detection and the TAF itself) and provides all necessary trust-related information based on which an up-to-date view of the trust state of all components across the CCAM continuum can be maintained.

The critical operations to verify are the *authenticity* and *integrity* of all computing elements that participate in the trust assessment process: from the AIV and MD that provide trust sources to the TAF, including also the in-vehicle sensors and components that are part of a function chain to be attested (as part of a trust model), to the TAF itself that needs to prove its integrity when providing a trust opinion on a trust property of interest for a node- or data-item. There is no inherent trust bootstrapped to any of these components, hence, it is also necessary to provide evidence about the correctness of the *CONNECT* security elements themselves so that an external verifier can have the necessary guarantees that the information reported is trustworthy. This comes as an additional layer of safeguarding the *CONNECT* trust assessment process by employing trust abstractions that can protect the construction of trustworthiness claims (through associated signatures) *if and only if* the majority of the participating components are deemed trustworthy. In other words, a valid signature can be associated to a TC *if and only if* at least k out of l entities correctly execute the overarching trust assessment protocol. All the above, led to the design of a new protocol towards the bootstrapping of vertical trust to all *CONNECT* security related components, leveraging strong crypto primitives including threshold signatures and DAA capabilities.

In what follows, we elaborate on all the crypto building blocks leveraged as part of *CONNECT*'s crypto agility layer for achieving the aforementioned security and privacy requirements through the entire life-cycle of a vehicle.

5.2 *CONNECT* Crypto Primitives & Building Blocks

5.2.1 Shamir Secret Sharing

Secret sharing refers to methods allowing the secure distribution of a secret among l entities, such that the secret can be retrieved only if a predefined threshold number of group members t coordinate and correctly execute the protocol. Shamir secret sharing realises that functionality by utilising degree t random polynomials and Lagrange coefficients to retrieve the constant factor of those polynomials.

More specifically, let s be the original secret to be shared among n entities, $\{E_i\}_{i\in[n]}$. Let D be the dealer which will perform the secret sharing operation, and distribute the share s_i to each E_i, for $i \in [n]$.

First, D will generate a random polynomial P of degree t (where t the threshold of the scheme), for which it holds that $P(0) = s$. They will then set $s_i = P(i)$, and share each s_i with the entity E_i. The original secret s can then be reconstructed from the shares of $m \geq t$ entities $\{s_i\}_{i\in[m]}$, by summing them up (times a coefficient), i.e., it holds that;

$$s = \sum_{i\in[m]} l_i * s_i$$

where l_i the Lagrange coefficient for the set $[m]$, calculated at 0, i.e., $l_i = \prod_{j\in[m],j\neq i} \frac{j}{(j-i)}$.

Shamir's secret sharing technique can be used to distribute a single signing key to multiple participants, such that, only a threshold number of participants can use it. In CONNECT, the DAA Issuer will sample a random secret key for calculating Schnorr signatures, and use Shamir secret sharing to divide it into shares for the vehicle's A-ECUs. Each A-ECU will use that share to participate in our enhanced DAA scheme, by executing a threshold signature protocol (see Section 5.2.2).

5.2.2 Threshold Signatures

In a k out of l threshold digital signature scheme, valid signatures can be generated only if at least k out of l entities (also called the Signers) participate and correctly execute the protocol. More precisely, in a threshold signature protocol considering a set of signers $\{S_i\}_{i\in[l]}$, a secret/public key pair (sk, PK) will be generated, either by a trusted third party authority, or in a distributed manner among the S_i. The PK will be made public, while the secret key will be "divided" into shares, and stored by each S_i (in the case where key generation is curried out by a trusted third party, this for example can be done with Shamir's secret sharing technique, as described in Section 5.2.1). During signing, each S_i will use their secret key share, to generate a signature share, which will later be included in the calculation of a signature, valid under the group public key PK. Security of such schemes dictates that it will not be possible for any subset of Signers

$\{S_j\}_{j\in J}$, where $J \subseteq [l]$, with $|J| < t$, to coordinate and calculate a valid signature under the group public key PK.

Threshold signatures are used in our enhanced DAA scheme, to showcase that at least a threshold number of protocol participants have correctly executed the protocol. In CONNECT, we will use this to sign trustworthiness claims, while at the same time showcasing that a number of A-ECUs greater than a threshold value (defined by the DAA Issuer) is in a correct state, which will allow them to participate in the threshold signature protocol.

5.2.3 Flexible Round-Optimised Schnorr Threshold Signatures (FROST)

In this Section we will give a quick overview of FROST [28] [8], a Flexible Round-Optimised Schnorr Threshold signature scheme. There are 3 types of entities participating in the execution of the FROST protocol, a set of $n \geq 1$ Signers $\{S_i\}_{i\in[n]}$, a Signature Aggregator SA and a Trusted Dealer TD. At the start of the protocol, each S_i will receive a secret key share sk_i from the TD, which will later use to contribute a signature share σ_i. The SA will then gather and combine each σ_i to the final Schnorr signature σ.

Frost comprises of two main stages; a key generation and distribution phase and the signature generation protocol. During key generation, the TD will compute a secret key sk and its corresponding public key $Y = g^{sk}$. Then, using Shamir's secret sharing technique, they will derive n shares sk_i of sk, with a threshold t (meaning that with any $m \geq t$ secret key shares sk_i, one could re-compute the original secret key sk). Note that the secret share sk_i are "additive". To see why, consider how TD uses Shamir secret sharing. First, the TD will generate a random polynomial P of degree t (where t the threshold of the scheme), for which it holds that $P(0) = sk$. They will then set $sk_i = P(i)$. The original secret sk can then be reconstructed from $m \geq t$ different $\{sk_i\}_{i\in[m]}$, by summing them up (times a coefficient), i.e., it holds that;

$$sk = \sum_{i\in[m]} l_i * sk_i$$

where l_i the Langrange coefficient for the set $[m]$, calculated at 0 (i.e., $l_i = \prod_{j\in[m],j\neq i} \frac{j}{(j-i)}$). This property will translate to the signatures shares σ_i, provided by each participant, meaning that generating the final signature σ, will only require adding the received signature shares together.

The signature generation protocol in FROST compromises by 2 single round phases; a pre-processing stage and a sign step. The SA will initiate the protocol by choosing a set $S = \{i_1, ..., i_m\}$ of Signers $\{S_i\}_{i\in S}$ to participate in the execution of the scheme.

Pre-processing Phase: During the prepossessing stage, each signer S_i, $i \in S$, will commit to 2 nonces; a blinding nonce d_i and a binding nonce e_i. The blinding nonce serves as the contribution of S_i to the Schnorr signature nonce, while the binding nonce is used to avoid attacks caused by combining protocol runs over different messages or set of Signers [23]. More specifically, the binding nonce d_i will be used to "bind" each Signer's responses, to the message to be signed and the set of Signer's participating in a specific run of the FROST protocol (see bellow). To complete the prepossessing step, each Signer S_i will return the commitments $(D_i = g^{d_i}, E_i = g^{e_i})$ to the SA.

Sign Phase: To start the sign step, the SA will gather all commitments (D_i, E_i) from each S_i, $i \in S$. They will then sent (m, B), for $B = \{(D_i, E_i)\}_{i\in S}$ to each signer S_i. After receiving (m, B), each Signer will verify m (i.e., that it is a message they are willing to sign), and also

check that $D_i, E_i \in G_1, \forall(D_i, E_i) \in B$. If successful, each S_i will then compute a binding value for each of the other Signers participating to the protocol; $p_i = H(i, m, B)$, $i \in S$, where H a hash function. Finally, each S_i in S will compute the group commitment $R = \prod_{i \in S} D_i E_i^{p_i}$, the challenge $c = H(R, Y, m)$, and their signature share $\sigma_i = d_i + (e_i * p_i) + l_i * s_i * c$. Note that the group commitment R and the challenge c will be components of the final Schnorr signature on m.

Each S_i will return their signature share σ_i to the SA. After receiving all signature shares, the SA will perform the following steps;

- For each S_i, $i \in S$, calculate its binding value $p_i = H(i, m, B)$, its contribution to the group commitment $R_i = D_i E_i^{p_i}$, as well as the group commitment it self $R = \prod_{i \in S} R_i$.
- The SA will then check if $g^{\sigma_i} = R_i Y_i^{c * l_i}$, $i \in S$ to validate the signature share of each Signer.
- If successful, the SA will aggregate the shares σ_i of each S_i to get $\sigma = \sum_{i \in S} \sigma_i$. The tuple $z = (R, \sigma)$ is the final signature output of the protocol. If executed correctly, z will be a valid Schnorr signature on m.

For completeness, we will outline here the proof of correctness of FROST. For simplicity, lets assume that every Signer executes the protocol correctly. If they are not, either they will be "caught" by the SA, in which case the protocol will be aborted, or the final signature z will be invalid. Note that if we set $k_i = d_i + e_i * p_i$, then $R = \prod_{i \in S} g^{k_i}$ and $\sigma_i = k_i + c * l_i * s_i$, where each k_i a random scalar. Setting then $k = \sum_{i \in S} k_i$, we have that $R = g^k$ and $\sigma = \sum_{i \in S} \sigma_i = k + c\,(\sum_{i \in S} l_i * sk_i)$. Using the fact that the secret key shares are additive, we get that $\sigma = k + c * sk$, meaning that $z = (R, \sigma)$ is a valid Schnorr signature on m, for the public key $Y = g^{sk}$.

FROST will become the basis on top of which we will build our enhanced DAA scheme. At a high level, our scheme is based on anonymising the public key Y which verifies the signature returned from FROST. Note that, our scheme is threshold Schnorr signature agnostic. That said, FROST provides desirable advantages in performance, robustness and maturity.

5.2.4 Direct Anonymous Attestation (DAA)

Direct Anonymous Attestation **(DAA)** is a platform authentication mechanism that enables the provision of privacy-preserving and accountable services. DAA is based on group signatures that allow remote attestation of a device associated to a Trusted Environment **(TE)**, while offering strong anonymity guarantees. In general Direct Anonymous Attestation **(DAA)** consists of a DAA Issuer, a set of Signers and a set of Verifiers. The DAA Issuer acts as a Trusted Third Party **(TTP)** and creates the DAA membership credential for each signer that wishes to participate in the DAA scheme. A DAA credential corresponds to a signature of the signer's identity produced by the DAA Issuer. A DAA Signer is usually consisted by a (HOST, Trusted Environment) pair, in CONNECT each DAA Signer will be consisted of the (HOST, Gramine) pair. Acquiring a DAA credential, indicates a membership credential to the DAA community and a trustworthy validation of the Signer. In other words, by providing a DAA signature, a Verifier can attest to the correct state of the signer by ratifying the validity of this signature construction. This allows for the provision of *local attestation capabilities* where attestation and verification mechanisms can be done in a zero-knowledge manner; without the need for the Prover to disclose any details on the configuration and implementation status of the device. Moreover, 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 DAA membership credential, without revealing any other information about the identity of the Signer. More concretely, the DAA provides the Signer with the ability to create signatures in an anonymous way, whilst still convincing the Verifier that it possesses valid DAA credential. Furthermore, the Direct Anonymous Attestation **(DAA)** is known to have the following properties:

- **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 (i.e., bsn). If a signer uses the same bsn in two signatures, they are linked; otherwise, they are not.
- **Unforgeability:** When the DAA Issuer and the HOST's Trusted Environment, (i.e., in *CONNECT* the Trusted Environment is Gramine), are honest no adversary can create a signature on a message μ with respect to the bsn when no Gramine enclave signed μ with respect to the bsn.
- **One-More-Unforgeability:** When the DAA Issuer is honest, an adversary can only sign in the name of compromised Gramine enclaves. More precisely, if n Gramine enclaves are compromised, the adversary can create at most n unlinkable signatures for the same base-name (i.e., bsn).
- **Anonymity:** An adversary that is given two signatures, with respect to two different base-names (i.e., bsn), cannot distinguish whether both signatures were created by one honest device, or whether two different honest devices created the signatures.
- **Non-frameability:** No adversary can create signatures on a message μ with respect to base-name (i.e., bsn) that links to a signature created by an honest device for the same bsn, when this honest device never signed μ with respect to bsn.

5.3 Preparing the Vehicle for the CCAM Continuum

The user stories described in Section 4.2 outline what needs to be done to prepare the vehicle for the execution of the *CCAM* functions and supply the data needed to support them. In a *CONNECT* vehicle, these functions are enhanced by the focus on trust: i) *Are the systems setup correctly*? ii) *Are the different software components behaving as they should*? iii) *What is the level of trust that should be placed on data stemming from another vehicle*? To provide an answer to these questions, along with many others, we must ensure that the vehicle is setup correctly; thus, we should verify that systems are operating as expected and ensure that communication between the different parts of the system is monitored and verified at all stages (this is the task of the *TEE-GSE*). Once this is completed, we leverage the *TAF* and *MBD* to provide extra assurances, combining the internal data it receives (i.e., measurements, configuration integrity verification and attestation results) together with the data from the CAM/CPM messages received from other vehicles in the vicinity and the DENM messages received from the MEC.

The *TEE-GSE*, *TAF* and *MBD* are all executed within containers deployed on the the vehicle computer, as described in D4.1 [10]. The setup and configuration of the aforementioned components within the containers is described in Section 3.2.3. Part of this setup is the provision of the necessary keys for ensuring the integrity and provenance of data in the system.

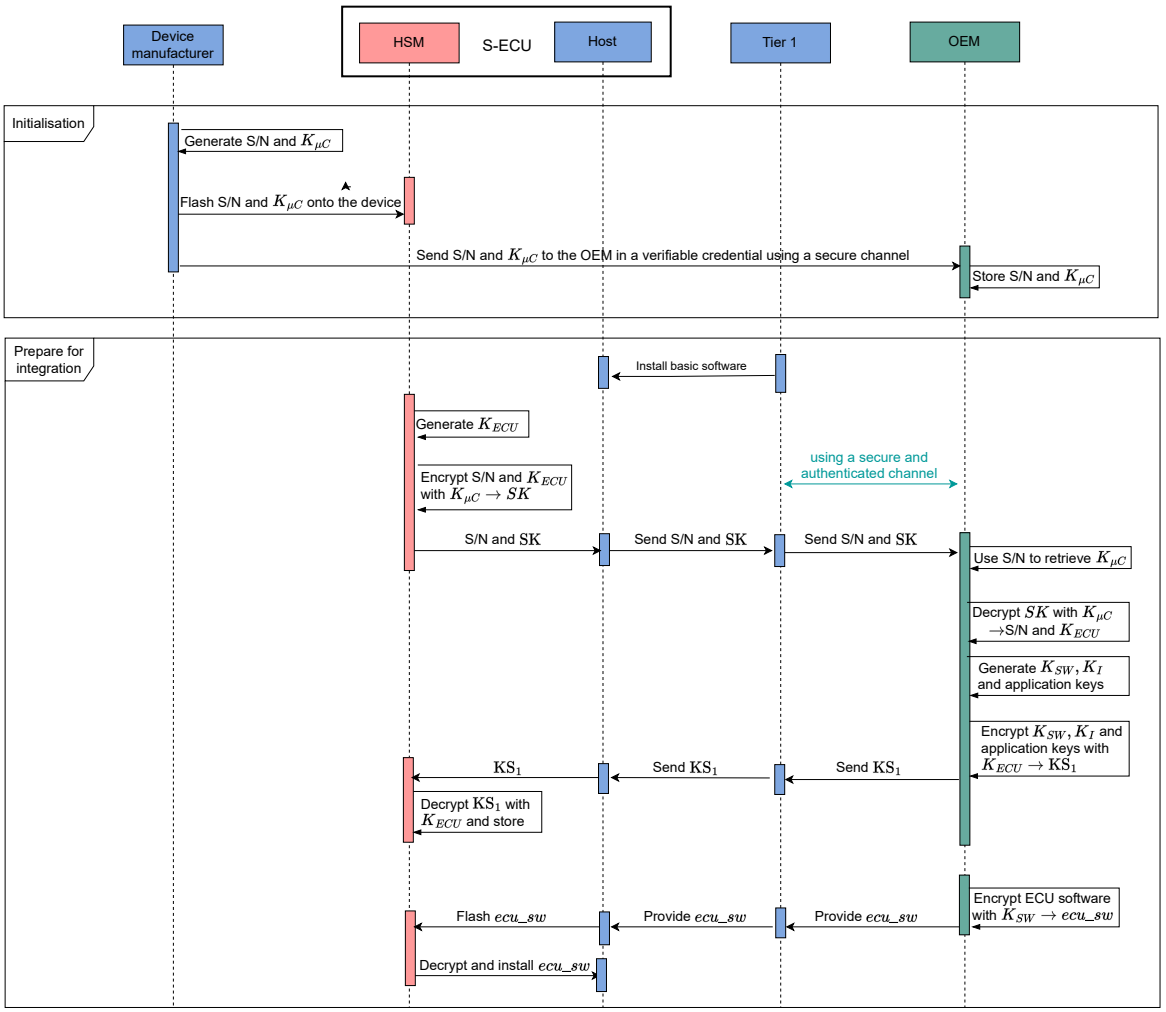

Figure 5.1: Initialisation and Preparation for Integration for an S-ECU.

The rest of the system components (i.e., ECUs, including *A-ECU* and *S-ECU*), also need to be on-boarded. By utilising the term on-boarding, we refer to the setup of the correct software and keys needed. The protocols supporting the secure on-boarding depend on the type of ECU; hence an *S-ECU*, may solely support symmetric cryptography, while an *A-ECU* can further support asymmetric cryptography. The *Zonal controller (ZC)* are *A-ECU* which in addition to other functions they provide, act as gateways between the different vehicle communication buses. It is these protocols that are illustrated by user stories: Story-I and Story-V.

The next two Sections describe the on-boarding processes, but before describing them we note that the industry identifies the following entities involved in production and integration of ECUs into a vehicle:

OEM: The vehicle manufacturer. When they receive the ECU they also receive the ECU's serial number and associated key data (the key for an S-ECU and the key and corresponding certificate for an *A-ECU*).

Tier 1: This is the entity that prepares the ECU for integration into the vehicle, installing basic software used to configure the ECU, setting up the keys and finally installing OEM specific software onto the ECU.

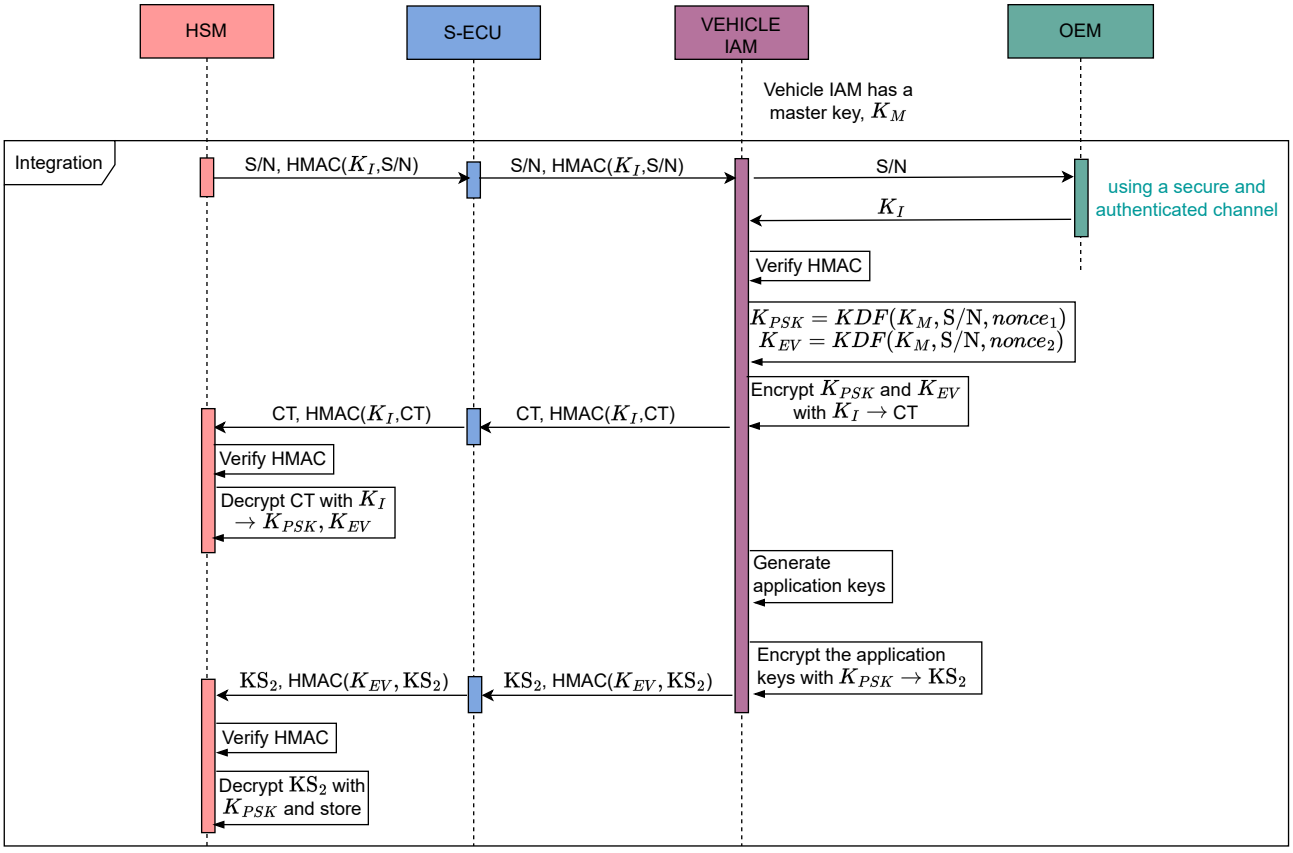

Figure 5.2: Integration into the vehicle for an S-ECU.

Tier 2: This entity is the ECU manufacturer. They initialise the ECU with a serial number and identity key. These are later provided to the OEM when they receive the ECU.

Notes:

1. Tier 1 and the OEM will often be the same entity.
2. The Tier 1 processes are carried out before the ECU is installed into the vehicle. Either when the vehicle is being manufactured, or when an ECU is being replaced at an OEM's service facility.
3. The *Zonal controller (ZC)* are integrated in the same way as the *A-ECU*.
4. Once the on-boarding protocols are executed the *CONNECT* attestation functions and key restriction policies that will be used are installed. This extra integration stage is described in Section 5.4.

All ECUs will use pre-shared keys to protect the integrity of the data communicated between them, the *Zonal controller (ZC)* and the different components in the vehicle computer.

Where data needs to be kept confidential, or access to the data needs to be controlled then it will be encrypted. In this case, the particular mechanisms used will depend on whether the ECU can do asymmetric cryptography, or not (*A-ECU* compared to *S-ECU*).

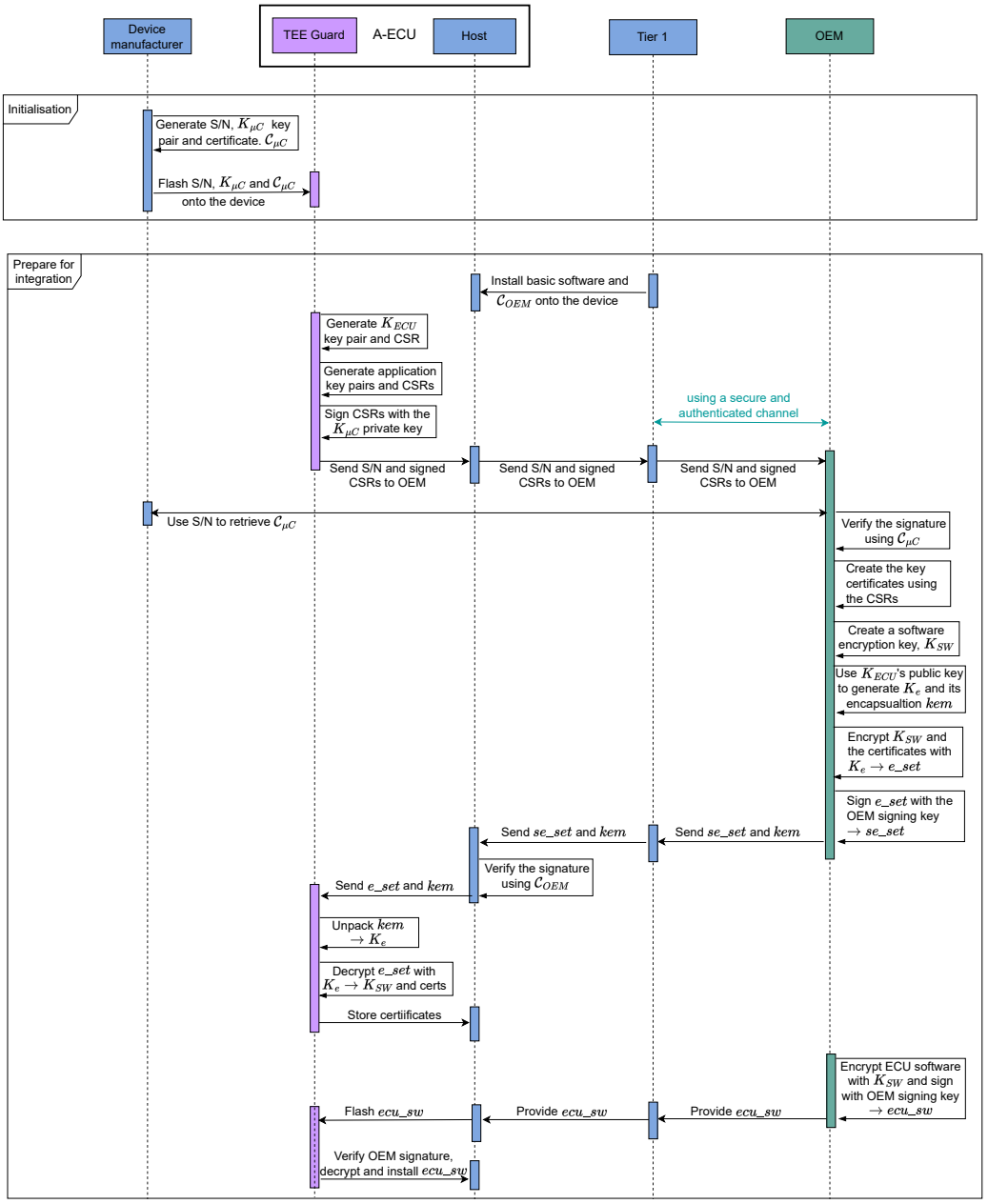

Figure 5.3: Initialisation and Preparation for Integration for an A-ECU.

5.3.1 Protocols for On-boarding the S-ECU

In Figure 5.1 the protocols used to initialise an *S-ECU* and prepare it for integration into the vehicle are illustrated. These stages are carried out by the device manufacturer and the Tier 1 entity before an *S-ECU* is installed into the vehicle. In the initialisation stage the device manufacturer flashes a serial number (S/N) and the device's identity key ($K_{\mu C}$) onto the device as it is manufactured. This information is retained and later provided to the OEM so that it can be used when preparing the device for integration into the vehicle.

The Tier 1 entity prepares the device for integration into the vehicle, this will be done with a direct connection to the ECU and so the signals are not protected as they would be if the device was communicating on a network. However, communication between the Tier 1 entity and the OEM will be protected for authentication and security (for clarity, this is not detailed on the diagram).

Table 5.2: S-ECU keys and their use.

$K_{\mu C}$	The ECU identification key, flashed onto the device as it is manufactured, used to enable the OEM to receive the ECU key.
K_{ECU}	The ECU key, used when preparing the device for integration into the vehicle.
K_{SW}	The OEM software key, used when installing software onto the device.
K_I	The key used when integrating the device into the vehicle.
K_{PSK}	A pre-shared encryption key used, when required, to protect sensitive data.
K_{EV}	The key used to protect the integrity of data passed between the ECU and the vehicle's main computer.

The first stage of the preparation is to install the basic software that will be used in this preparation process. Once this is done, the device's HSM generates the K_{ECU} key, encrypts this and the device's serial number using $K_{\mu C}$ and sends this together with the serial number in plain text to the OEM. The OEM uses the serial number to retrieve $K_{\mu C}$ which it uses to decrypt the message and obtain the serial number and K_{ECU}. Once the serial numbers are checked K_{ECU} can be used to encrypt and send data back to the ECU.

The OEM generates a key to be used when installing the software, K_{SW}, and the integration key, K_I. Depending on how the ECU is to be used it may also generate some application keys, even at this early stage. These keys are all encrypted with K_{ECU} and sent to the HSM which can decrypt and store the keys securely. The final stage of the preparation phase is to install the software on the ECU, to do this the OEM encypts the software using K_{SW} and sends it via Tier 1 to the ECU where it is decrypted and installed. Note that which application keys are necessary and when they are installed on the ECU will depend upon how the ECU is to be used in the vehicle.

Once the ECU is prepared for integration it can be installed in the vehicle and the protocols used to do this are given in Figure 5.2. The ECU integration is managed by the IAM, it starts with the ECU sending its serial number, protected with an HMAC using the integration key, K_I, to the IAM. The IAM uses the serial number to retrieve K_I (this may have been stored, or is re-generated as required). The OEM can the check the HMAC and, provided the check proceeds, can then go on to generate keys that it will use to protect communication with the ECU. These are K_{PSK} a pre-shared key that it will use for protecting data that it exchanges with the ECU and K_{EV} a key that is used to protect the integrity of communications between the ECU and the vehicle's main computer.

These keys are encrypted using K_I and sent to the ECU's HSM protected again using an K_I in an HMAC. The HSM verifies the HMAC and then decrypts and stores these keys. From now on K_{EV} will be used to protect communications with the IAM (and other components in the vehicle computer). The final integration stage is to generate the necessary application keys, encrypt them using K_{PSK} and send them to the ECU, where the HSM can decrypt and store them. Table 5.2 shows the keys that have been defined.

5.3.2 Protocols for On-boarding the A-ECUs

On-boarding the A-ECUs (which includes the Zonal Controllers) follows the same pattern with initialisation carried out by the device manufacturer, preparation for integration by the Tier 1 supplier and then integration into the vehicle by the OEM. The protocols are shown in Figures 5.3

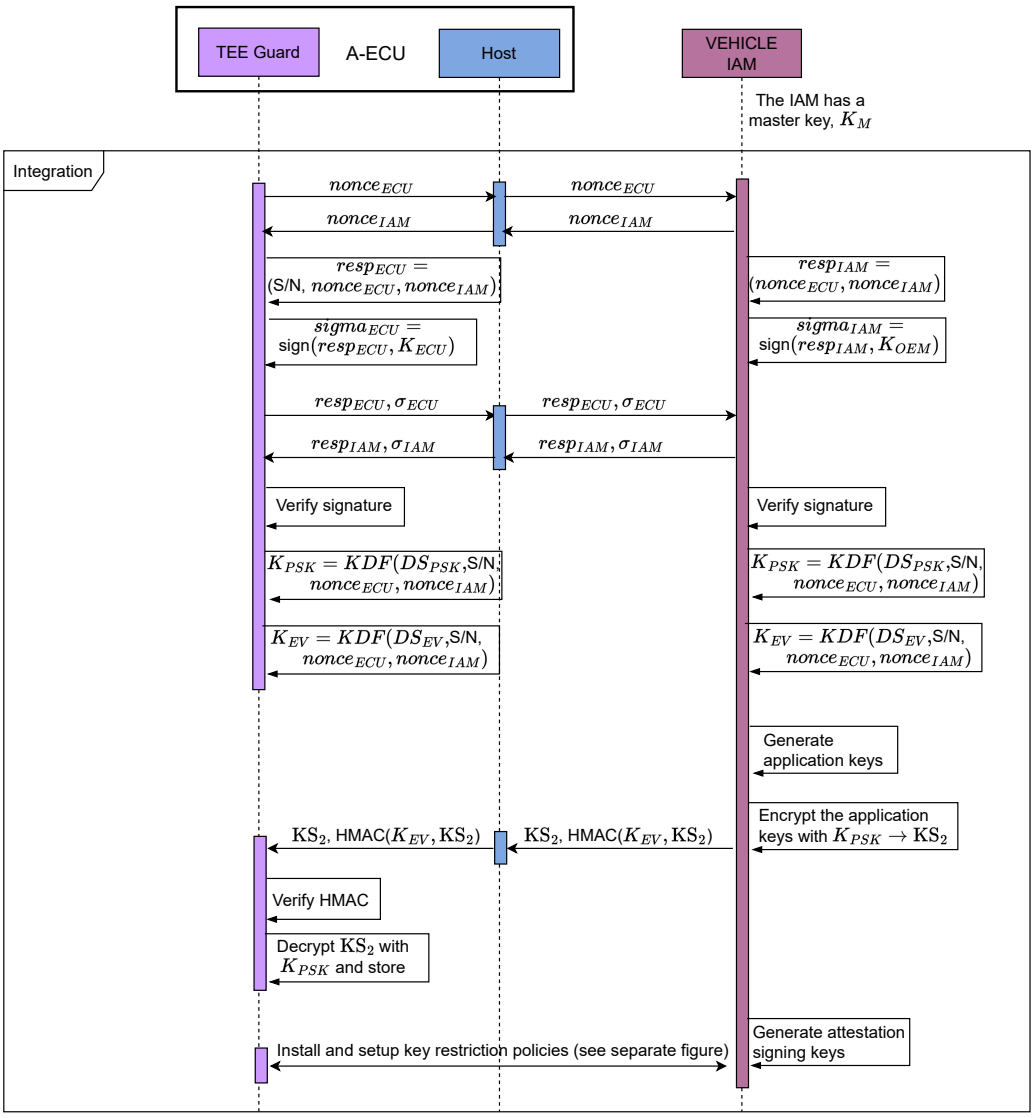

Figure 5.4: Integration into the vehicle for an A-ECU.

and 5.4. The significant difference is how the keys are generated and handled, the A-ECUs can use asymmetric cryptography, although pre-shared symmetric keys are also used for protecting the integrity of messages and for encryption. For the asymmetric keys installed on the ECU the OEM acts as the signing authority.

Referring to Figure 5.3, initialisation in this case consists of the provision of a serial number. S/N, an asymmetric key, $K_{\mu C}$, and its associated certificate, $\mathcal{C}_{\mu C}$. The OEM can later use the serial number to retrieve the certificate. In preparing for integration, the Tier 1 supplier starts by installing the basic software need to carry out this process. The next stage is to generate K_{ECU} and any application keys needed and their certificate signing requests (CSRs).

The CSRs are signed with $K_{\mu C}$ and sent together with the device's serial number to the OEM. The OEM retrieves $\mathcal{C}_{\mu C}$ from the device manufacturer and uses it to verify the signature on the CSRs. Once verified, it creates the key certificates, it also creates a software encryption key, K_{SW} that ii will share with the device. In order to send the key certificates and K_{SW} to the device it uses a key encapsulation method to generate an encryption key, K_e and its encapsulation, kem, from the public key for K_{ECU}. K_e is used to encrypt K_{SW} and the key certificates, this encrypted data is then signed and sent to the device. Provided the signature validates correctly the kem is

Table 5.3: A-ECU keys and their use.

$K_{\mu C}$	The ECU identification key, flashed onto the device (with its certificate) as it is manufactured, used to enable the OEM to receive the ECU key.
K_{ECU}	The ECU key, an asymmetric key used when preparing the device for integration into the vehicle.
K_{SW}	The OEM software encryption key, used when installing software onto the device.
K_{PSK}	A pre-shared encryption key used, when required, to protect sensitive data.
K_{EV}	The symmetric key used to protect the integrity of data passed between the ECU and the vehicle's main computer.

unpacked, the K_e retrieved and the data decrypted. The key, K_{SW} is stored in the TEE Guard and the certificates provided to the host. As for the S-ECU, the last part of this preparation phase is the provision of the ECU software.

Integrating the A-ECU into the vehicle proceeds without the involvement of the OEM, it starts with an exchange of nonces between the TEE Guard and the vehicle's IAM. Each side then signs the nonces and sends then back to the other party (the ECU also identifies itself, by including its serial number in its message). Once the signatures are verified the nonces are used to generate the pre-shared encryption key, K_{PSK} and the key for protecting the integrity of communications between the ECU and the vehicle, K_{EV}. Given the nonces, each party can generate the keys themselves using a KDF, the nonces and any other pre-agreed data. Keys for different purposes use different domain separators (DS) in the KDF. To complete the process the application keys are generated by the IAM, encrypted using K_{PSK} and sent to the ECU, message integrity is now protected using an HMAC with K_{EV}. On receipt the TEE Guard verifies the HMAC and decrypts and stores the keys. Table 5.3 shows the keys that have been defined.

5.4 *CONNECT* Configuration Integrity Verification as a Trust Assessment Source

CONNECT plans on employing several attestation mechanisms so as to provide Trust Sources (TS) to the *TAF*, enabling the assessment of the integrity trust property (i.e., details on the trust properties of *CONNECT* are available in D3.1 [9]). Such an attestation mechanism is the **Configuration Integrity Verification (CIV)**, used to ensure the correctness of the configuration of any device that is participating in the service graph chain. The algorithm that is executed for the CIV varies, depending on the capabilities of each device and the cryptographic primitives that it can support. The *S-ECU* for example, cannot provide verifiable evidence, contrary to the TEE capable devices that can support asymmetric cryptography and are capable as such, of providing verifiable evidence.

To this end, we are presenting a novel CIV attestation scheme, for all the TEE capable devices, based on the notion of **Attestation by Proof**. More specifically, we are introducing a challenge-based protocol, where a Verifier challenges the Prover with a fresh nonce; if the Prover is able to handle the Verifier's challenge (i.e., which means that it is capable of producing a valid signature with its Attestation Key), then the Verifier knows that the Prover is in a correct configuration state.

By creating a valid signature, the Prover provides Trustworthiness Evidence in a **Zero Knowledge** manner, as he is not disclosing the actual trace. This is achieved by integrating the creation of Attestation Keys as part of the Secure On-Boarding, where the *IAM* sets the Key Restriction Usage Policies to be enforced by the Local Attestation Mechanisms, for each newly installed ECU.

Therefore, in the remainder of this section, we present the finalised design of *CONNECT* enhanced CIV attestation scheme, which is capable of constructing the appropriate **Key Restriction Usage Policies** for separating the local and the global Attestation Keys.

In this context, the *CONNECT* CIV scheme aims to alleviate the Trustworthiness requirements of the *TAF*, *AIV*, *TCH* that act as Verifiers as we consider that the *CONNECT* operates in a Zero Trust manner. To this end, any Verifier is capable to attest the desired device/component, while simultaneously we consider that it should be difficult for the Verifier to infer any meaningful information regarding the Configuration/State of the attested device/component. It has to be noted here that *CONNECT* CIV scheme provides the Trustworthiness Evidence for both *CONNECT* architectures In-Vehicle and MEC, where for each architecture they are provided different types of Trustworthiness Evidence, as described in Chapter 2.

As aforementioned, the *CONNECT* enhanced CIV scheme which will be thoroughly described in the following Section concerns the **zero-knowledge** variant of *CONNECT* attestation capabilities and is based on the definition of a **Policy-Restricted Attestation Key** that is considered as the trust anchor of the attestation process. This enables enhanced authorisation mechanisms to the Prover's Attestation Key *if and only if* the respective protection policies, that are deployed as part of the underlying Root of Trust (i.e., in this case Gramine and Intel SGX), are satisfied. In *CONNECT*, the key restriction usage policies are dictated by the OEM and are enforced by the *IAM* component, acting as a Trusted entity that governs the overarching process of the proposed Vehicle and MEC architectures, thus ensuring that the Prover can only use the key to sign challenges if its configuration satisfies the predefined policy. By predicating its ability to sign "*Requests for Evidence*" (triggered by the *TAF* upon request by the *CCAM* service of interest) based on its configuration correctness (i.e., regarding both the configuration of the hosted application and the configuration of the enclave for handling the Attestation mechanisms), the conformance of the Prover can be verified using a simple challenge-response protocol that neither requires, nor reveals any configuration information to the Verifier.

Moreover, as we described in Section 4.7 (and more specifically Story-XXII and Story-XXIII), we need to support not only Software Updates, to resolve bug issues or to introduce new features, for the applications running within the TEE, but also to off-load security critical applications that are protected by a TEE. More specifically, such an action leads to a change of the configuration of the TEE. As a result, the previously issued Key Restriction Usage Policy is no longer valid even if no errors occurred during the software upgrade/migration. Such actions opens a wide attacks surface, as a compromised HOST can have two different valid policies, enabling the the creation of valid signatures even though the device now is acting maliciously. In this regard, the ***CONNECT* CIV** is capable of creating **time-limited policies**, enabling the ***IAM*** to control which policy can be satisfied at a given time.

As it was previously mentioned, *CONNECT* CIV scheme should be able to dynamically verify/identify that it is indeed the correct software version that is running inside a TEE enclave and not a deprecated one that has identified vulnerabilities. In other words, *CONNECT* enhanced CIV scheme scopes to eliminate any rollback or denial of update possibility that should not be allowed, unless specified otherwise in a security policy. To the best of our knowledge, this is a known challenge to the community which the *CONNECT*'s enhanced CIV scheme comes to

solve. The innovation of our new design is that it enables the Prover to provide Attestation Evidence regarding not only the configuration correctness but also the version of the software that was supposed to be running, in a **ZERO KNOWLEDGE** manner. In order to enable a TEE capable device to meet such high security requirements, we are introducing a new component to be part of our *TCB* called **Verifiable Policy Enforcer (VPE)** and is responsible for not only attesting the version of the running TEE application, but in general to not allow the enforcement of Key Restriction Usage Policies that are deemed obsolete.

Table 5.4: Notations used in the description of the CONNECT CIV scheme.

Notation	Description
$RootID_{priv}$	The private part of the Device identity key.
$RootID_{pub}$	The public part of the device identity key.
HW_secret	The MRENCLAVE KEY, the hash digest of the CPU's secret and the MRENCLAVE measurement.
IAM_{pub}	The public key of the IAM.
IAM_{priv}	The private key of the IAM.
$Tracer_{pub}$	The public key of the Tracer.
$Tracer_{priv}$	The private key of the Tracer.
$rand$	Random 32 bytes number.
P	The policy that the Attestation Key is to be bound with.
$seed$	The random seed under which the Attestation Key is going to be created.
AK_{priv}	The private part of the Attestation Key.
AK_{HASH}	The hash digest of the private part of the Attestation Key.
AK_{pub}	The public part of the Attestation Key.
AK_{name}	The name of the Attestation Key, aka the hash digest of the public part of the Attestation Key.
VPE_{priv}	The private part of the VPE Key.
VPE_{pub}	The public part of the VPE Key.
VPE_{ENC}	The Encrypted private part of the VPE KEY.
VPE_{HASH}	The hash digest of the private part of the VPE Key.
$P_{authorized}$	The approved runtime configuration hash digest of the Attestation Agent.
$Ticket$	A signature over the $P_{authorized}$ provided by the IAM.
$MRAA$	The enclave measurement of the Attestation Agent.
$MRTracer$	The enclave measurement of the Tracer.
$Secret1$	The secret that is targeted to be sent to the VPE.
$Secret1_{ENC}$	The encrypted $Secret1$.
$HMAC_{key}$	A random HMAC Key.
$Secret1_{MAC}$	The Authentication digest of $rand$ over the $HMAC_{key}$.
$rand1_{ENC}$	The Encrypted random $rand$ under the $RootID_{pub}$.
$Secret2$	The secret that is targeted to be sent to the Attestation Agent.
$Secret2_{ENC}$	The encrypted $Secret2$ under the $rand$.
$Secret2_{MAC}$	The authentication digest of $rand$ under the $HMAC_{KEY}$.
$rand2_{ENC}$	The Encrypted random $rand$ under the $RootID_{pub}$.
$Secret1'_{MAC}$	The Authentication digest of $rand$ over the $HMAC_{key}$.
$Secret2'_{MAC}$	The Authentication digest of $rand$ over the $HMAC_{key}$.
$nonce1$	A random 32 bytes number.
$nonce2$	A random 32 bytes number.
$TraceHash$	The hash digest of the runtime configuration of attested applications.
$\sigma 1$	The signature of a hash digest containing the $nonce1$ and the $TraceHash$ over the private key of the Tracer.
$\sigma 2$	The signature of a hash digest containing the $nonce2$ and the $MRTracer$ over the private key of the Tracer.
CC	The command code of specific restriction policy command.

$\sigma 3$	The signature of a hash digest containing the $nonce1$ and the $MRAA$ over the private key of the VPE.
σ	The signature of a hash digest over the private part of the Attestation key.

Throughout the remained of this Section, we will elaborate on the *CONNECT* enhanced CIV scheme, and we outline how the aforementioned properties are achieved.

5.4.1 System Setup & Notation

In *CONNECT* as mentioned we are planning to adopt a CIV scheme for all capable ECUs (both *A-ECU* and *S-ECU*). As the *A-ECU* lacks the cryptographic capabilities to provide verifiable Trustworthiness evidence we are not going to go into further details. Contrary to the *S-ECU* that provide unverifiable evidence, all the TEE capable device belonging to the in-Vehicle or the MEC architecture needs to provide verifiable attestation/Trustworthiness/evidence proving their configuration correctness, while fulfilling the security requirements set by the OEM. Specifically, in order to describe the *CONNECT* enhanced CIV scheme, we consider the following roles that will be taken by the target *CCAM* ecosystem.

- **Prover:** A TEE capable device with a certified identity key by the manufacturer. In CONNECT we focus on **Intel SGX** and of course **Gramine** enabled devices, which is arguably one the most mature and efficient open-source solution for developing trusted applications in the x86, x86-64 architectures. Moreover, each Prover is equipped with three (internal) functional components as part of their *TCB*.

  1. The **Attestation Agent:** which is managing the creation and usage of the Attestation Key of the respective device and is assumed to be running as part of the trusted world (protected/isolated by the host Gramine TEE).
  2. A **Runtime (Attestation) Tracer** which is responsible for monitoring and reporting runtime measurements of the applications/functions of which the level of trust need to be assessed. An example of such an application could be an ECU function that is processing the received kinematic data; thus, forwards these data to the *CCAM* service, which is instantiated in the in-vehicle computer. This *CCAM* service could responsible for making a decision regarding whether it is safe to change lane or not (i.e., *CONNECT* C-ACC Use Case). In this case, the function that is producing or processing the data, needs to be assessed as part of the process for creating a trust opinion on the data item itself.

     We consider that the tracer is operating both in the trusted and the untrusted world (i.e., as defined in Section 2.1). However, it has to be noted that all the security critical functionalities are hosted within the trusted world. More specifically, the untrusted part of the Tracer is responsible for fetching the the raw traces of the attested application, whereas the Trusted part of the Tracer is responsible for decoding the raw traces, in order to calculate the runtime configuration hash, and for executing all the cryptographic operations that takes place within the tracer.
  3. The **Verifiable Policy Enforcer (VPE)**. We consider that the VPE is part of our *TCB*, as defined in Section 2.1, which means that once the secure On-Boarding is completed successfully, we assume that it is trusted; hence it is impossible to get compromised during runtime. As aforementioned, in *CONNECT* we need to employ security

mechanisms in order both to prevent software version rollbacks and denial of software update attacks. To this end, we consider that the VPE is authorised to check during runtime whether or not the version of the Attestation Agent, the Tracer and any other security critical application that is running within an enclave is the expected one, as a defence against denial of software updates. Regarding defence against software rollbacks, *CONNECT* introduces a novel security mechanism, the Monotonic Counter, that will be thoroughly described in Section 6.3.

- **Verifier:** As we are moving towards a Zero Trust architecture, we consider that the Verifier is an untrusted device/component which wants to remotely check the correctness of the Prover's configuration. The Verifier can either be the *AIV*, the *TAF*, or even another vehicle, hence we cannot make any Trust assumptions. In this regard, it has to be highlighted here ,that we need to employ strong cryptographic mechanisms enabling the verification of any step of the attestation chain. We consider that the Verifier is also equipped with a Intel SGX and Gramine in order to host trusted applications. We identify that *CONNECT* approach for verifying the integrity of an ECU is quite heavy compared to the current standards, where it is used an 8 bytes key (for the calculation of digital signatures) and each communication channel is instantiated with CAN-buses that have very limited capabilities. Therefore, CONNECT is envisioned to be adopted by vehicles that are equipped with next generation CAN buses that are capable of communicating through Ethernet, so as to not have any restriction regarding the size of the Attestation/Trustworthiness evidence.

- ***IAM***: In the context of CIV, the *IAM* is considered to be a trusted entity that:

  1. Is responsible for managing the Secure On-Boarding each device to the in-Vehicle architecture by verifying their unique certified identity key.
  2. Performs the Secure On-Boarding of each device that is part of the In-Vehicle architecture as described in Section 4.3.
  3. Injects a secret Key to the VPE, which is blinded with the correct configuration of the rest of the TCB components.
  4. Maintains and authorises each device's approved configuration, as this will lead to the definition of new key restriction usage policies.

5.4.2 Enhanced CIV - *High Level Overview*

Having described all the entities and components that participate in the *CONNECT* CIV scheme, we will now provide a high level conceptual overview of the scheme. More specifically, the scheme can be broken down into two distinct main phases, namely i) Join and ii) runtime Attestation. During the Join phase, the *IAM* basically sets up the Key Restriction Usage Policies and creates the VPE key, whereas in the runtime Attestation phase the attested device gets challenged by the *AIV* to provide Trustworthiness/Attestation evidence.

5.4.2.1 JOIN Phase

1. Consider a newly TEE capable *A-ECU* that is installed on a Vehicle. First the *A-ECU* has to complete successfully the secure On-Boarding protocol so as to create unique keys per application and therefore establish a Trusted and Authenticated channel with the *IAM*.

2. Afterwards the *IAM* makes the necessary actions for the creation and authorisation of the Attestation Key. The *IAM* fetches the reference values of the CCAM applications, in order to compute the accepted configuration of the Attestation Agent. Apart from the reference values of the CCAM application, for the computation of the accepted configuration are appended the reference values of the Attestation Agent Enclave (MRSigner and MREnclave and of course a validation from the Tracer and the VPE. The expected configuration is signed and sent back to the ECU along with the key restriction usage policy that the Attestation Key needs to be bound.

3. The ECU receives the signature computed by the IAM, called **Authorisation Ticket**, and creates its Attestation Key bound with the respective (issued) key restriction usage policy. It has to be noted that this operation happens in the Attestation Agent of the ECU. Upon creation of the Attestation Key, the Attestation Agent sends to the VPE, the Attestation Key Name.

4. The VPE then requests from the *IAM* to create the VPE Key pair which is bound with a Key Restriction Usage Policy, representing the correct configuration and the versioning of the Tracer and the Attestation Agent. In this case, we bind the key with the Key Restriction Usage Policy by encrypting its private part with the respective policy. The VPE key then is injected to the challenging ECU's VPE.

5.4.2.2 Amending with Verifiable Key Restriction Usage Policies

1. Considering that a Verifier wishes to attest (during runtime) an ECU that is part of the in-Vehicle architecture, it will initiate an attestation challenge response protocol.

2. Upon reception of the challenges, the Attestation Agent of the Prover creates a fresh nonce and forwards it to the VPE.

3. The VPE creates another fresh nonce. Both nonces are forwarded to the Tracer, in order for him sign them preventing replay attacks during the attestation process.

4. The Tracer executes its Tracing algorithms in order to extract the runtime security measurement of the attested application's configuration. Upon extraction of the appropriate measurements, the Tracer signs the extracted value bound with the nonce circulated by Attestation Agent and signs the measurement, of the enclave environment where it has been instantiated, bound to the nonce initially created by the VPE. Both signatures are sent back to the VPE. By binding the two signatures the two distinct nonces by verifying the two signatures both the VPE and Attestation Agent acquire a proof of liveliness off the Tracer.

5. The VPE then will try to gain access to his injected key by satisfying the Key Restriction Usage Policy that was determined by the IAM. If the Key Restriction Usage Policy is satisfied the VPE signs the accepted software version of the Attestation Agent bound with the nonce issued from the Attestation Agent.

6. The Attestation Agent receives the two signatures, the signed value of the runtime untrusted application and signed accepted software version, and goes off to satisfy his own key restriction usage policy. To satisfy the Key Restriction Usage Policy, the Attestation Agent computes a runtime configuration from the enclave measurements and by verifying the two signatures. Upon successful completion of the policy enforcement algorithm the Attestation Agent signs the challenge he initially received from the Verifier.

5.4.3 Enhanced CIV - *Architectural Details & Mode of Operation*

In this Section, we provide a detailed overview containing all the details of the CONNECT enhanced CIV scheme, along with all the interactions between all involved components.Throughout this Section, we elaborate on each of the phases of the designed protocol.

Next, we describe the **Join** interface initiated by the **IAM** , performed when an A-ECU or a CONNECT component doesn't have a registered Attestation Key. The IAM acting as a Trusted Authority regarding the In-Vehicle/MEC architecture, is responsible for issuing the Key Restriction Usage Policies for each Attestation Key and creating unique key pairs for every VPE. It has to be highlighted here that the Verifiable Policy Enforcer is instantiated in a different enclave as it is part of CONNECT's Trusted Computing Base either this is an ECU along side with the Attestation Agent or it is the MEC along side the AIV. Given all the above, the actions followed in the context of the Join phase are shown in Figure 5.5, and are described as follows:

1. The IAM sends to the corresponding Attestation Agent the Policy that its Attestation Key has to be bound with. Such a policy represents that a specific policy restriction command was executed (prior to the Attestation Agent been able to use the AK construction) and that the IAM approves the runtime configuration of the Attestation Agent itself. By binding the Attestation Key with such a Key Restriction Usage Policy, it is ensured that an old set of Key Restriction Usage Policy assertions can be valid at the same time since the SW version will be invalid.To this end, the Attestation Agent creates a random number using a secure RNG and fetches the unique secret key that is a fingerprint of both the CPU and the Enclave application (the Gramine-specific **MREnclave Key**.

2. The Attestation Agent computes the Hash Digest of the concatenation of the aforementioned values, $seed = H(HW_Secret||rand||P)$, in order to compute the key seed that is going to be fed into a secure Key Derivation Function(KDF) and compute the private Attestation Key $AK_{priv} = KDF(seed)$. The public part of the Attestation Key is $AK_{pub} = AK_{priv} * G$, where G is the Group generator. Upon the successful creation of the Attestation Key pair, the AA computes the Hash Digest of the Attestation Private Key, $AK_{HASH} = H(AK_{priv}$, and computes the name of the Attestation Key, $AK_{Name} = H(AK_{pub}$. This action completes the creation of the Attestation Key structure and the Attestation Private Key is discarded. By disgarding the AK_{priv} we make sure that the Attestation Agent will have to recreate its key *if and only if* he is in correct configuration state, aka the correct policy is enforced. The Attestation Key is then forwarded from the Attestation Agent to VPE and finally to the *IAM*.

3. The *IAM* then calculates the Key pair of the VPE, $VPE_{priv} = KDF(random), VPE_{pub} = VPE_{priv} * G$. Afterwards *IAM* calculates the Key Restriction Usage Policy that the VPE key should be bound to. With that Policy Digest, *IAM* Encrypts the VPE_{priv} and computes its Hash Digest, $VPE_{HASH} = H(VPE_{priv})$, in order to finalise the creation of the VPE Key structure. The VPE Key structure is going to be wrapped properly and sent to the VPE enclave.

    - The **VPE Policy** represents that the VPE is indeed running on the approved version, that the application is actually signed by the *IAM* and finally that the version of the Tracer is the expected one.

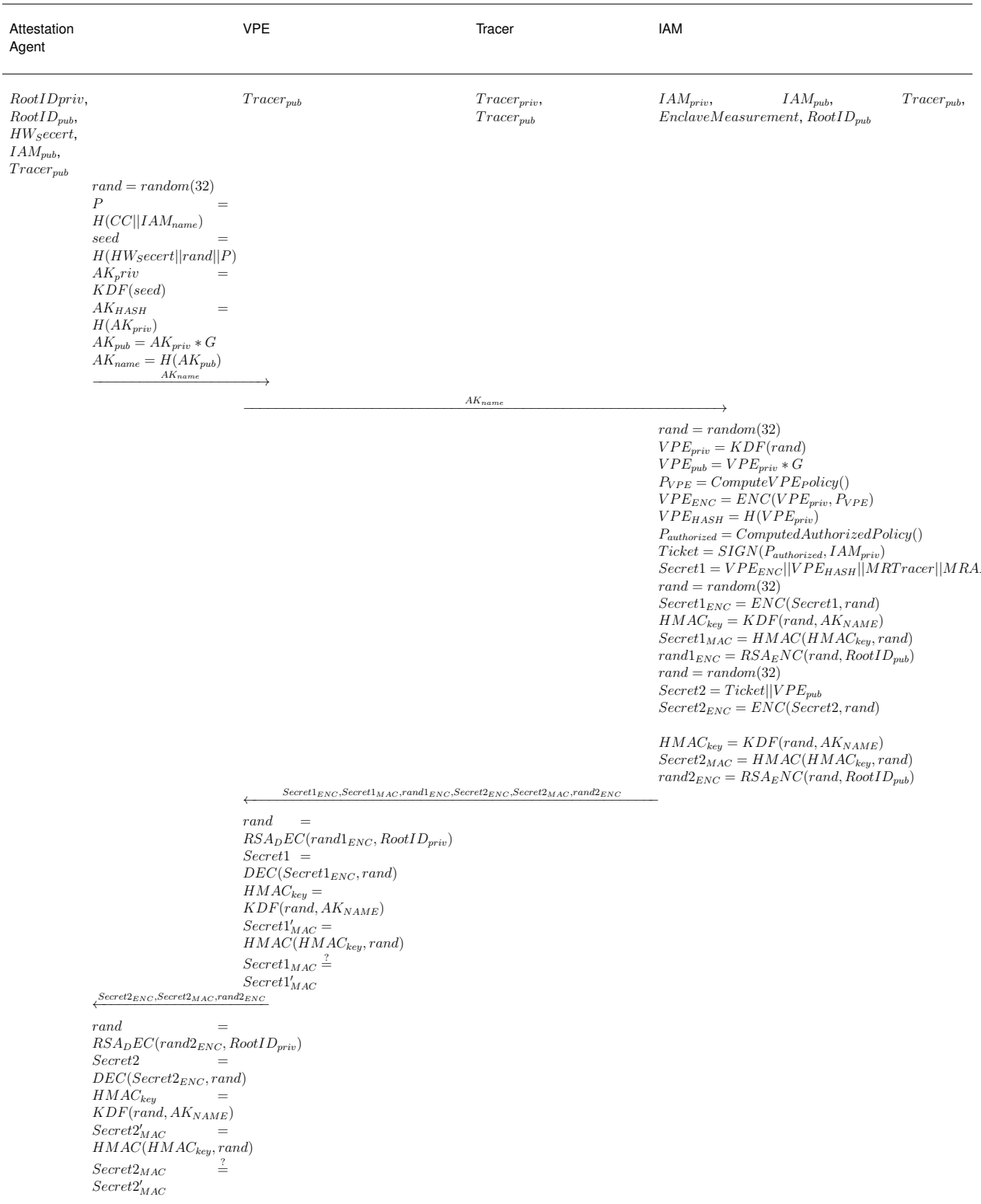

Figure 5.5: CONNECT Verifiable Key Restriction Policy Update: JOIN

4. With the VPE key creation now finalised, the *IAM* computes the accepted configuration of the A-ECU/component and signs it with his own private key. Thus creating an Authorisation Ticket.
    - The A-ECU/component accepted configuration is dependent on the enclave measurement of the Attestation Agent, on whether or not the software of the Attestation Agent was signed by the *IAM*, on the runtime value acquired by the Tracer and finally on the response of both the Tracer and the VPE.
5. The *IAM* creates two encrypted and authenticated secrets, the first secret is determined for the VPE and contains the Encrypted VPE_{priv}, the VPE_{HASH},the approved enclave measurement of the Attestation Agent and the approved enclave measurement of the Tracer. The other secret is destined for the Attestation Agent and contains the Authorisation Ticket and the VPE_{pub}. To create these encrypted and authenticated secrets we execute the following algorithm:
    (a) Create a Random number with a secure RNG, $rand = random(32)$.
    (b) Symmetrically encrypt the desired secret with the Hash of that freshly random generated number and the reference value of the enclave that is supposed to read it .
    (c) Create an a MAC key derived by the freshly created random number and the Attestation Key name ($MAC_{Key} = KDF(rand, AK_{Name})$).
    (d) Compute an authentication digest over the secret we wish to encrypt, $Secret_{MAC} = HMAC(Secret, MAC_{KEY})$.
    (e) Encrypt the random number generated in the first step with the unique identity key of the A-ECU. This unique identity key is considered to be a public RSA Key.
6. Both the VPE and the Attestation Agent receive their Encrypted and authenticate secrets respectively. The two components execute the following algorithm to extract their secrets:
    (a) Decrypt using the unique identity key of the A-ECU to extract the random number generated by the *IAM*.
    (b) With the extracted random number each component computes the symmetric encryption secret key $ENC_{Key} = H(rand, MR)$.
    (c) Decrypt with the ENC_{Key} the encrypted secret.
    (d) Create an a MAC key derived by the freshly created random number and the Attestation Key name.$MAC_{Key} = KDF(rand, AK_{Name})$.
    (e) Compute an authentication digest over the secret we wish to encrypt, $Secret_{MAC} = HMAC(Secret, MAC_{KEY})$.
    (f) If the acquired and the computed authentication digest match then the component can accept the secret and store for future use.

5.4.3.1 Runtime Attestation Assertions

Next, we describe the **runtime attestation** interface initiated by a remote Verifier, performed each time we need to collect evidences from an A-ECU or a CONNECT component, igniting basically a challenge response mechanism. Through this protocol the Prover creates evidence

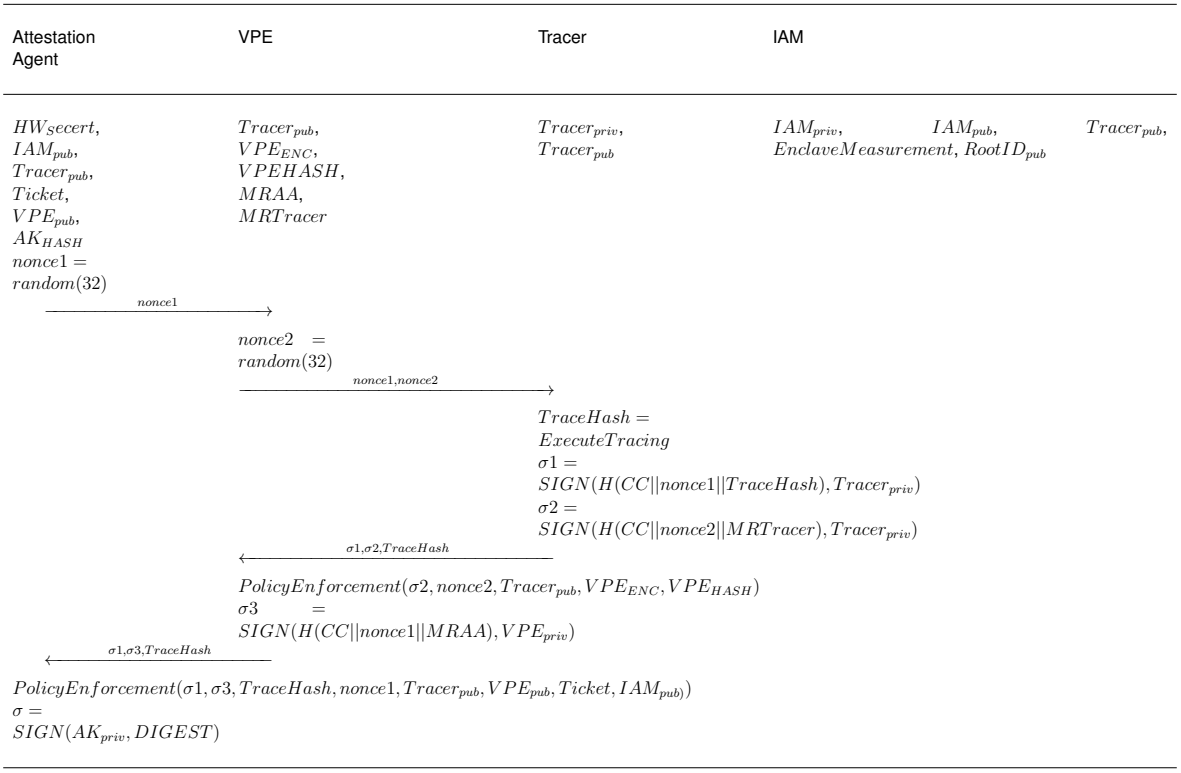

Figure 5.6: CONNECT Verifiable Key Restriction Policy Update: Run Time Attestation

for the Verifier that he is indeed in a correct configuration but without disclosing any information regarding its state. Given all the above, the actions followed in the context of the Join phase are shown in Figure 5.6, and are described as follows:

1. A Verifier, which could either be the AIV, the TAF or the TCH, issues a challenge that is sent to the Prover's Attestation Agent.
2. The Attestation Agent creates a fresh nonce, $nonce_{AA}$, that is sent to the VPE.
3. The VPE, creates on his end another fresh nonce , $nonce_{VPE}$, and forwards them both to the Tracer.
4. The Tracer takes security measurements for each requested untrusted application and creates two signatures one for the Attestation Agent $\sigma 1$ and one for the VPE $\sigma 2$.
   - The first signature is to be sent to the Attestation Agent, signing a Command Code (CC) that represents the execution of a specific command policy, the nonce that was issued by the Attestation Agent and the measurements taken from the runtime execution of the requested untrusted application, $\sigma 1 = Sign(H(CC||nonce_{AA}||MR), Tracer_{priv})$.
   - The second signature is to be consumed by the VPE, signing a Command Code (CC) that represents the execution of a specific command policy, the nonce that was issued by the VPE and the enclave measurement of the Tracer, $\sigma 2 = Sign(H(CC||nonce_{VPE}||MRTracer), Tracer_{priv})$
5. The VPE creates a fresh session and starts executing the policy enforcement algorithm in order to get access to his key, $RuntimePolicy = (00...00)$.

(a) Appends to the fresh session the measurement of the entity that signed his configuration manifest, $RuntimePolicy = H(RuntimePolicy||MRSigner)$.

(b) Appends to the session the measurement of the enclave application that is instantiating the VPE, $RuntimePolicy = H(RuntimePolicy||MREnclave)$.

(c) Verifies the signature of the Tracer that is corresponding to the VPE, by first recomputing the singed Digest, $SignedDigest = H(CC||nonce_{VPE}||MRTracer)$ and using the public key of the Tracer completes the verification process, $Verify(\sigma 2, SignedDigest, Tracer_{pub})$. If the verification is completed successfully the VPE computes the name of the Tracer's key, $Tracer_{Name} = H(Tracer_{pub}$, and appends it to the runtime policy, $RuntimePolicy = H(RuntimePolicy||Tracer_{Name})$.

(d) The VPE then uses the $RuntimePolicy$ to decrypt the encrypted VPE_{priv} that was injected to him by the IAM. The now decrypted value is hashed and then compared with the VPE_{HASH}. If these two digests match, the VPE can use its secret key.

6. With the acquired VPE_{priv} the VPE computes a digital signature over the nonce that was issued by the Attestation Agent and the approved measurement of the Attestation Agent, $\sigma 3 = Sign(H(CC||nonce_{AA}||MRAA), VPE_{priv})$.The $\sigma 1$ and $\sigma 3$ are forwarded to the Attestation Agent. The VPE_{priv} after the compilation of the signature gets destroyed.

7. The Attestation Agent creates a fresh session and starts executing the policy enforcement algorithm in order to get access to his key, $RuntimePolicy = (00...00)$.

(a) Appends to the fresh session the measurement of the entity that signed his configuration manifest, $RuntimePolicy = H(RuntimePolicy||MRSigner)$.

(b) Appends to the session the measurement of the enclave application that is instantiating the Attestation Agent, $RuntimePolicy = H(RuntimePolicy||MREnclave)$.

(c) Appends to the session the Security measurement that the Tracer extracted.$RuntimePolicy = H(RuntimePolicy||ReferenceValue_i)\forall i$ of the untrusted attested applications.

(d) Verifies the signature of the Tracer that is corresponding to the Attestation Agent , by first recomputing the singed Digest, $SignedDigest = H(CC||nonce_{AA}||ReferenceValue_i)\forall i$ of the untrusted attested applications and using the public key of the Tracer completes the verification process, $Verify(\sigma 2, SignedDigest, Tracer_{pub})$. If the verification is completed successfully the Attestation computes the name of the Tracer's key, $Tracer_{Name} = H(Tracer_{pub}$, and appends it to the runtime policy, $RuntimePolicy = H(RuntimePolicy||Tracer_{Name})$.

(e) Verifies the signature of the VPE, by first recomputing the signed Digest, $SignedDigest = H(CC||nonce_{AA}||MRAA)$ and using the public key of the VPE completes the verification process, $Verify(\sigma 3, SignedDigest, VPE_{pub})$.If the verification is completed successfully the Attestation computes the name of the VPE's key, $VPE_{Name} = H(VPE_{pub}$, and appends it to the runtime policy, $RuntimePolicy = H(RuntimePolicy||VPE_{Name})$.

(f) With the $RuntimePolicy$ the Attestation Agent will verify the Authorization Ticket that he acquired in the Join Phase, $Verify(RuntimePolicy, AuthorizationTicket, IAM_{pub})$. If the verification is completed successfully the $RuntimePolicy$ is resetted to $RuntimePolicy =$

$H(CC||IAM_{Name})$, where the CC is the command code of a specific policy command.

(g) The Attestation Agent will recompute his AK_{priv}, using the same KDF that was used during the Join Phase. The newly derived key gets hashed and compared with the AK_HASH. If these two digests match, then the Attestation Agent has successfully recreated his AK_{priv}.

8. The Attestation Agent uses his AK_{priv} to sign the initial challenge the Verifier has sent to him and then discards the AK_{priv}.

5.5 Constructing Zero-Knowledge Trustworthiness Claims

After the TCH has received the attestation attributes (together with the resulting trust opinions from the TAF), it will then proceed in constructing the necessary Verifiable Presentations [11] for proving the integrity of the trustworthiness claims to external entities (other vehicles or the MEC TAF) in an anonymous and privacy-preserving manner. This is achieved through a newly designed Threshold Anonymous DAA protoco by anonymising threshold Schnorr signatures, and extending them to meet the standard security definitions of DAA, as described in Section 5.2.4.

The operation with which the TCH will calculate a verifiable presentation including the neceseary trustworthiness claims can be seen in Figure 5.7. At a high level, after the TCH requests the generation of the trustworthiness claims, it will initiate a anonymous threshold signature protocol with the appropriate A-ECUs. Each A-ECU, based on its defined CIV policies, will retrieve a secret share with which they will participate in a threshold DAA scheme, as to calculate a digital signature over the trustworthiness claim requested and a challenge submitted by the TCH. After the TCH receives the responses from the A-ECUs, it will calculate a DAA signature by anonymising the threshold digital signature as well as the public key that validates it. It will then construct the ‘asymmetric-evidence‘ data structure described in Deliverable 5.1, Section 6.2.3.5, by setting the signature value to be the calculated DAA signature. Lastly, the TCH will combine it with the misbehaviour report received from the MD and the trust opinion received from the TAF to construct the final verifiable presentation to be returned to the IAM.

By constructing the ‘asymmetric-evidence‘ in this manner, the TCH can demonstrate in a privacy preserving manner, that at least a threshold number of A-ECUs are in the correct state, based on the defined CIV policies. The whole protocol is based on zero-knowledge proofs, meaning that the entity receiving the verifiable presentation will not be able to extract any additional information regarding the vehicle. In the rest of this Section we will present our anonymised threshold signature cryptographic protocol, together with details of its utilisation in the context of CONNECT.

5.5.1 High-Level Overview

The goal of our threshold DAA scheme is to distribute the DAA Signature generation operation among multiple participants. At a high level, other than the DAA key and credential, generation of a valid DAA signature will also require cooperation of at least t participants, where t is a predefined threshold. More precisely, in our system we will consider the following entities:

- DAA Member (M). Similar to the standard DAA protocol, a DAA Member (TPM, secure applications utilising TEE enabled microprocessors etc.,) being in possession of a DAA

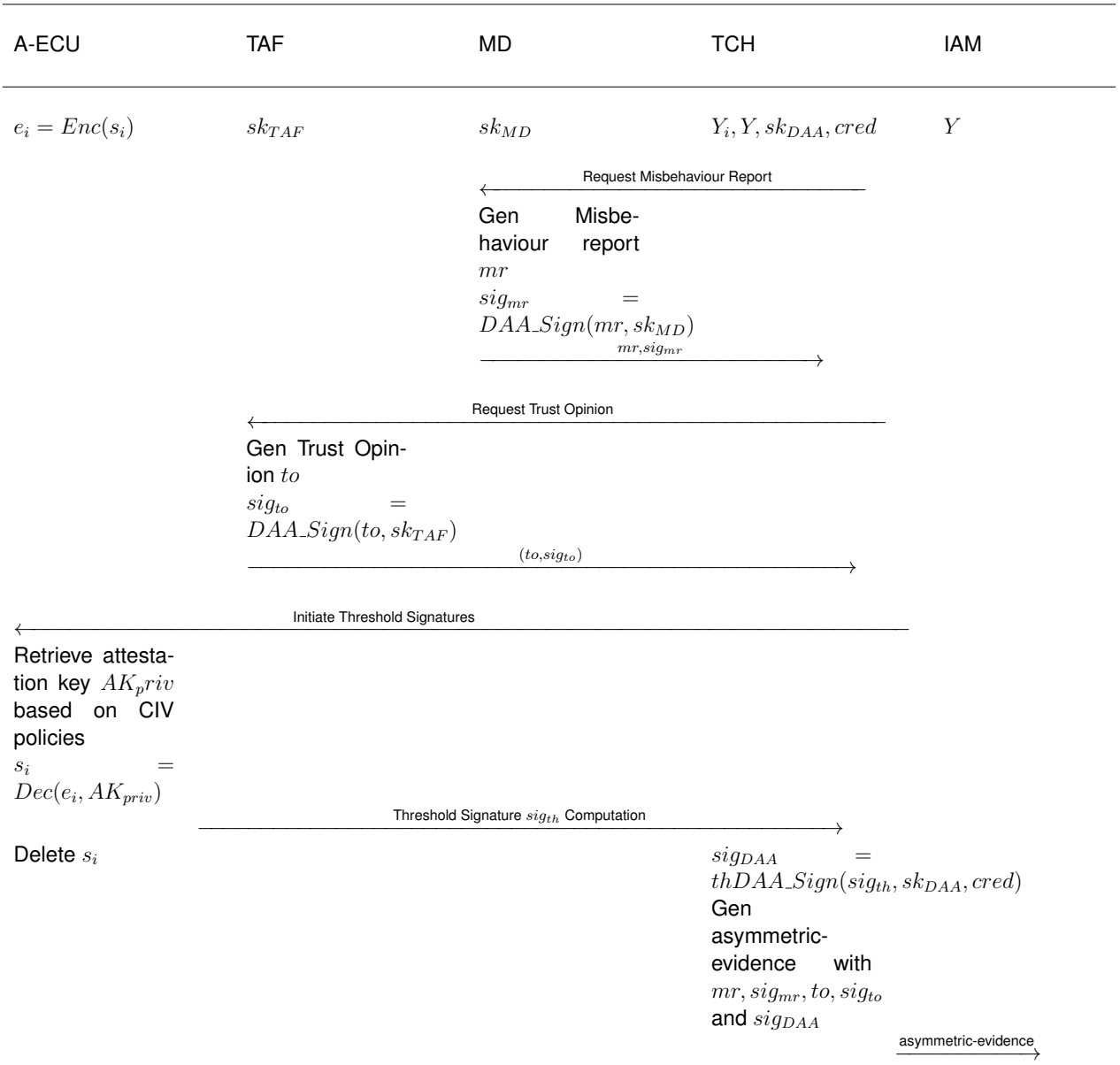

Figure 5.7: High-Level Flow of Actions of CONNECT Threshold Anonymous DAA scheme for constructing anonymous trustworthiness claims comprising the "harmonized" attestation attributes extracted from the Enhanced CIV mechanism

credential and key, will be the entity producing the DAA signature. In the context of CONNECT, that functionality will be realised by secure containers using Intel's SGX as the Root of Trust. When using our threshold DAA scheme, the role of the DAA Member will be played by the TCH.

- DAA Participant (P_i). In addition to the DAA Member, we will also consider multiple entities that will receive a secret share from the DAA Issuer. At least t participants P_i will need to correctly execute the threshold protocol and cooperate with M, for the DAA signature generation procedure to succeed. In the context of CONNECT, this role will be played by the A-ECUs.

At a high level, the standard DAA scheme is extended with the additional proof that M knows a threshold Schnorr signature and a public key, signed by the DAA Issuer, which validates that Schnorr signature.

5.5.1.1 DAA Join

During the DAA Join phase, M will generate a DAA key and receive a DAA credential (on that key) from the Issuer. Additionally, each participant P_i will get a secret share s_i from the DAA Issuer, which will then use in the threshold Schnorr signature computation.

More precisely, let us assume that n participants P_i will take part on the protocol. DAA Join will start with M generating a secret DAA key (sk_{DAA}), a commitment to that secret key $Comm_{sk}$ and a proof of correctness of that commitment PoC. It will send $Comm_{sk}$ and the PoC to the DAA Issuer, who, if able to verify the PoC, will generate a secret for the threshold scheme sk_{th} and corresponding public key Y. The Issuer will then divide sk_{th} into n shares $\{s_i\}_{i\in[n]}$, one for each of the n participants P_i (using Shamir secret sharing). They will then generate a credential $cred$ over Y and $Comm_{sk}$ which will be returned to M. Lastly, they will return s_i to each participant P_i and delete sk_{th}.

5.5.1.2 DAA Sign

To generate a signature on message m with random challenge c, the DAA member M will first execute the threshold signature computation protocol with k participants P_i, for $k \geq t$, to get a Schnorr signature on the challenge c. Note that at least t of the participants need to execute the protocol correctly for that signature to be valid. In our scheme we will use FROST, which is a 2-rounds Schnorr signature calculation protocol, with M as the aggregator. After receiving the responses from each P_i and calculating the Schnorr signature z on c, M will produce a DAA signature as follows:

- First, it will blind the threshold public key Y to get $\bar{Y}$. It will then generate a proof π_1 that $\bar{Y}$ "hides" a public key that is part of a valid credential received by the DAA Issuer. Then, M will generate a proof π_2 showcasing knowledge of a Schnorr signature z. Lastly, it will generate a proof π_3, showcasing that $\bar{Y}$ is an anonymised public key that correctly validates the Schnorr threshold signature z on the challenge c.
- Generate a signature proof of knowledge (SPK) on message m using the secret key sk_{DAA} and the DAA credential $cred$.

The returned DAA Signature will include the generated proofs of knowledge, the blinded Schnorr signature, and the blind commitment to the threshold signature's scheme group public key.

5.5.2 Preliminaries

We will use pairing friendly groups for type 3 pairings. A pairing, is an efficiently computable bilinear and non-degenerate map between two groups to a third group. Specifically, we will use the following constructions:

- Groups G_1, G_2, G_T, where G_1 and G_2 of prime order p and with no efficiently computable homomorphism between them. We will denote the pairing operation as $\hat{h} : G_1 \times G_2 \rightarrow G_T$.
- We will denote the identity point of G_* as 1_{G_*}. We will also reserve randomly sampled group points $g, g_0, h, h_0, q \in G_1$ and $g_2 \in G_2$ for use by our protocol. Those points are considered system parameters.
- We will use H to denote a hash function. We will assume that all calls to H are independent from each other. This can be achieved by using different domain separation tags or seeds during each call to H.

Following, we will define the threshold-DAA related operations utilised by our scheme. Later, we will present concrete instantiations for each function, as well as their usage during the Join and Sign phases of our threshold DAA scheme, or during the verification of the returned DAA signature.

- $\boldsymbol{Gen(1^{\lambda}) \rightarrow ps}$: Gen is a deterministic algorithm that given a security parameter λ, it returns a parameters set ps, including description of groups G_1, G_2 and G_T, the prime order p of G_1 and G_2, the pairing function $\hat{h}$, the hash function H and the points $g, g_0, h, h_0, q \in G_1$ and $g_2 \in G_2$.
- $\boldsymbol{thDAA\_Issue(ps, sk_I, C, Y) \rightarrow cred}$: $thDAA\_Issue$ is a randomised algorithm that given the parameters set ps, the issuance secret key sk_I, and 2 G_1 points C and Y (corresponding to the DAA secret key commitment C and threshold signature group public key Y), returns a DAA $cred$ including the point Y and the BBS signature (A, e) over C and Y (i.e., $cred = (Y, (A, e))$).
- $\boldsymbol{thDAA\_Sign(ps, sk_{DAA}, (R, \sigma), cred, m) \rightarrow sig_{DAA}}$: $thDAA\_Sign$ is a randomised algorithm that given the parameters set ps, DAA secret key sk_{DAA}, Schnorr signature (R, σ), DAA credential $cred$ and message m, returns a DAA signature sig_{DAA}.
- $\boldsymbol{thDAA\_Verify(ps, pk_I, sig_{DAA}, m) \rightarrow b}$: $thDAA\_Verify$ is a deterministic algorithm that given the parameters set ps, the issuing public key pk_I, the DAA signature sig_{DAA} returned by the DAA_Sign algorithm and a message m, returns a bit b, indicating if the signature is valid for the inputted message.

Table 5.5: Notations used in the description of the CONNECT threshold DAA scheme.

Notation	Description
TD	The Trusted Dealer of the threshold signature scheme.
S_i	The i'th Signer participating in the threshold signature scheme.
SA	The Signature Aggregator of the threshold signature scheme.
I_{DAA}	The trusted authority issuing DAA credentials.
P_i	The i'th participant of the threshold DAA scheme.
M	The DAA Member.
Del	The Delegated authority, able to update the threshold of the threshold DAA scheme.
sk_{DAA}	The DAA key, held by the DAA Member M.
sk_i	The i'th secret key share, held by P_i, used to generate Schnorr signature shares.
Y	The group public key of the threshold signature scheme.
Y_i	The public key of the i'th participant P_i, used to verify its Schnorr signature share.
sk_D	The secret key of the delegated authority Del.

PK_D	The public key of delegated authority Del.
sk_I	The secret key of the DAA Issuer I_{DAA}.
PK_I	The public key of the DAA Issuer I_{DAA}.
sig_{DAA}	The DAA signature produced by M.
ps	The parameter set for the threshold DAA scheme (curve definition, public points etc.,).
$cred$	The DAA credential issued by the DAA Issuer I_{DAA}.
m	The message signed by the DAA signature.
SPK	A signature proof of knowledge.

5.6 Anonymising Threshold Signatures

The main idea of our scheme is to use the FROST protocol described in Section 5.2.3, to generate a Shnorr signature $z = (R, \sigma)$ and then "hide" the public key Y which verifies z, by proving knowledge of both Y, R and σ such that $RY^c = g^\sigma$, (where c the challenge of the Schnorr signature), and of a DAA credential $cred$ over Y (among other things), signed by the DAA Issuer I_{DAA}. This allows proving the correct execution of the threshold protocol (in our case FROST), in a privacy preserving manner. For CONNECT, this will enable a vehicle to showcase integrity, i.e., the correct state of a threshold number of A-ECUs, in zero-knowledge, meaning without reviling any other information to external entities, since the (unique) public key Y and DAA credential $cred$ will never be reviled. Lastly, our scheme contributes to the zero-trust architecture of CONNECT, given that in our scheme, the only entity that we will consider trusted is the DAA Issuer I_{DAA}.

In our threshold DAA protocol, we consider the following entities; the set of participants P_i, a DAA Member M and the DAA Issuer I_{DAA}. Mapped to the entities participating in the FROST threshold signature protocol as described in Section 5.2.3, each P_i will execute the Signer's S_i functionality, the DAA Member M will execute the Signature Aggregator's SA functionality and finally the DAA Issuer I_{DAA} will play the role of the Trusted Dealer TD. Of course, each of these entities will be tasked with additional functionalities.

More formally, in our scheme, initially the DAA Issuer will generate an issuing key pair $(sk_I, PK_I = g_2^{sk_I})$ and publish the public key PK_I. I_{DAA} will also generate a private/public key pair $(sk_{th}, Y = g^{sk_{th}})$ for the threshold scheme. M will derive a DAA key sk_{DAA} and submit a commitment $C = h^{sk_{DAA}}$ to the Issuer. After going though an integrative proof of correctness of the commitment C, they will receive a credential issued by I_{DAA} over their key, as well as the public key of the threshold protocol Y. To generate the DAA credentials, we will use BBS Signatures [42]. Specifically, I_{DAA} will sample random scalar $e \xleftarrow{\$} \mathbb{Z}_p$ and calculate the credential's signature to be (A, e), where $A = (g_0 h^{sk_{DAA}} Y)^{1/(sk_I+e)}$. The final DAA credential will be $cred = (Y, (A, e))$.

To prove knowledge of such a credential, as well as a the Schnorr signature (R, σ) with challenge $c = H(R, PK_I, m)$, the DAA member M will sample random $\tilde{a}_1, \tilde{a}_2, \tilde{b} \xleftarrow{\$} \mathbb{Z}_p$ and set $\bar{Y} = Y^{\tilde{a}_2}$, $\bar{A} = A^{\tilde{a}_1\tilde{a}_2}$, $\bar{R} = (Rg^{\tilde{b}})^{\tilde{a}_2}$. M will also set $D = (g_0 * C)^{\tilde{a}_2}$ and $B = D^{\tilde{a}_1}\bar{Y}^{\tilde{a}_1}\bar{A}^{(-e)}$ (note that $B = \bar{A}^{sk_I}$) and $\tilde{a}_2' = 1/\tilde{a}_2$. Finally, M will set $\gamma = \tilde{a}_2(\sigma + \tilde{b})$ and calculate a signature proof of knowledge (SPK) π over a message m as follows;

$$\begin{aligned} \pi = SPK\{(\gamma, \tilde{a}_1, \tilde{a}_2', \tilde{b}, e, sk_{DAA}) : \\ R = \bar{R}^{\tilde{a}_2'} g^{-\tilde{b}} \wedge \bar{R}\bar{Y}^c = g^\gamma \wedge B = D^{\tilde{a}_1}\bar{Y}^{\tilde{a}_1}\bar{A}^{(-e)} \wedge g_0 = D^{\tilde{a}_2'} \cdot h^{-sk_{DAA}}\}(m) \end{aligned} \tag{5.1}$$

The final DAA Signature on m will be $sig_{DAA} = (R, \bar{R}, \bar{Y}, \bar{A}, D, B, \pi)$ (note that π is a signature

$thDAA\_Issue(ps, sk_I, C, Y) \rightarrow cred$	$thDAA\_Sign(ps, sk_{DAA}, (R, \sigma), cred, m) \rightarrow sig_{DAA}$
1. $(G_1, G_2, G_T, p, g, g_0, h, h_0, g_2, H) = ps$ 2. Sample random $e \xleftarrow{\$} \mathbb{Z}_p$ 3. $A = (g_0 CY)^{1/(sk_I+e)}$ 4. return $cred = (Y, (A, e))$	1. $(G_1, G_2, G_T, p, g, g_0, h, h_0, g_2, H) = ps$ 2. $(Y, (A, e)) = cred$ 3. sample $\tilde{a}_1, \tilde{a}_2, \tilde{b} \xleftarrow{\$} \mathbb{Z}_p$ 4. Set the following: (a) $\bar{Y} = Y^{\tilde{a}_2}$ (b) $\bar{A} = A^{\tilde{a}_1 \tilde{a}_2}$ (c) $\bar{R} = (Rg^{\tilde{b}})^{\tilde{a}_2}$ (d) $D = (g_0 * C)^{\tilde{a}_2}$ (e) $B = D^{\tilde{a}_1} \bar{Y}^{\tilde{a}_1} \bar{A}^{(-e)}$ (f) $\tilde{a}'_2 = 1/\tilde{a}_2$ and $\gamma = \tilde{a}_2(\sigma + \tilde{b})$ 5. $c = H(R, PK_I, m)$ 6. Sample random $r_\gamma, r_{a_1}, r_{a'_2}, r_{\tilde{b}}, r_e, r_{sk} \xleftarrow{\$} \mathbb{Z}_p$. 7. $T_1 = \bar{R}^{r_{a'_2}} g_0^{-r_{\tilde{b}}}$ 8. $T_2 = g^{r_\gamma}$ 9. $T_3 = D^{r_{a_1}} \bar{Y}^{r_{a_1}} \bar{A}^{-r_e}$ 10. $T_4 = D^{r_{a'_2}} h^{-r_{sk}}$ 11. $ch = H(R, \bar{R}, \bar{Y}, D, B, \bar{A}, T_1, T_2, T_3, T_4, m)$ 12. Set $\hat{s}_1 = r_\gamma + ch * \gamma$, $\hat{s}_2 = r_{a_1} + c * \tilde{a}_1$, $\hat{s}_3 = r_{a'_2} + ch * \tilde{a}'_2$, $\hat{s}_4 = r_{\tilde{b}} + ch * \tilde{b}$, $\hat{s}_5 = r_e + ch * e$ and $\hat{s}_6 = r_{sk} + ch * sk_{DAA}$. 13. return $sig_{DAA} = (R, \bar{R}, \bar{Y}, \bar{A}, D, B, \{\hat{s}_i\}_{i \in \{1,2,\ldots,6\}}, ch)$

Figure 5.8: Threshold DAA Issue and Sign operations.

proof of knowledge over m). To validate the signature, the Verifier will check if $\hat{h}(\bar{A}, PK_I) = \hat{h}(B, g_2)$, calculate the challenge $c = H(R, PK_I, m)$ and validate π. Note that our scheme proves knowledge of a Schnorr signature (R, σ), without directly showcasing that it was produced by a threshold signature scheme. The fact that (R, σ) was produced by the correct execution of the FROST protocol is guaranteed given that the public key that validates that signature is signed by I_{DAA}. Given that I_{DAA} signs the public key corresponding to the group $\{P_i\}_{i \in [n]}$, then M must correctly execute FROST, to get a signature valid under that public key. This allows our scheme to be threshold Schnorr signature protocol agnostic.

Analysis. Showcasing correctness of the above protocol is straight forward. We will quickly outline here that the described scheme is a proof of knowledge of a valid DAA credential, over a threshold group public key Y, which verifies the Schnorr signature (R, σ). Run the extractor of π to get $(\gamma, \tilde{a}_1, \tilde{a}'_2, \tilde{b}, e, sk_{DAA})$ so that all equations of π hold. We assume that all steps of the verification process complete successfully. From the fact that $\hat{h}(\bar{A}, PK_I) = \hat{h}(B, g_2)$ we get that $B = \bar{A}^{sk_I}$. Using that fact and the third equation of π we get that $\bar{A}^{sk_I+e} = D^{\tilde{a}_1} \cdot \bar{Y}^{\tilde{a}_1}$.

If $\tilde{a}_1 = 0$ since $\bar{A} \neq 1_{G_1}$ and $\bar{A}^{sk_I+e} = 1_{G_1}$, we can conclude that $sk_I = -e$. Such an extractor can be used to break discrete log in G_1. As a result we assume that $\tilde{a}_1 \neq 0$ meaning that

$thDAA\_Verify(ps, pk_I, sig_{DAA}, m) \rightarrow b$

1. $(G_1, G_2, G_T, p, g, g_0, h, h_0, g_2, H) = ps$
2. $(R, \bar{R}, \bar{Y}, \bar{A}, D, B, \{\hat{s}_i\}_{i\in\{1,2,\ldots,6\}}, ch) = sig_{DAA}$
3. If $\hat{h}(\bar{A}, PK_I) \neq \hat{h}(B, g_2)$, return 0.
4. $c = H(R, PK_I, m)$
5. $\hat{T}_1 = R^{-ch}\bar{R}^{\hat{s}_3}g^{-\hat{s}_4}$
6. $\hat{T}_2 = (RY^c)^{-ch}g^{\hat{s}_1}$
7. $\hat{T}_3 = (B)^{-ch}(D\bar{Y})^{\hat{s}_2}\bar{A}^{-\hat{s}_5}$
8. $\hat{T}_4 = g_0^{-ch}D^{\hat{s}_3}h^{-\hat{s}_6}$
9. Calculate $ch_v = H(R, \bar{R}, \bar{Y}, D, B, \bar{A}, \hat{T}_1, \hat{T}_2, \hat{T}_3, \hat{T}_4, m)$
10. If $ch \neq ch_v$, return 0.
11. return 1

Figure 5.9: Threshold DAA Verify operations.

$(\bar{A}^{1/\tilde{a}_1})^{sk_I+e} = D \cdot \bar{Y}$.

From the last equation of π we get that $D^{\tilde{a}'_2} = g_0 \cdot h^{sk_{DAA}}$. We assume that $\tilde{a}'_2 \neq 0$, since otherwise we will get that $g_0 = h^{-sk_{DAA}}$ and such an extractor could be used to solve DL in G_1. By setting $Y = \bar{Y}^{\tilde{a}'_2}$ we get that $(\bar{A}^{\tilde{a}'_2/\tilde{a}_1})^{sk_I+e} = g_0 h^{sk_{DAA}} Y$, meaning that $(\bar{A}^{\tilde{a}'_2/\tilde{a}_1}, e)$ is a valid DAA credential over the DAA key sk_{DAA} and a point Y.

All that remains is to show that $Y = \bar{Y}^{\tilde{a}'_2}$ is a valid public key that verifies a Schnorr signature known to the DAA Member M. From the first equation of π we get that $\bar{R} = (Rg^{\tilde{b}})^{1/\tilde{a}'_2}$. From the second equation of π we get that $Rg^{\tilde{b}}(\bar{Y}^{\tilde{a}'_2})^c = g^{\gamma * \tilde{a}'_2} => RY^c = g^{\gamma * \tilde{a}'_2 - \tilde{b}}$. Setting then $\sigma = \gamma * \tilde{a}'_2 - \tilde{b}$ we get that (R, σ) is a valid Schnorr signature for the public key $Y = \bar{Y}^{\tilde{a}'_2}$, which is signed by the DAA Issuer.

5.6.1 Enhanced Direct Anonymous Attestation (DAA) Commitments

In the following Sections we will describe the Join, Sign and Verify operations of our DAA Scheme, as shown in Figures 5.10 and 5.11. The main difference with a standard, non-threshold, DAA protocol, is that during the Sign procedure, at least t participants P_i need to correctly execute the protocol, for the final outputted DAA signature to be valid. Our scheme will use the $thDAA\_Issue$, $thDAA\_Sign$ and $thDAA\_Verify$ operations, as shown in Figures 5.8 and 5.9, which follow the method described in Section 5.6. Note that we consider the implementation of the Gen algorithm, described in Section 5.5.2, to be implementation specific and outside the scope of this document. In this Section, we assume that the parameters set ps has already be generated and distributed to all involved parties.

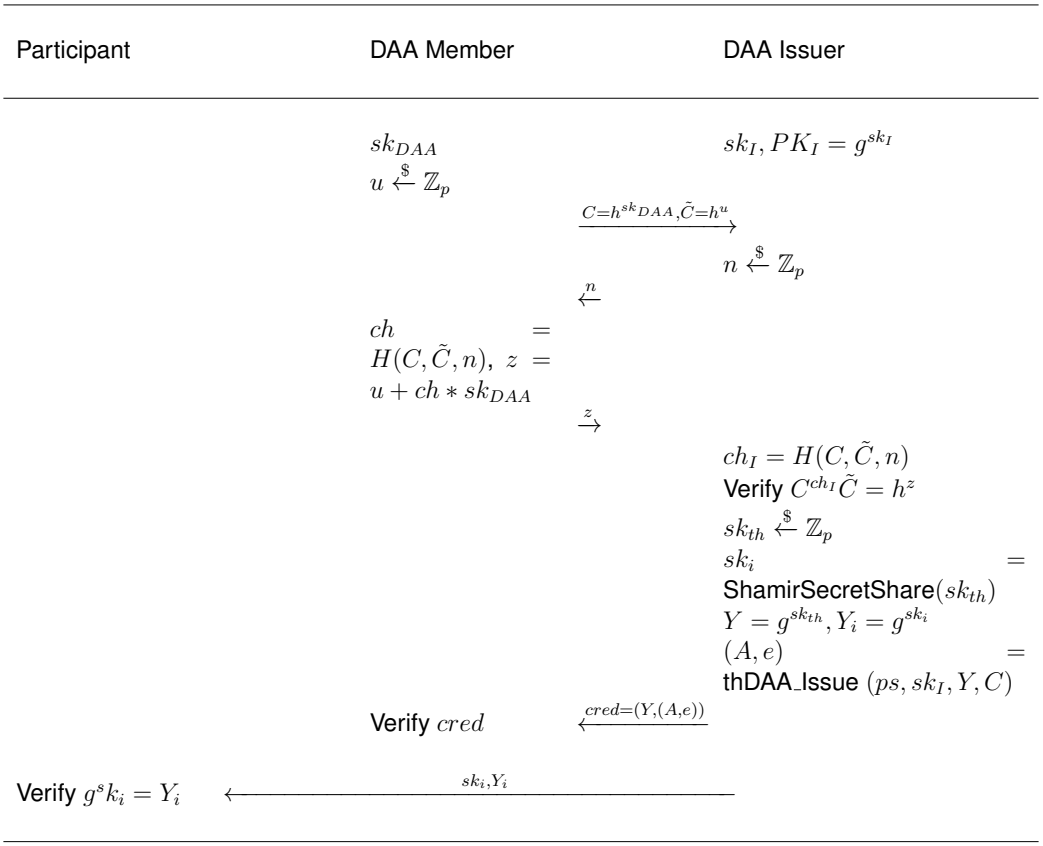

Figure 5.10: Threshold DAA Join.

5.6.1.1 Join

During the Join phase, M will get a credential over a secret key that they hold. In addition, each participant P_i will get a secret key share sk_i, with which they will participate to the execution of the threshold protocol.

To start the procedure, M will follow the steps bellow;

1. M will generate its DAA key, sk_{DAA}, together with a commitment $C = h^{sk_{DAA}}$ on that key. They will then execute an interactive protocol with I_{DAA}, as to prove knowledge of the sk_{DAA} and correctness of the commitment C. The steps are the following;
   (a) M will choose random scalar $v \xleftarrow{\$} \mathbb{Z}_p$ and send $(C, \tilde{C} = h^v)$ to I_{DAA}.
   (b) Upon receiving the tuple $(C, \tilde{C})$, I_{DAA} will check that it has a valid value (i.e., that $C, \tilde{C} \in G_1 \setminus \{1_{G_1}\}$) and if true, choose random nonce $n \xleftarrow{\$} \mathbb{Z}_p$ to return to M.
   (c) After receiving n, M will calculate $ch = H(C, \tilde{C}, n)$ and respond with $z = v + ch * sk_{DAA}$.
   (d) Finally, after I_{DAA} receives M's response, will calculate $ch_I = H(C, \tilde{C}, n)$ and verify the proof by checking if $C^{ch_I}\tilde{C} = h^z$.
2. If the above protocol executes successfully, I_{DAA} will first set up the Threshold Signature protocol by generating a random $sk_{th} \xleftarrow{\$} \mathbb{Z}_p$ and using Shamir secret sharing to divide it into n shares $\{sk_i\}_{i\in[n]}$ with a threshold t for re-calculating the original secret sk_{th}.

3. I_{DAA} will calculate a credential $cred$ to sign the Threshold Signature's group public key $Y = g^{sk_{th}}$ and M's DAA Key commitment C. To do that, I_{DAA} will use the $thDAA\_Issue$ operation described in Figure 5.8, to get $(Y, (A, e)) = thDAA\_Issue(ps, sk_I, C, Y)$. I_{DAA} will then return the credential $cred = (Y, (A, e))$ and all the $\{Y_i = g^{sk_i}\}_{i\in[n]}$ to M. Additionally, I_{DAA} will send each secret key share sk_i to each participant P_i through a secure and authenticated channel (setting up such a channel is outside the scope of this document).

4. Lastly, M will verify the received credential $cred = (Y, (A, e))$ by checking if $\hat{h}(A, PK_I g_2^e) \stackrel{?}{=} \hat{h}(g_0 YC, g_2)$.

If the above protocol executes correctly, each P_i will have receive a secret key share sk_i with which they can participate to the execution of a Threshold Signatue protocol and M will have received a credential $cred$, over the their DAA secret key sk_{DAA} and the group public key Y.

5.6.1.2 Sign

To sign a message m, the steps are the following;

1. First, M will execute a Threshold Schnorr Signature scheme, (in our case FROST) with $\{P_i\}_{i\in S}$ to calculate a valid Schnorr signature $z = (R, \sigma)$ on m. The main difference between a standalone execution of FROST, is that in our threshold DAA scheme the calculation of the Schnorr signature challenge will substitute the group's public key Y with the public key PK_I of the DAA Issuer. More precisely, the challenge c will be calculated as $c = H(R, PK_I, m)$. Although that change normally would be insecure, since it would allow the forging of a Schnorr signature, valid under a different public key. We solve this issue, by proving that the public key which validates the Schnorr signature is signed by I_{DAA}. Note that if z is valid, it means that $Y^c R = g^\sigma$.

2. If the Threshold protocol executes correctly, M will use the procedure described in Section 4, to generate the DAA signature using $sig_{DAA} = thDAA\_Sign(ps, sk_{DAA}, (R, \sigma), cred, m)$, where $sig_{DAA} = (R, \bar{R}, \bar{Y}, \bar{A}, D, B, \pi)$ over m, showcasing knowledge of a credential $cred$ signed by I_{DAA}, over a secret key that they possess sk_{DAA} and a public key Y that correctly validates a Schnorr signature known to M.

5.6.1.3 Verify

To verify a DAA signature sig_{DAA} on m, the steps are the following;

1. The Verifier will first have to parse sig_{DAA} to get $(R, \bar{R}, \bar{Y}, \bar{A}, D, B, \pi)$ and check that all points (i.e., $R, \bar{R}, \bar{Y}, \bar{A}, D, B$ and all points in π) are in $G_1 \setminus \{1_{G_1}\}$ and that all scalars (i.e., the scalars in π) are in $\mathbb{Z}_p \setminus \{0\}$.

2. They will check if $\hat{h}(\bar{A}, PK_I) \stackrel{?}{=} \hat{h}(B, g_2)$, calculate $c = H(R, PK_I, m)$ and verify π.

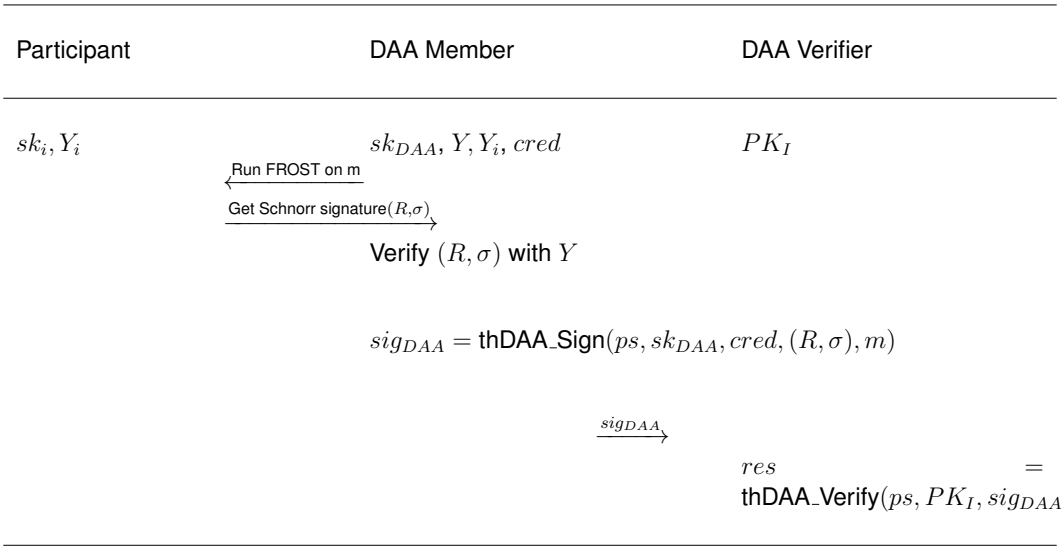

Figure 5.11: Threshold DAA Sign.

5.7 Threshold Update Delegation

We will present here a protocol with which I_{DAA} will delegate to a third authority (Del) the ability to update the threshold in the Threshold Schnorr signature scheme. At a high level, I_{DAA} will set a minimum threshold t for the threshold signature protocol. Del will then be able to coordinate with each P_i, as to generate a new instantiation of the threshold protocol, with threshold $t' \geq t$. To guarantee that only the designated entity will be able to update the threshold scheme, Del will generate a key pair and execute a flow similar to DAA Join described in Section 5.6.1.1. If successful, I_{DAA} will include Del's secret key (or rather a commitment to that secret key) to the DAA credential $cred$, returned to M. At a high level, the steps to generate a DAA signature are the following:

- **Updating the threshold:** Del will send update values to each P_i, which they will use to update their secret key shares. Del will also calculate the updated public key Y' of the new threshold scheme, which they will send signed to M.
- **Blinding the public key:** After receiving the Y' and Del's signature, M will commit and blind Del's public key. They will then generate a proof of knowledge of Del's signature, over the updated public key Y', valid under the committed public key.
- **Generating the signature:** Finally, M will generate a proof of knowledge of a Schnorr signature, valid under Y' and a credential issued by I_{DAA}, over their DAA key and Del's public key. Note that the verifier of the DAA Signature will never see either the public key Y' of the threshold scheme (M will prove knowledge of a signature from Del over Y'), neither the public key of Del (M will commit to their public key and prove that it is signed by I_{DAA}).

The above procedure showcases that M is in possession of a valid threshold signature, generated by a Schnorr threshold signature protocol, instantiated by I_{DAA} and updated by a proper (i.e., certified) delegated authority Del. The verifier of the signature will still be unable to extract any additional information (for example the identity of Del or the threshold value t).

In CONNECT, the TAF will undertake the role of the delegated authority Del, where, based on runtime measurements, will update threshold of our DAA scheme as needed by different applications.

5.7.1 Updating the Threshold

Let the delegate authority Del requesting an update to a threshold t'. Note that the new threshold t' can only be greater or equal than the original threshold t set by I_{DAA}. Del will coordinate with each P_i in order for the threshold to be updated. They will then compute the new group public key Y', which they will provide signed to M. We will note that any entity can coordinate an update to a new threshold t'. Our scheme is based on the fact that DAA Sign will not work unless the update is coordinated by a authority "trusted" by I_{DAA}. That said, it would be a good practice for each P_i to also authenticate Del during the threshold update phase. We will omit such details in this version of the document.

To create an instantiation of the threshold protocol scheme, with threshold $t' \geq t$ the steps are the following:

1. Let Del choose random polynomial P' of degree t' so that $P'(x) = s' + a_{t+1}x^{t+1} + a_{t+2}x^{t+2} + ... + a_{t'}x^{t'}$ for random $\{s', a_{t+1}, a_{t+2}, ..., a_{t'}\} \xleftarrow{\$} \mathbb{Z}_p$.
2. For each participant P_i, Del will calculate the "update value" $\triangle_i = P'(i)$. To calculate the updated group public key Y', Del will compute $Y' = Yg^{s'}$.
3. After each P_i receives $\triangle_i$ they will updates their secret key shares to get $sk'_i = sk_i + \triangle_i$.

Note that, if we set $F = P + P'$ (where P the original t-degree polynomial used by I_{DAA} to divide the secret sk_{th} to n shares) the new secret key share of P_i is $sk'_i = F(i)$, where F of degree t' so that $F(0) = sk_{th} + s'$. Additionally, $Y' = g^{sk_{th}+s'}$. So each sk'_i is a valid secret key share for a threshold Schnorr signature with threshold t' and Y' is a valid group public key for that scheme.

In the following Sections, We will present an extension of the DAA protocol described above, as to allow M to also prove knowledge of a signature from Del, on the updated threshold signature group public key Y'. To do that, we will require from Del to also execute a protocol similar to the DAA Join procedure described in Section 5.6.1.1. More specifically, Del will generate a key pair $(sk_D, PK_D = g_2^{sk_D})$, and then commit to the their secret key with $Q = q^{sk_D}$. To prove correctness of that commitment to the I_{DAA}, the procedure is the same as the one detailed at step 1 of the DAA Join operation, described in Section 5.6.1.1. Del will then send Q and their public key PK_D to M, who will forward them to I_{DAA}, together with their own commitment $C = h^{sk_{DAA}}$ and proof of correctness to their DAA secret key sk_{DAA}. I_{DAA} will generate the threshold signature group key pair $(sk_{th}, Y = g^{sk_th})$ and sign everything together to get the credential (A, e), where $A = (g_0 \cdot Q \cdot C \cdot Y)^{1/(sk_I+e)}$. I_{DAA} will return $(Y, (A, e))$ to M and Y to Del.

After Del updates the threshold and calculates the new group public key Y' as mentioned above, they will sign Y' by calculating (A_D, e_D) where $A_D = (g_0 \cdot Y')^{1/(sk_D+e_D)}$ for randomly sampled $e_D \xleftarrow{\$} \mathbb{Z}_p$. They will send $Y', s', (A_D, e_D)$ to M who during the DAA Sign phase of our protocol, in addition to the procedure described in Section 5.6.1.2, will also have to prove knowledge of a signature from Del, without of course reveling Del's public key PK_D. They will do that by committing to Del's public key, and simultaneously showcasing that the committed public key is signed in their DAA credential (A, e) and that they know a valid signature under a committed public key.

5.7.2 Blinding the Public key

In this Section, we will consider the more general case, where a prover generates a zero-knowledge proof of knowledge of a valid signature from an issuer to a verifier, without revealing the issuer's public key. As a signature, we will use a BBS signature (A, e, s) over a point P with $A = (g_0 h^s P)^{1/(x+e)}$ for random $e, s \xleftarrow{\$} \mathbb{Z}_p$, where x the secret key of the issuer of the signature. Let $PK = g_2^x$ be the public key that validates that signature. Notice that we re-introduce the randomness s as part of the signature, as described in [TBD], in contrast to the construction of [TBD], where is proven that even signatures without the s value are secure. The reason will become apparent later, but at a high level, in our construction the s value is used to guarantee zero-knowledge without relying to computational assumptions.

In our scheme, the prover will commit to the Issuer's public key, and then calculate a zero-knowledge proof of knowledge of a signature, valid under the committed public key. For our use case, during the proof of knowledge calculation, the "signed point" P will not be disclosed.

To that end, first the prover will choose random r_1, r_2, set $r_i' = 1/r_i$, $i = 1, 2$ and calculate $\bar{PK}_D = PK_D^{r_1'}$ to blind the public key. To blind the signature, they will set $\bar{A} = A^{r_1 r_2}$, $\bar{P} = P^{r_2}$ and $B_1 = (g_0 h_0^s)^{r_2} \bar{P} \bar{A}^{-r_1' e}$ $(= (\bar{A}^{r_1})^{sk_D})$. Then, they will choose random $r_3 \xleftarrow{\$} \mathbb{Z}_p$, set $D_1 = (g_0 h^s)^{r_2} h^{r_3}$, $r' = r_1' r_2'$ and $s' = s - r_2' r_3$.

Finally, the prover will sample $r_4 \xleftarrow{\$} \mathbb{Z}_p$, calculate $K = \bar{A}^{r_1'} h^{r_4}$, set $r_4' = r_4 e - r_3$ and calculate the signature proof of knowledge π_1 as follows,

$$\begin{aligned} \pi_1 = SPK\{(e, r_4', r_1', r_4, r_2', s'), \\ B_1 = D_1 \bar{P} K^{-e} h^{r_4'} \wedge K = \bar{A}^{r_1'} h^{r_4} \wedge g_0 = D_1^{r_2'} h^{s'}\} \end{aligned} \tag{5.2}$$

The final proof will be $(\bar{PK}, B_1, D_1, \bar{A}, \bar{P}, K, \pi_1)$. To validate the proof, the Verifier will check if $\hat{h}(\bar{A}, \bar{PK}) = \hat{h}(B_1, g_2)$ and verify π_1.

Analyses: We will give here an overview of the security proof, showcasing that the above procedure is a proof of knowledge of a valid signature under the public key PK committed in $\bar{PK}$. Run the extractor of π_1 to get $(e, r_4', r_1', r_4, r_2', s')$ so that all equations of π_1 hold. Let δ be the discrete logarithm of $\bar{PK}$ on the basis of g_2 (i.e., $\bar{PK} = g_2^{\delta}$). From the fact the $\hat{h}(\bar{A}, \bar{PK}) = \hat{h}(B_1, g_2)$, we can conclude that $\bar{A}^{\delta} = B_1$.

From the last equation of π_1 we get that $g_0 = D_1^{r_2'} h^{-s'}$. We assume that $r_2' \neq 0$ (otherwise for the extracted s' it will hold that $g_0 = h^{-s'}$, and that extractor will be able to solve DL in G1). As a result, we can deduce that $D_1 = (g_0 h^{s'})^{1/r_2'}$. Substituting D_1 in the first equation of π_1 we get

$$\bar{A}^{\delta} = (g_0 h^s)^{1/r_2'} \bar{P} K^{-e} h^{r_4'} \tag{5.3}$$

From the second equation of π_1, we get that $K = \bar{A}^{r_1'} h^{r_4}$. Substituting K in 5.3 we get that,

$$\bar{A}^{\delta} = (g_0 h^s)^{1/r_2'} \bar{P} \bar{A}^{-e r_1'} h^{r_4' - e r_4} \tag{5.4}$$

We will distinguish between two cases; $r_1' = 0$ and $r_1' \neq 0$.

$\boldsymbol{r_1' = 0}$**.** From 5.4, we get $\bar{A}^{r_2' \delta} = g_0 h^{s'} \bar{P}^{r_2'} h^{r_2'(r_4' - e r_4)}$. Setting, $\hat{s} = s' + r_2'(r_4' - e r_4)$, $\hat{e} = 0$, $\hat{P} = \bar{P}^{r_2'}$ and $\hat{A} = \bar{A}^{r_2'}$, we get that $\hat{A}^{\delta + \hat{e}} = g_0 h^{\hat{s}} \hat{P}$, meaning that $(\hat{A}, \hat{e})$ is a valid signature generated from

the secret key δ, over the point $\hat{P}$, committed in $\bar{P}$, valid under the public key $\hat{PK}$, committed in $\bar{PK}$.

$\boldsymbol{r'_1 \neq 0}$**.** From 5.4 we get that,

$$(\bar{A}^{r'_1})^{\delta/r'_1+e} = (g_0 h^s)^{1/r'_2} \bar{P} h^{r'_4 - e r_4} =>$$
$$(\bar{A}^{r'_2 r'_1})^{\delta/r'_1+e} = g_0 h^{s+r'_2(r'_4-er_4)} \bar{P}^{r'_2}$$

Setting $\hat{s} = s + r'_2(r'_4 - er_4)$, $\hat{\delta} = \delta/r'_1$, $\hat{e} = e$, $\hat{P} = \bar{P}^{r'_2}$, $\hat{\delta} = \delta/r'_1$, $\hat{PK} = \bar{PK}^{1/r'_1}$ and $\hat{A} = \bar{A}^{r'_2 r'_1}$, we get that $\hat{A}^{\delta+\hat{e}} = g_0 h^{\hat{s}} \hat{P}$, meaning again that $(\hat{A}, \hat{e})$ is a valid signature generated from the secret key $\hat{\delta}$, over the point $\hat{P}$, committed in $\bar{P}$, valid under the public key $\hat{PK}$, committed in $\bar{PK}$.

As a result of independent interest, the above methodology can be used be used to straighten the privacy properties of an anonymous credential system, by allowing the Prover to hide the Issuer of a credential.

5.7.3 Generating the Signature

To generate a DAA signature, M will combine the techniques described in Section 5.6 and 5.7.2, to prove knowledge of a Schnorr signature (R, σ) valid under a public key Y', which is signed by a proper delegated party (i.e., an entity whose public key is included in the DAA credential, created by I_{DAA}).

Lets assume that M, $\{P_i\}_{i\in[n]}$ and I_{DAA} executed the DAA Join protocol described in Section 5.6.1.1, with the addition of Del's inputs, as described in Section 5.7.1. Upon successful resolution, each P_i, $i \in [n]$, will have received a secret key s_i from I_{DAA}, while M will have received the DAA credential $(Q, Y, (A, e))$, where $A = (g_0 h^{sk_{DAA}} QY)^{1/sk_I+e} = (g_0 h^{sk_{DAA}} q^{sk_D} Y)^{1/sk_I+e}$, where sk_{DAA} the DAA secret key of M, $Q = q^{sk_D}$ the commitment to Del's secret key (sk_D) and Y the public key of the initial Threshold Signature Scheme, initiated by I_{DAA} (with threshold t).

After Del updates the threshold, they will communicate to M all the necessary values to generate the DAA signature for the new threshold. Then, during the DAA Sign phase, M will initiate the threshold signature scheme with the chosen P_i. At the end, M will have;

- A Schnorr signature $z = (R, \sigma)$.
- The public key Y', validating z.
- A signature from Del, over that public key (Y').
- A credential from I_{DAA}, over the secret key of Del and the DAA secret key of M.

M needs to prove knowledge of those elements, without disclosing any of them. To that end, they will create a commitment $\bar{Y'}$ to the public key of the threshold scheme Y' and a commitment $\bar{PK}_D$ on Del's public key PK_D. Then they will use the techniques described in the previous Sections as follows:

- **Instantiation of threshold scheme by trusted entity.** To prove that z is produced by a threshold signature scheme whose threshold is updated by a trusted (by I_{DAA}) entity (i.e., Del), M will execute the following steps,

  - Prove knowledge of the signature (A_D, e_D) over the public key that validates z (committed in $\bar{Y'}$).
  - Prove knowledge of a DAA credential, over the secret key corresponding to the public key committed to $\bar{PK}_D$, used to generate (A_D, e_D).

- **Correct threshold scheme execution.** M will prove knowledge of z and validity of z under the public key committed in $\bar{Y'}$.

The above steps achieve two goals. First, they establish the trust relationship between the involved parties, i.e., the fact that I_{DAA} has certified both Del and M, one to update the threshold scheme and the other to produce DAA signatures. Second, they prove that the protocol is executed correctly, i.e., that M has coordinated correctly with at least t' participants P_i to produce the DAA signature, where t' is set by Del and is greater or equal to the minimum threshold established by I_{DAA}. In the rest of this Section, we will describe the above steps in detail,

Instantiation of threshold scheme by trusted entity. To prove that the threshold of the Threshold Signature scheme is set by a certified delegated authority Del, M will commit to Del's public key and prove knowledge of a signature (A_D, e_D) valid under the committed public key.

To that end, M will execute the procedure described in Section 5.7.2, setting $(A, e) = (A_D, e_D)$, $PK = PK_D$ and $P = Y'$. Note that this will prove knowledge of a signature valid under a public key committed in $\bar{PK}_D$, over the point committed in $\bar{Y'}$ (see bellow).

We will use those facts later to prove that the entity that updated the threshold scheme is certified by I_{DAA}, by showing that M's DAA credential contains a commitment to the same value as $\bar{PK}_D$ (i.e., the secret key of Del), and that the point they signed (committed in $\bar{Y'}$) is a public key that verifies the Schnorr threshold signature z.

Specifically, M will choose random $r_1, r_2 \xleftarrow{\$} \mathbb{Z}_p$, set $r'_i = 1/r_i$, $i = 1, 2$ and calculate $\bar{PK}_D = PK_D^{r'_1}$ to blind the public key. To blind the signature, M will set $\bar{A}_D = A_D^{r_1 r_2}$, $\bar{Y'} = Y'^{r_2}$ and $B_1 = (g_0 h_0^s)^{r_2} \bar{Y'} \bar{A}_D^{-r'_1 e_D}$ $(= (\bar{A}^{r_1})^{sk_D})$. Then, M will choose random $r_3, r_4 \xleftarrow{\$} \mathbb{Z}_p$, set $D_1 = (g_0 h^s)^{r_1 r_2} h^{r_3}$, $K = \bar{A}^{-e/r_1} h^{r_4}$, $r' = r'_1 r'_2$ and $s' = s - r' r_3$, $r'_4 = r_4 r_1$. Finally, M will calculate the signature proof of knowledge π_1 as follows,

$$\begin{aligned} \pi_1 = SPK\{(e, r'_4, r'_1, r_4, r'_2, s'), \\ B_1 = D_1 \bar{Y'} K h^{-r_4} \wedge 1_{G_1} = K^{r_1} \bar{A}^e h^{-r'_4} \wedge g_0 = D_1^{r'_2} h^{-s'}\} \end{aligned} \tag{5.5}$$

Note that the DAA credential also "signs" the point $Q = q^{sk_D}$. To showcase that $\bar{PK}_D$ and the DAA credential contain a commitment to the same value (sk_D, meaning that the threshold is updated by a certified entity), M will first generate a commitment to Q by choosing random $a_1 \xleftarrow{\$} \mathbb{Z}_p$ and setting $\bar{Q} = Q^{a_1}$. M needs to show that they know an opening of the commitment $\bar{Q}$, which will it self be a commitment to the secret key sk_D that used to calculate the signature (A_D, e_D) (but not knowledge of that value, since M doesn't know sk_D). M will do this by showcasing that the opening of the commitment $\bar{Q}$, commits to the same value (sk_D) as the opening of the commitment $\bar{PK}_D$. Later, M will also showcase that the aforementioned opening of the commitment $\bar{Q}$, is signed by their DAA credential $cred$ (showing that way that the threshold is updated by a certified authority, or more precisely, by a certified public key).

To that end, M will set $a'_1 = 1/a_1$ and $\hat{PK}_D = \bar{PK}_D^{1/a'_1 r'_1}$ $(= PK_D^{a_1})$ and $\hat{\bar{PK}}_D = \hat{PK}_D^{r'_1}$. They will then calculate the following proof of knowledge,

$$\pi_2 = SPK\{(a'_1, r'_1), \bar{PK}_D = \bar{PK}_D^{a'_1} \wedge \hat{PK}_D = \hat{PK}_D^{r'_1}\} \tag{5.6}$$

The returned proof will be $(\bar{Q}, \hat{PK}_D, \bar{\hat{PK}}_D, \pi_2)$. Note that we will need to calculate π_1 and π_2 together, as to prove that the r'_1 that M proves knowledge of in π_1 is the same with the one in π_2. Additionally, M will need to prove knowledge of a Schnorr signature (R, σ), valid under Y'. To that end, M will shamble random $\tilde{a}$ and set $\bar{R} = (Rg^{\tilde{a}})^{r_2}$ and $\gamma = r_2(\sigma + \tilde{a})$. M will then extend π_1 to the following proof,

$$\begin{gathered}\pi_{1,2} = SPK\{(e_D, r'_1, r'_4, r', s', r_4, a'_1, \gamma, \tilde{a}) : \\ B_1 = D_1\bar{Y'}Kh^{-r_4} \wedge 1_{G_1} = K^{r_1}\bar{A}_D^{e_D}h^{-r'_4} \wedge g_0 = D_1^{r'}h^{-s'} \\ \wedge \bar{PK}_D = \bar{PK}_D^{a'_1} \wedge \bar{\hat{PK}}_D = \hat{PK}_D^{r'_1} \\ \wedge R = \bar{R}^{r'_2}g^{-\tilde{a}} \wedge \bar{R}\bar{Y'}^c = g^{\gamma}\}\end{gathered} \tag{5.7}$$

To verify that $\bar{Q}$ and $\bar{PK}_D$ are commitment to the same value, the verifier will check if $\bar{\hat{PK}}_D \neq 1_{G_2}$ and if $\bar{PK}_D \neq 1_{G_2}$, validate π_2 and then check that $\hat{h}(\bar{Q}, g_2) = \hat{h}(q, \hat{PK}_D)$.

Analyses. We will give here a quick overview for the correctness of the above protocol. Set δ to be the discrete log of $\bar{PK}_D$ in the base of g_2, i.e., $\bar{PK}_D = g_2^{\delta}$. Running the extractor described in Section 5.7.2, we get a scalar r'_1 and a signature (A_D, s_D, e_D) valid under a public key PK_D (known by M).

By running the extractor on the extended proof $\pi_{1,2}$, we also get a'_1 (in addition to all the other values) such that $\bar{PK}_D = \bar{\hat{PK}}_D^{a'_1}$ and $\bar{\hat{PK}}_D = \hat{PK}_D^{r'_1}$. From that we can conclude that $r'_1 \neq 0$, since $\bar{\hat{PK}}_D \neq 1_{G_2}$ and that $a'_1 \neq 0$ since $\bar{PK}_D \neq 1_{G_2}$. Additionally, it follows that $\bar{PK}_D = \hat{PK}_D^{r'_1 a'_1}$. Lets denote with d_1 the discrete log of $\hat{PK}_D$ in the base of g_2 (i.e., $\hat{PK}_D = g_2^{d_1}$). We can conclude that $\hat{PK}_D = g_2^{d_1} = g_2^{\delta/r'_1 a'_1} => d_1 = \delta/r'_1 a'_1$. From the fact that $\hat{h}(\bar{Q}, g_2) = \hat{h}(q, \hat{PK}_D)$, we can conclude that $\bar{Q} = q^{d_1} = q^{\delta/r'_1 a'_1}$. Set $Q = \bar{Q}^{a'_1}$ $(= q^{\delta/r'_1})$. From the analyses of Section 5.7.2, and from the fact that $r'_1 \neq 0$, its easy to see that the secret key that generated the signature (A_D, s_D, e_D) is $sk_D = \delta/r'_1$ and $Q = q^{sk_D}$. Later, M will also prove that the extracted Q value is included in their DAA credential.

Additionally, similar to the analyses in Section 5.6, the extractor will return $(\gamma, a'_1, \tilde{a}, e_D, sk_{DAA})$ so that all equations of π hold. We assume that all steps of the verification process complete successfully. From the analyses in Section 5.7.2, we know that the extractor can return signature (A_D, s_D, e_D) over the point $Y' = \bar{Y'}^{r'_2}$, valid under the public key PK_D, committed to $\bar{PK}_D$.

All that remains is to show that $Y' = \bar{Y}^{r'_2}$ is a valid public key that verifies a Schnorr signature known to the DAA Member M. From the sixth equation of $\pi_{1,2}$ we get that $\bar{R} = (Rg^{\tilde{a}})^{1/r'_2}$. From the seventh equation of $\pi_{1,2}$ we get that $Rg^{\tilde{a}}(\bar{Y'}^{r'_2})^c = g^{\gamma * r'_2} => RY'^c = g^{\gamma * r'_2 - \tilde{a}}$. Setting then $\sigma = \gamma * r'_2 - \tilde{a}$ we get that (R, σ) is a valid Schnorr signature for the public key $Y' = \bar{Y}^{r'_2}$, which is signed by the delegated authority Del.

Correct threshold scheme execution. To generate the DAA signature, after following the steps outlined above, M will also follow the same procedure described in 5.6, to prove knowledge of a (threshold) Schnorr signature $z = (R, \sigma)$, a public key Y' that validates that signature and of a signature $sig = (A_D, s_D, e_D)$ over that public key Y'. The difference with the protocol described in Section 5.6 is that the signature sig will be produced by Del and not I_{DAA}. As mentioned above,

M will have to follow the procedure described previously to showcase that their DAA credential, includes a commitment to the secret key of Del, as to prove that Y' is set by a certified entity.

At this point, M will have generated a proof $proof_{1,2} = (\bar{PK}_D, \hat{\bar{PK}}_D, B_1, D_1, \bar{A}_D, \bar{Y}', K, R, \bar{Q}, \bar{R}, \pi_{1,2})$, showing knowledge of a signature (A_D, s_D, e_D) over a point $Y' = \bar{Y}'^{r_2'}$, for some known to M scalar r_2'. Additionally, $proof_1$, showcase knowledge of a point $Q = \bar{Q}^{a_1'} = q^{sk_D}$, where sk_D the secret key that was used to calculate the signature (A_D, s_D, e_D).

What remains know is for M to prove knowledge of a DAA credential, which includes Q, and of a Schnorr signature, valid under Y'. Recall that, $cred = (Y, (A, e))$, where $A = (g_0 QY)^{1/(sk_I+e)}$, where sk_I the DAA issuer's secret key, with corresponding public key $PK_I = g_2^{sk_I}$. M will blind the signature (A, e) by choosing $a_2 \xleftarrow{\$} \mathbb{Z}_p$ setting $\bar{A} = A^{a_1 a_2}$, $\bar{Y} = Y^{a_1}$ (note that a_1 is the random scalar that was used to blind Q, i.e., to create the commitment $\bar{Q} = Q^{a_1}$). M will also set $D_2 = (g_0 h^{sk_{DAA}})^{a_1}$ and $B_2 = D^{a_2} \bar{Y}^{a_2} \bar{Q}^{a_2} \bar{A}^{-e}$ $(= \bar{A}^{sk_I})$. Finally, M will calculate a signature proof of knowledge (SPK) π_3, over a message m as follows (recall that will set $a_1' = 1/a_1$),

$$\begin{aligned} \pi_3 = SPK\{(a_1', a_2, e, sk_{DAA}) : \\ B_2 = (D_2 \cdot \bar{Y} \cdot \bar{Q})^{a_2} \bar{A}^{-e} \wedge g_0 = D_2^{a_1'} \cdot h^{-sk_{DAA}}\}(m) \end{aligned} \tag{5.8}$$

Let $proof_3 = (\bar{A}, \bar{Y}, \bar{Q}, D_2, B_2, \pi_3)$. Note that in the above construction, there is no relationship between Y (the threshold signature group public key set by I_{DAA}, corresponding to the original threshold value t) and Y' (the threshold signature group public key corresponding to the updated threshold t'). This means that Del could create a (sk', Y') key pair independent of the threshold signature protocol and (coordinating with M), bypass the need to execute the threshold signature protocol. To avoid this issue, M will prove that Y' is correctly constructed by Del. Recall that, $Y' = Y g^{s'}$ for some randomly sampled by Del s', which Del will send to M (together with it's signature over Y'). M will check that indeed Y' is correctly calculated and prove knowledge of a value s' so that the above equation holds. Note that $\bar{Y} = Y^{a_1}$ and that $\bar{Y}' = Y'^{r_2}$.

To that end, M will generate the following zero-knowledge proof-of-knowledge,

$$\pi_4 = SPK\{(a_1', r_2', s') : 1_{G_1} = \bar{Y}'^{r_2'} \cdot \bar{Y}^{-a_1'} g^{-s'}\}(m) \tag{5.9}$$

We can construct the final DAA Signature as

$$\begin{aligned} sig_{DAA} = (proof_{1,2}, proof_3, \pi_4) = \\ (\bar{PK}_D, \hat{\bar{PK}}_D, B_1, D_1, B_2, D_2, \bar{A}_D, \bar{A}, \bar{Y}, \bar{Y}', K, \bar{Q}, R, \bar{R}, \pi_{1,2}, \pi_3, \pi_4) \end{aligned} \tag{5.10}$$

At a high level, to validate the DAA signature the verifier needs to verify each included proof, i.e., $proof_1, proof_2$ and π_4. Concretely, the steps to verify a DAA signature supporting threshold update are the following,

1. $\hat{h}(\bar{A}_D, \bar{PK}_D) = \hat{h}(B_1, g_2)$
2. Check that $\bar{PK}_D \neq 1_{G_2}$, that $\hat{\bar{PK}}_D \neq 1_{G_2}$ and that $\hat{h}(\bar{Q}, g_2) = \hat{h}(q, \hat{PK}_D)$.
3. $\hat{h}(\bar{A}, PK_I) = \hat{h}(B_2, g_2)$
4. Calculate $c = H(R, PK_I, m)$ and verify the SPKs $\pi_{1,2}, \pi_3$ and π_4

Analyses. We will give here an outline of the fact that the above protocol is proof of knowledge. The construction of our sig_{DAA} is equivalent To the following; set,

$$sig_{DAA} = (\bar{PK}_D, \hat{\bar{PK}}_D, B_1, D_1, B_2, D_2, \bar{A}_D, \bar{A}, \bar{Y}, \bar{Y}', K, \bar{Q}, R, \bar{R}, \pi)$$

,

where π the following zero-knowledge proof-of-knowledge;

$$\begin{gathered}
\pi = SPK\{ \\
(r_4, e, r'_4, r'_1, s', a'_1, a_2, \gamma, sk_{DAA}, r'_2, s, \tilde{a}) : \\
B_1 = D_1 \bar{Y}' K h^{-r_4} \wedge 1_{G_1} = K^{r_1} \bar{A}^e h^{-r'_4} \wedge g_0 = D_1^{r'} h^{-s'} \\
\wedge R = \bar{R}^{r'_2} g^{-\tilde{a}} \wedge \bar{R} \bar{Y}'^c = g^\gamma \\
\wedge B_2 = (D_2 \cdot \bar{Y} \cdot \bar{Q})^{a_2} \bar{A}^{-e} \wedge g_0 = D_2^{a'_1} \cdot h^{-sk_{DAA}} \\
\wedge \bar{PK}_D = \hat{\bar{PK}}_D^{a'_1} \wedge \hat{\bar{PK}}_D = \hat{PK}_D^{r'_1} \\
\wedge 1_{G_1} = \bar{Y}^{r'_2} \bar{Y}^{-a'_1} g^{-s}\}
\end{gathered} \tag{5.11}$$

Run the extractor of π to get the values $(r_4, e, r'_4, r'_1, s', a'_1, a_2, \gamma, sk_{DAA}, r'_2, s, \tilde{a})$ so that all equations of π hold. From the first 3 equations of π, using the analyses of Section $5.7.2$ we can get signature (A_D, s_D, e_D) valid under the public key PK_D (known by M) committed in $\bar{PK}_D$. Given that $\bar{PK}_D \neq 1_{G_2}$ and that $\hat{\bar{PK}}_D \neq 1_{G_2}$ we get that $a'_1, r'_1 \neq 0$. Using the aforementioned analyses we get that $PK_D = \bar{PK}_D^{1/r'_1} = g_2^{\delta/r'_1}$, where δ the discrete log of $\bar{PK}_D$ in the basis of g_2. Setting $sk_D = \delta/r'_1$ and $Q = \bar{Q}^{a'_1}$, we know from the analyses of the proof $\pi_{1,2}$ of this Section, that $Q = q^{sk_D}$, where sk_D the secret key that generated the signature (A_D, s_D, e_D). Using the extractor from the analyses of $\pi_{1,2}$ we know that the signature (A_D, s_D, e_D) is over a point Y', committed to $\bar{Y}'$, so that $Y' = \bar{Y}^{r'_2}$. Additionally, that extractor returns a Schnorr signature $(R, \sigma = \gamma * r'_2 - \tilde{a})$, valid under the public key Y'. From the sixth and seventh equation of π, as well as the fact that $\hat{h}(\bar{A}, PK_I) = \hat{h}(B_2, g_2)$, we get that $\bar{A}^{sk_I+e} = D_2 \bar{Y} \bar{Q} => (\bar{A}^{a'_1})^{sk_I+e} = g_0 h^{sk_{DAA}} \bar{Q}^{a'_1} \bar{Y}^{a'_1}$. From that we can conclude that M, knows a signature (A, e), issued by I_{DAA} over their DAA secret key sk_{DAA}, Del's secret key sk_D and the threshold signature group public key $Y = \bar{Y}^{a'_1}$. From the last equation of π we can get that $Y' = Y g^s$, meaning that the calculation of the updated public key is done correctly.

Chapter 6

Securing the Edge Components of *CONNECT*

While we outlined the secure container management in Chapter 3, we now provide additional design details for the required services. This includes secure migration and update for TEE-enabled workloads as well as some required infrastructure services that are required for this design.

6.1 High-level Architecture for Secure Migration of Intel SGX and Gramine

One goal of the *CONNECT* project is to enable secure upgrade and migration of security-critical workloads:

Upgrade: Software is constantly evolving to add features and reduce potential bugs. To allow software innovation and improvement, it is essential that hardware-protected services can be updated securely.

Migration: To increase availability of services, it is important that services can be migrated. This can happen between different ECUs in the vehicle as well as from the vehicle to the edge.

We now outline how we plan to achieve these objectives. The heart of our design is a *TMC* as a Service that we will detail in section 6.2.1. Before detailing design elements in the subsequent chapters, we will first outline the high-level architecture described in Figure 6.1. In the sequel, we focus on migration. From a high-level perspective, secure upgrade is a migration within one machine from a TEE with one software version to another TEE with a later software version. The depicted architecture includes a source and a target machine. The source machine initially hosts a TEE with the Gramine LibraryOS and a given application (both may include some state). The goal is to create a replica of this TEE on a target machine while ensuring that integrity and confidentiality is protected while the TEE is migrated from the source to the target machine.

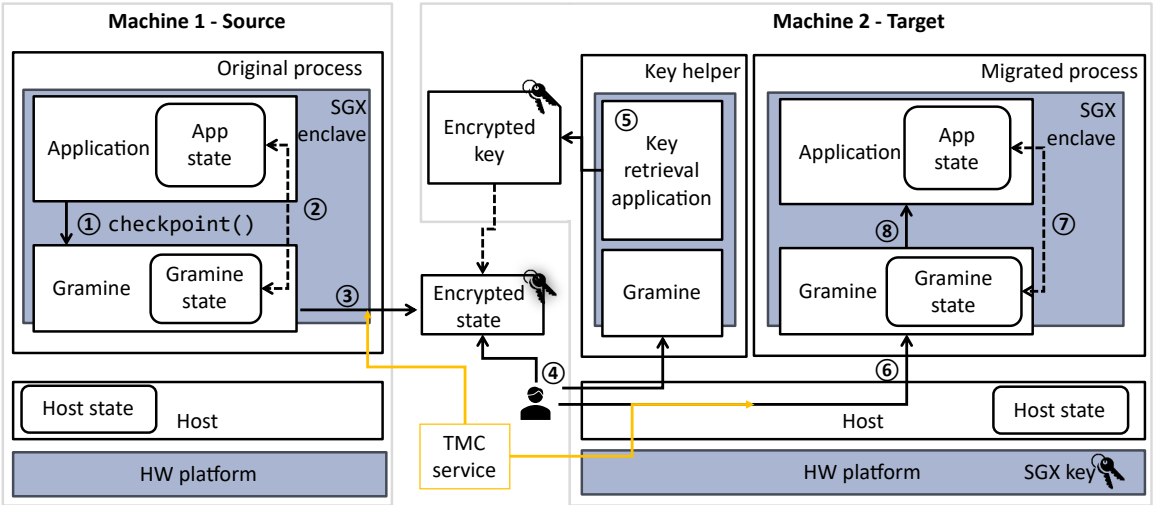

Figure 6.1: High-level Architecture for Migrating intel SGX Enclaves and the Gramine LibraryOS.

6.1.1 Security Objectives

Both procedures, upgrade and migration, share a common set of security objectives that we will later explore further in Section 6.3:

Secure Cloning a TEE from Source to Target: The main objective is to migrate/upgrade a service, i.e. given a service within a TEE that runs a certain software version with a given state on a given host, the same service with the same state shall be established either using a new software version (upgrade) or on a new host (migrate). Services may maintain a state that needs to be preserved during upgrade/migration. An example is the database of valid keys in a cryptographic key-store. Preservation means that a valid state is transitioned from one software version to the subsequent one (when upgrading) or migrating the state from one node to another (when migrating).

Integrity and Confidentiality while Cloning: The TEE (e.g. Intel SGX) ensures that the state is hardware-protected while in use. To maintain integrity and confidentiality, it is essential that the state is directly migrated from one TEE to another while protecting integrity and confidentiality. In practice, this means that the state must be encrypted and integrity protected before exporting it and the integrity must be validated while decrypting in the target TEE.

Permanent Disabling of the Source TEE: If a service is to be migrated or upgraded, a common security requirement is to permanently disable the source service.[1] E.g. if a key management service includes revocation, then it is important that there is only one single ground truth of what keys are still valid. In this scenario, one needs to prevent that an attacker can force a ”switch back” to the original source service that may still lists some keys as valid while they have been revoked on the new cloned service. We address this requirement by devising a method to guarantee that the original service is permanently and irrevocably disabled in Section 6.3.3.

[1] Disabling the source is alwas required for upgrade (since the outdated TEE should not continue to operate). However, for increasing capacity, there are cases where cloning a TEE is desirable. In this scenario the original TEE as well as a series of clones all continue to exist.

Rollback Protection: An important attack (in particular against secure upgrade) is a so-called rollback. This is similar to switching back to the source TEE. This means that an adversary can use migration or upgrade to actually downgrade the software version to an earlier release that may contain known but unfixed vulnerabilities. For security-critical services, rollback protection is commonly required.

Singleton guarantees: For some services, there needs to be a "single source of truth". One example is key revocation - once a key has been revoked, there must not be claims that it is still valid. One attack against such services is to create unauthorised siblings. E.g. backup a revocation services, let the revocation happen, and relay queries to the unauthorised copy that still claims that the key is valid. To mitigate such attacks, it important to enable "singleton services" where the infrastructure guarantees that at most one service is authorised at any point in time.

6.1.2 High-level Flow for TEE Migration/Update

We now outline the high-level flow for migration/upgrade. Note that this flow only implements confidentiality and integrity, it does not yet implement the other desired security guarantees like rollback protection, guaranteed disablement of the source service, and prevention of multiple instances. We will elaborate how to add these security guarantees in Section 6.3. The high-level flow of migration in our architecture works as follows (the numbers correspond to the circled numbers in Figure 6.1):

1. To start the migration process, the application developer triggers the `checkpoint()` system call in the original process and Gramine receives this system call and initiates the export of its state.

2. This application and Gramine collect and dump their state in into a single blob that is ready for export. Note that depending on the programming language used, this may require explicit support of the used applications.

3. The enclave now encrypts the blob with a key available to the target enclave. One option is to use the `MRsigner` key of SGX. In this case, any enclave signed by the same signer can decrypt the blob. Another alternative is to use a specific *KBS* to ensure that only the desired target machine has the key to decrypt. In this scenario, an ephemeral symmetric key is provided by the *KBS* detailed in Section 6.2.2. This key is shared between source and target, the state is encrypted and authenticated. This can be done by generating a message authentication code (MAC) and encrypting or by using an authenticating encryption mode/scheme. Gramine then dumps the checkpoint blob into a file specified as an argument to checkpoint(). The *KBS* (not shown here) should provision the key to the original Gramine enclave and later to the migrated Gramine enclave, and then destroy this ephemeral key. A public transaction handle can be provided by the *KBS* to identify a given key that is to be used in a given transaction.

4. The encrypted state is made available on the target Machine 2.

5. To obtain the key to decrypt, there are again two options. If the `MRsigner` key was used to encrypt, then this key is available to any TEE signed by the same signer. If the *KBS* was used, the system must first start the "key helper" Gramine SGX enclave (4) that manages

the import.The purpose of this helper enclave is to establish a connection to the *KBS*, obtain the ephemeral key, encrypt this key using the SGX-platform-specific key and write it as a separate file (5) (Notice that without this double-encryption, the ephemeral key would be stored on the host of Machine 2 in plaintext and could be stolen by a malicious host).

6. The developer starts Gramine in a special “restore” mode, specifying as two command-line arguments: the file that contains the encrypted state and the file that contains the ephemeral key.

7. Upon SGX enclave initialisation, Gramine first decrypts the ephemeral key file using the SGX-platform-specific key, and then decrypts the encrypted-state file. Finally, Gramine restores the decrypted checkpoint.

8. All Gramine and application data is ready for use, and Gramine passes control to the application, to run from the location right after checkpoint() (8).

Note that the service outlined preserves integrity and confidentiality but does not guarantee migration (i.e. the original service is not reliably disabled and may continue to run), rollback-protection, or guarantee that only one clone is created. We will investigate corresponding extensions in Section 6.3.

6.2 Required Infrastructure Services

Our goal is to provide robust security guarantees that are aligned with the Intel SGX trust model: Using Intel SGX, only hardware-protected software that is running inside a verified Intel SGX enclave is trusted by default. Neither the operating system nor any other software on the machine is considered trusted. By default, an enclave can be dumped (e.g. to allow hibernation). In addition, the enclave can be restored multiple times. As a consequence, Intel SGX does not provide hardware guarantees sufficient to prevent unauthorised clones or rollback after an upgrade.

We now outline two infrastructure services that are required for the security of TEE migration. The Trusted Monotonic Counter service guarantees that counters can be authenticated, increased but not decreased. We will argue in Section 6.3 that these services are sufficient to implement the desired security guarantees. Another alternative can be Blockchain-based services.

6.2.1 Trusted Monotonic Counters as a Service (TMCaaS)

We now describe the *TMC* as a Service design. The goal of this service is to provide a “global single source of truth” that can be used by an enclave. For upgrade on a single machine, this can be implemented locally [41]. For a distributed system with multiple machines - such as vehicles and the edge - a dedicated service that can be centralized or distributed will be required to provide this ”ground truth”. We now outline our high-level design. An alternative proposal has been published in [38].

6.2.1.1 Introduction

Trusted monotonic counter services are a security primitive that essentially provides ways to ensure that a counter has a deterministic behaviour (i.e., increase) and cannot be reset or rolled

back. In the context of confidential computing, this is crucial for preventing replay attacks and ensuring the integrity and uniqueness of operations in security-sensitive applications. A monotonic counter is designed to be tamper-resistant; it should be infeasible for an adversary to manipulate the counter without detection. Monotonic counters can be used in a variety of scenarios, such as ensuring the integrity of secure storage, preventing replay attacks by associating a counter with a data object, or restricting the usage of a feature (e.g., application migration) to a certain number of times. To increase their robustness and fault-tolerance properties, trusted monotonic counter services are often designed as distributed services that leverage secure hardware modules (e.g., Intel SGX). In this way, the counter operations are isolated inside a secure environment (e.g., secure enclave) that cannot be tampered and their state is securely replicated across different nodes inside a trusted network, where the communication is performed via secure, encrypted channels (e.g., TLS). Overall, such counters are essential for maintaining the integrity and trustworthiness of a system, especially in environments where security is important.

6.2.1.2 Motivation

The reasoning behind our project's need for a Trusted Monotonic Counter Service is twofold; first, we have to ensure that an applications running inside an Intel SGX enclave is migrated **only once** and, second, we must guarantee a **secure upgrade process and prevent rollback attacks**.

Precisely, to ensure that a service is migrated only once, a robust migration process must be established. This process should include a unique identifier (i.e., a unique counter value) associated with each migration operation, which is recorded in a secure database. Thus, the system can check against this database before initiating any migration task to guarantee that the service has not been previously migrated. This approach prevents duplicate migrations and ensures that each service is migrated **exactly once**.

Additionally, to guarantee a secure upgrade process and prevent rollback attacks, the system must ensure that once an upgrade is applied, it cannot be reverted to a previous, potentially less secure version. This can be achieved by using trusted monotonic counters that increment with each update. These counters provide a way to verify that the system state is the expected has **not** been rolled back. During the upgrade process, the system should verify the integrity and authenticity of the update. After a successful upgrade, the monotonic counter can be incremented, and the new version number should be securely stored. If a rollback attempt is detected, the system can compare the current counter value with the stored version number to detect or prevent the rollback.

6.2.1.3 Background - Microsoft Confidential Consortium Framework (CCF)

To facilitate the development of our Trusted Monotonic Counter Service, we choose to use Microsoft's Confidential Consortium Framework [33]. Microsoft's Confidential Consortium Framework (CCF) is an open-source framework designed to enable the creation of applications that require a high degree of trust and data confidentiality. CCF is suited for scenarios that involve multi-party governance, integrity protection, and programmable confidentiality, even in situations where the datacenter or host operator may be compromised. CCF leverages Trusted Execution Environments (TEEs). TEEs ensure that the code and data loaded inside them are protected with respect to confidentiality and integrity. CCF uses TEEs to create a decentralised trust model similar but with the added benefit of maintaining data confidentiality through secure, centralized

computation. The framework provides a simple programming model based on a key-value store. It is designed as a distributed system and uses a consensus protocol that provides Crash Fault Tolerance (CFT) [34] and is based on Raft [40]. This allows for the creation of flexible policies that enforce access control and other governance features [35]. CCF is designed to be efficient, extensible and easy-to-use. It is particularly useful for building enterprise-grade applications that require a common source of truth and decentralized trust, such as financial services, supply chain management, and more.

6.2.1.4 Trusted Monotonic Counter Design

We design our Trusted Monotonic Counter Service on top of Microsoft's Confidential Consortium Framework (CCF). The CCF network is composed by a set of nodes, called *member nodes*. To set up a CCF network, we must initialise it with at least one active member (*initiator*). After this point, more *members* can join the network. This is typically done through proposals, which are formal requests for changes that are voted on by the consortium members. Each CCF node is identified by a public-key certificate, known as the Node Identity Certificate, which is endorsed by an attestation report. This certificate is used to authenticate the node when it joins the network. After the finalization of the *member joining process*, the CCF network can become public; a process that is also performed through a separate proposal.

Once the CCF network becomes public, clients can access the network by establishing TLS connections. The service certificate, which is shared by all nodes trusted to join the network, is used as the root of trust for server authentication. In our Trusted Monotonic Counter Service, we enhance this process so that the CCF network attests the clients (mutual attestation) before adding their credentials in the *trusted* users list. Beyond this point, the clients can access the endpoints of the service and retrieve or check their desired counter value or instruct the service to perform an atomic increment.

Importantly, the service logic of each node is running inside a TEE, which is Intel SGX in our case. Our service supports multiple counters. Each counter has its own ID and is securely stored a key-value pair, where the key is the counter ID and the value is the actual counter value. The state of the counters is replicated across all *member* nodes to improve fault-tolerance. Currently, in our design, we have not integrate the failure-recovery process as we consider a long-running service with the majority of nodes available. However, CCF provides such recovery mechanisms [36] through its ledger.

For most of the above operations, the CCF *member* nodes have to vote for the proposals made in order to perform any action. For simplicity, we currently require the majority of nodes to vote for a proposal before proceeding with the appropriate operation (e.g., add a client to the *trusted* users list).

6.2.1.5 Trusted Monotonic Counter Implementation Details

The Trusted Monotonic Counter Service is built using the Microsoft CCF framework. It leverages Intel SGX as its backend Trusted Execution Environment.

Each member node has its own self-signed public-key certificate. The initiator node bootstraps the network and waits for other nodes to join. It exposes endpoints where the nodes can send their information (e.g., public key, certificate etc.) and after being attested by the initiator that they run the correct and expected version of the *TMC* logic, they can be admitted in the network. This

operation is seamlessly provided by CCF using the OpenEnclave API. After the member nodes have joined the network, the initiator instructs the service to go public through a specific internal endpoint. Once this is successfully done, the clients can reach the *TMC* endpoints.

We currently expose a public endpoint for clients to send their information. After receiving this information, the CCF member nodes attest the client(s). In our case, the clients are Gramine-SGX applications. After a successful attestation, the client can be added in the *trusted* users list and then access the private endpoints that allow for *TMC* manipulation.

The proposals are constructed in a json format. Similarly structured are the votes, as instructed by the CCF framework.

As stated above, our current logic for the constitution (governance) is not sophisticated. We require the majority of the *member* nodes to vote for a proposal or transaction before it is applied.

Our Trusted Monotonic Counter Service exposes the following interfaces in its respective public endpoints:

1. $TMC\_init()) \rightarrow id$: Creates a new counter and returns its id.

2. $TMC\_get(id) \rightarrow cnt\_val$: Retrieves the current counter value.

3. $TMC\_inc(id) \rightarrow cnt\_val$: Returns the current counter value and triggers an atomic (transactional) increment.

4. $TMC\_check\_and\_inc(id, count) \rightarrow cnt\_val/false$: If cnt_val matches the current counter value, the counter value is returned, and an atomic (transactional) counter increment is triggered, otherwise an error is returned and no operation is performed.

5. $TMC\_destroy(id) \rightarrow true/false$: Destroys the counter with id and returns $true$ or returns $false$ if a counter with this id does not exist.

Using these endpoints and functionalities, we can provide solutions about the **uniqueness** of a Gramine-SGX app migration and **roll-back protection** provided that our *TMC* service is up and running.

6.2.2 Key Broker Service (KBS)

The goal of the key broker service is to “pair” a source enclave and key retrieval application (as the target of the migration) and to provision an ephemeral symmetric key to both enclaves. This will require the following steps to be implemented:

1. The originating enclave establishes a secure channel to the *KBS* and register its trust requirements for the target of the migration. This can include integrity requirements on the target key retrieval application or on the target enclave (e.g. version information). It may also specify the crypto to be used for encrypting/authenticating the state. In return, it will obtain a suitable ephemeral symmetric key and a non-confidential key/transaction handle.
2. We assume that the encrypted/authenticated state and the transaction handle is transferred to the target machine and input to the key retrieval service.
3. The target key retrieval application establishes a secure connection to the *KBS* and queries for the key while providing the transaction handle.

4. The *KBS* verifies that the target key retrieval application satisfies the trust requirements of the source enclave. If not, it declines the request.

5. The *KBS* provides the ephemeral key.

Overall, this ensures that (a) source and target share a common ephemeral key and (b) only authorised targets can obtain access to the key.

6.3 Implementing TEE Security Guarantees

In general, we can employ the trusted monotonic counter service to ensure that an action or a service is started or performed only once (e.g., during a migration). In a nutshell, for our project, the term singleton highlights that **at most one** Gramine instance is launched. The singleton concept is used both for upgrade (Section 6.3.2) and migration (Section 6.3.3), making sure the ”old” Gramine instance is no longer active. Either the sealing of Intel SGX or else the Key Broker Service can be used to constrain the target machine of a migration (Section 6.3.3).

6.3.1 Implementing Singleton Services using a Trusted Monotonic Counter (TMC) Service

Security Requirements: We assume that a Service Control Authority defines and controls a service. The goal of the protocol is to ensure that a service can only be started at most once. I.e. a new instance of the service can only be started from the state that the earlier copy has exported during exit.[2]

Protocol Outline and Extensions: The core idea of the singleton guarantee is to define a counter that allows a service to (a) verify whether this is the first run (globally) or else (b) whether the state of the counter corresponds to the counter value stored in its saved state (i.e. while the service was asleep no other service was started). The protocol depicted in Figure 6.2 has three phases:

In the first phase ”*Register new service*”, the authority controlling the global set of authorized services (usually the OEM) defines a globally unique identifier for the service and other information that uniquely identifies a given service. Given this ID, we assume that a specific service can be identified (i.e. it may also include a cryptographic hash). Both are included in a unique service definition $Service - id$. It then creates a new counter with a unique $\mathrm{counter} - \mathrm{id}$ and signs a service definition statement $srv = sign(Service - id, counter - id, ”0”)$ that associates the service with a specific counter that is initially ”0”.

In the second phase, ”*Service start*”, the service is to be started given a service definition and, optionally, a saved state from a prior execution. The service then retrieves the expected counter value from the sealed state (or ”0” for the initial start without state). It then invokes ”check and increase” for the counter. If this fails, the service exits. Else, the service is launched. We assume that all authorized software includes this check - starting from the first release.

[2]Note that this focuses on stateful services; we did not investigate stateless services.

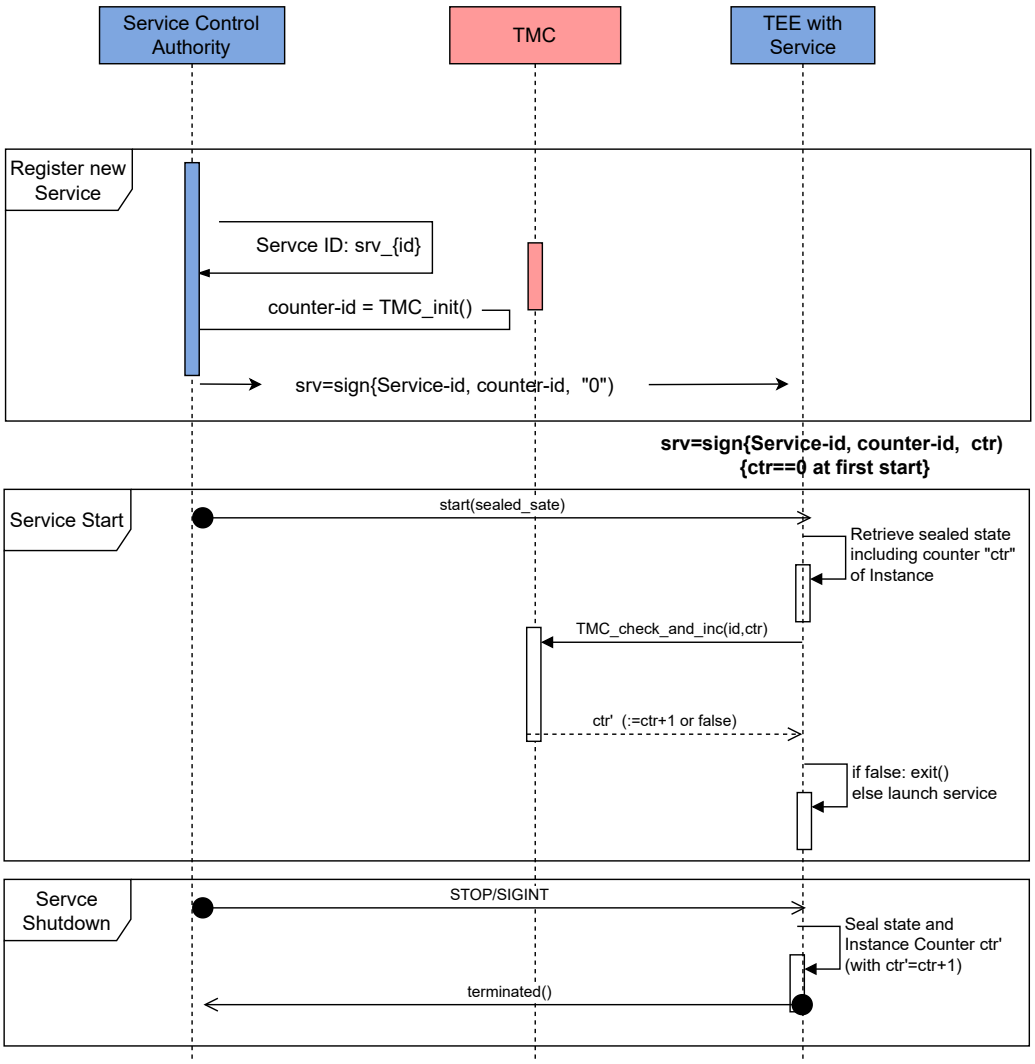

Figure 6.2: Using *TMC*aaS to prevent multiple copies of a Singleton Service.

The third phase is the controlled *Service Shutdown*. When the service manager signals that a service shall shutdown (e.g. by sending `SIGINT`), its state is sealed including the currently valid instance counter, which is $ctr + 1$ and corresponds to the state of the global counter - thus enabling a restart.

Security Assumptions and Security Argument: We assume that the protocol is correctly implemented and that all the required keys are correctly distributed. We also assume that all authorised services (with a signed manifest) include and execute the specified singleton check before starting the core service. For a given Service ID $Service - ID$, the TEE only launches software where (a) the manifest was signed by the authority and (b) that includes the specified checks. We now consider two cases: For the first run (without a prior saved state), the service only starts if the counter is still 0, else it exits. We can show the singleton property by induction: Before the service starts the monotonic counter is increased and the internal state is updated. Any parallel start will fail since any saved state still has the outdated counter included that will not pass the equality check with the monotonic counter. The only way to obtain a new state on disk that contains the increased counter is to then save the state of the single service that was started first and that increased the counter. Note that the software is required to terminate after saving

its state to avoid duplicates. This saved image can then be used to later re-start the singleton service.

Note that while the service is running, the central maintained counter is higher than any saved state. I.e. whenever someone attempts to launch a service, the counter (in the state) should be lower than the central counter and thus the service should exit.

6.3.2 Guaranteeing Secure Upgrade and Preventing Rollback Attacks

Security Requirements: We assume that a Software Release Authority (SRA) such as an OEM authorises a given software version to be executed within a TEE. The SRA may also release subsequent updates. The goal of this protocol is to ensure that once a new software update has been released, then it is impossible to successfully launch a TEE and execute an outdated version of a given service/software.

Protocol Outline and Extensions: The core idea of anti-rollback is as follows: Whenever a new software version is released, a well-known counter corresponding to this software package is increased. This counter is then used by the TEE to check that its software is not outdated (by checking that the counter value stored in its software is equal to the counter value published by the Trusted Monotonic Counter Service). The protocol depicted in Figure 6.3 has three phases:

In the first phase "*Initial Software Release*", the authority controlling software releases (usually to OEM) defines a globally unique identifier for the software and a version to uniquely identify a given software release. Given this ID, we assume that a specific package can be verified (i.e. it may also include a cryptographic hash). Both are included in a unique software identifier $SW-id$. We assume that subsequent releases are strictly ordered. It then creates a new counter with a unique $\mathrm{counter-id}$ and signs a package release statement $pkg = sign(SW-id, counter-id, "0")$ that associates the software with a specific counter that is initially "0".

In the second phase (optional), the Software Release Authority (SRA) authorises a new software release by (a) increasing the associated counter to a new value ctr' and then signing an updated package release statement $pkg' = sign(SW - id', counter - id, ctr')$ for the updated counter value.

The third phase is the actual *Launching the Service* within a TEE: We assume that all authorized software includes this check - starting from the first release. When the TEE starts executing, it will first check that the software is one of the binaries authorised to run at all (by verifying software integrity via the signed manifest). The first steps that all authorised software releases will do is to retrieve the signed package release statement pkg for itself and verifies its signature. This statement contains a $counter - id$. The software then retrieves the latest value of this counter. If the value is higher than the counter in the statement, then the current software is outdated and the TEE exits. If the counter in the statement is equal to the up-to-date counter value in the package info, then it continues. This process ensures that the latest software version is running and prevents rollback attack as it checks the tamper-proof counter value of the trusted monotonic counter service.

Security Assumptions and Security Argument: We assume that the protocol is correctly implemented and that all the key-pairs are correctly distributed. We also assume that all authorized software packages (with a signed manifest) include and execute the specified up-to-date check

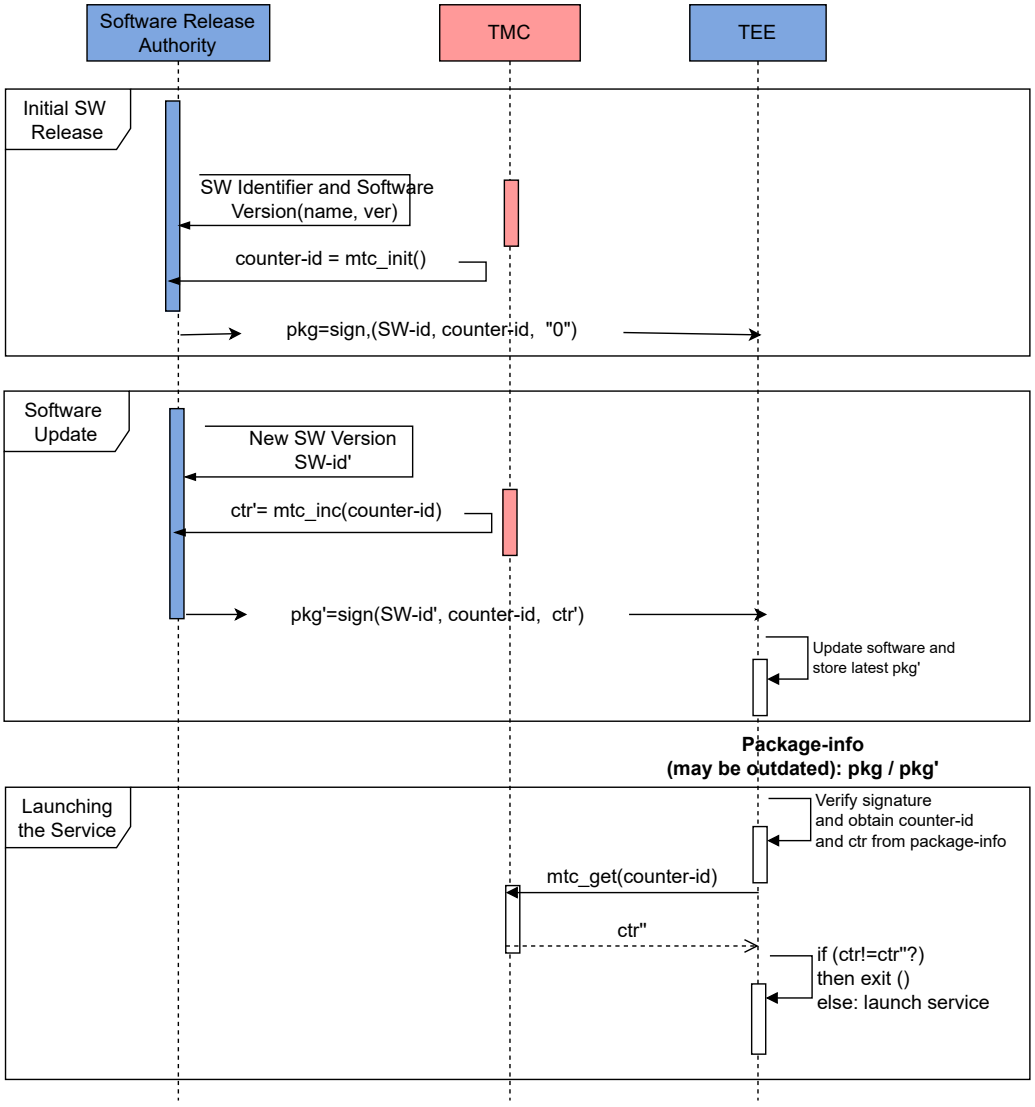

Figure 6.3: Using *TMC*aaS to Prevent Rollback Attacks.

before starting the core service. For a given Software ID $SW - ID$, the TEE only launches software where (a) the manifest was signed by the authority and (b) that includes the specified checks. The authority ensures that each new release contains a signed counter never used before. If we now assume a TEE is successfully launched (i.e. the software manifest was signed and thus includes the specified checks), then the checks ensure that the latest counter is retrieved. If we now assume that a later release exists, then the TEE will exit without continuing the execute the desired service.

6.3.3 Constraining the Target TEE of the Migration of a Service

Migration of a Service refers to the process of moving a running service from one machine (source) to another (target). As described in Section 6.2.2, we cannot rely only on the special SGX sealing keys (i.e., `sgx_mrsigner` and `sgx_mrenclave`) to guarantee the desired security properties of migration. `sgx_mrenclave` can guarantee that the service can only be unsealed and read by the same enclave on the same machine and `sgx_mrenclave` can guarantee that the state can only be unsealed by a enclave where the manifest is is signed by the same signer. Neither

can guarantee that the state can only be read by a specific enclave on a specific machine.

Therefore, we decided to introduce a dedicated Key Manager to enable the decryption of the encrypted state only by an authorised enclave. However, apart from the secure key management for the migration, we have to make sure that the migrated enclave is launched only once and that the latest enclave version is launched. One goal is to ensure that the *original service is permanently disabled* during migration. We aim to achieve this by (a) using the singleton service to ensure that the original service can no longer be started once the new service is up and running and (b) including the target machine into the state before dumping it and then checking that the restore-and-restart happened on the correct target machine.

We outline the full integration of all services into a migration in Figure 6.1 and provide a step-by-step description in Section 6.1.2.

Chapter 7

Conclusion and Outlook

Deliverable D4.2 focused in the refinement and finalisation of the *CONNECT* Trusted Execution Architecture as a core building block for supporting the first-of-its kind Trust Assessment Framework (TAF), capturing the trust model and complex trust relationships that need to be continuously assessed towards the materialisation of a trusted CCAM continuum. Towards this direction, this deliverable extended the overarching trusted computing base to also capture the security requirements of all information and functional assets comprising the MEC layer as an additional operational plane for supporting the provision of safety-critical CCAM services. This transformation of the stand-alone vehicle domain into a safe and security solution distributed from vehicles to MEC and Cloud facilities, is a critical enabler for Day 2 automotive operations (with a high degree of efficiency and automation [12]) providing vehicles seamless access to computational capabilities by deploying services at both the far edge and edge layers of the CCAM continuum, thus, also facilitating the vision of task offloading. However, this updated operational model introduces new attack vectors, necessitating sufficient safeguards to assess and establish trust in both vehicles and the virtualised infrastructure where services are deployed.

Hence, a core trust pillar for this extension lies the integration of a *TEE* into the *MEC*, safeguarding all operations, including continuous and evidence-based trust assessment of any *CCAM* actor or data object. Towards this direction, D4.2 provides the final and refined description of *CONNECT*'s overarching Trusted Computing Base considering also the provision of the necessary mechanisms for ensuring the **secure lifecycle management of any CCAM (and beyond) service deployed on the MEC.** This does not only include the runtime attestation of the service itself (enacting upon the previously mentioned attestation procedures) but also captures the integrity and resilience of all the MEC infrastructural elements as well as the SW components running on them, and especially those facilitating the network orchestration plane. Attestation of a virtualized infrastructure element (i.e., measurement and verification) leads to the ability to establish information security assurance. **This, in turn, constitutes another core trust source based on which the vehicle can assess the level of trust it can put on the MEC to protect its information and functional assets.** All these capabilities set the scene for the detailed experimentation and evaluation activities of all these trust extensions in the context of the envisioned use cases on Intersection Movement Assistance, Collaborative Cruise Control and Slow-Moving Traffic Detection (WP6).

The first version of these advanced trust extensions is also presented in this deliverable comprising *CONNECT*'s crypto agility layer focusing mainly on safeguarding the operation of the vehicle. More specifically, D4.2 provides the detailed description of a new set of trusted computing capabilities towards the runtime integrity verification of a component (such as an *ECU*)

through abstractions, called *policy-restricted attestation keys*, that allow for the verification of the component's integrity correctness in a zero-knowledge manner; i.e., without disclosing any details on the actual configuration or operational profile of the target device. This functionality, together with those libraries offered for secure software upgrade and/or state migration (as reaction policies to the change of a node's trust level), comprise *CONNECT*'s *TCB* capabilities for supporting the operations of the Trust Assessment Framework: *All attestation attributes constitute one of the trustworthiness sources based on which the TAF (as the core WP3 artefact) calculates the Actual Trust Level (ATL) of a (SW and/or HW) element which leads to the trust state characterisation of the target node*.

These attestation attributes are then shared (as trustworthiness evidence/claims) with external entities in a secure and privacy-preserving manner so as to avoid the linkability and fingerprinting of the vehicle itself - which in turn might lead to further implementation disclosure attacks. This high degree of privacy is achieved through *CONNECT*'s new Threshold Anonymous Direct Anonymous Attestation (DAA) scheme that allows the construction of anonymised signatures as proof-of-conformance to those trust properties of interest; i.e., device integrity, communication integrity, safety, etc. Without, however, disclosing any of the details of the runtime configuration and behavioural profiles of any of the in-vehicle elements comprising the CCAM function that is assessed. To the best of our knowledge, this is the first attempt in the literature that try to combine the strict properties of platform authentication and integrity (offered by DAA) with the privacy guarantees of anonymous (FROST) signatures. This allows ***CONNECT* to ensure that sharing of attributes related to vehicle trustworthiness do not create any privacy threats by avoiding the disclosure of excessive or unnecessary information.**

7.1 Open Questions and Next Steps for Workpackage 4

After outlining the complete security architecture for *CONNECT*, our focus will be on design and implementation of key components to enable the evaluation of our architecture. The actual design, implementation, and its evaluation will finally be documented in Deliverable D4.3. A detailed road-map towards the consortium's upcoming design and implementation activities is presented in Table 7.1.

CONNECT Trust Extension	Implementation Status & Research Plan
ECU On-Boarding Protocol Implementation	Demos and detailed evaluation of the on-boarding protocols for ECUs (to be documented in the context of D6.1 [14]). Where information is available, we will use the data structures that are typically used in an actual vehicle. Initial demos will be single programs with the different entities running on different threads and passing protocol messages which will be displayed as the protocol proceeds.

Enhanced Configuration Integrity Verification (CIV)	The focus of the remaining activities are twofold: (i) Proceed with the finalisation of the implementation of the CIV scheme and its integration with the *CONNECT* TAF for providing one of the trust sources based on which data-centric and node-centric trust quantification can be performed, and (ii) Extension of the design to also capture the security assurances needed for attesting not only the far-edge (Vehicle) but also all edge (MEC) information and functional assets. For the former, particular emphasis will be given on the introduction of advanced tracing capabilities (exposed through *CONNECT*'s Hardened- and Softened-TDIs (Section 2.1)) allowing for the runtime monitoring and measurement of the configuration profile of a target device. For the latter, the mapping of our attestation procedures to a pre-defined scale of varying Levels of Assurance will be finalised for establishing the trustworthiness of a particular set of components (e.g. system or platform), according to the nature of the requested service, the threats being considered, and the applicable policies at all levels. This clause will also introduce new attestation primitives for asserting to the correct operational status of all MEC elements including the network plane (i.e., Virtualised Network Functions).
Verifiable Key Restriction Usage Policies	Perform a detailed security analysis on the provided properties especially as it pertains to assuming near-zero trust assumptions for the in-vehicle network. This essentially considers a Dolev-Yao adversarial model, which allows an adversary to monitor and modify all interactions between a host and the underlying trusted component. This analysis will also be coupled with an extension evaluation of the performance overhead that this additional safeguard puts on the overall system as part of *CONNECT*'s experimentation activities in D6.1 [14].
Trusted Computing Base	Design and implementation of the *CONNECT* trust extensions needed for supporting the security and trust requirements of the MEC and Cloud layers considered in the overall CCAM continuum. Essentially, this would extend our currently designed crypto and attestation primitives (focusing on the far-edge) to also attest to the correct integrity and operational assurance of any virtualized infrastructure onboarded for supporting the deployment and execution of safety-critical services. This will be facilitated through the Slow-Moving traffic Detection (SMTD) use case and the task offloading specific scenario where all MEC/Cloud functional assets will be integrated and evaluated in the context of D6.2. Documentation of this extended set of trust extensions (comprising *CONNECT*'s final version of its crypto agility layer) will be documented in D4.3.
Threshold Anonymous DAA Signature scheme	Formal analysis of the provided security and privacy properties - especially, against linkability and fingerprinting activities that may be performed by an attacker. This will also be coupled with a detailed evaluation analysis so as to check the feasibility of such advanced crypto primitives in the context of resource-constrained (in-vehicle) devices and their footprint in the safety profile of the vehicle (execution of timely-critical automotive operations). Finally, the new version of the scheme will be presented allowing for the upgrade of the threshold number of entities that need to provide correct signature shares without the need to change the primary DAA Key.
Selective disclosure / Threshold signatures	Integration and wrapping of the produced threshold (anonymous) signatures (constructed by the TCH) within the VC/VP abstractions so as to also enable properties including *selective disclosure*: VPs should hold collection of claims that the TCH can construct disclosing only those attestation attributes needed for verifying the trust level of an element without revelaing further information on the claims. This will be part of the second and final version of the overall *CONNECT*integrated framework to be documented and evaluated in the context of D6.2.
Application State Upgrade & Migration	Demos to validate and showcase secure migration and software update. These functionalities will also be evaluated: first, as standalone components (outside the context of the use cases) as part of the experimentation activities documented in D6.1; and, secondly, as part of the overall *CONNECT*service portfolio in the context of D6.2 (considering the detailed scenarios that were fleshed out in D2.1 [12] as part of the Collaborative Cruise Control use case).

Table 7.1: CONNECT Trust Extensions Implementation Road-Map.

Appendix A

Design Details of the *CONNECT* Trusted Execution Environment

A.1 The Intel SGX Trusted Execution Environment used in *CONNECT* in Detail

Compared to D4.1, we substantially refined the documentation of the technologies underlying the *CONNECT TEE*. In this appendix we document the refinements. In Section A.1 we first provide additional details on Intel SGX. In the subsequent Section A.2 we then provide additional details on the Gramine Library OS.

Intel SGX is a feature offered by many Intel CPUs. Its goal is to offer hardware protection for user-space processes. It is a specific type of Trusted Execution Environment (TEE). A high-level overview of the core functionalities of Intel SGX was provided in [10]. We now provide more details on underpinnings and mode of operation of Intel SGX towards supporting the secure execution of software binaries.

> **Connection to User Stories:** The SGX protection features outlined in the next subsections are required to provide a robust protection of a TEE while attestation specifically is required for Story-VII.

A.1.1 Intel SGX Example Scenario: Protecting a Business application in a Public Cloud

Consider a small business that sells pastries. This business decides to start selling their pastries online and hires a software developer to build a website. The software developer creates a website and decides to work with a cloud provider to host the website and handle the transactions. The developer sends the business data to the public cloud and works with the cloud provider to setup the environment. Finally, when everything is set up, the website starts running and serving content to users. When users purchase pastries from the website, all transactions are handled on the servers of the cloud provider.

After reading about a series of high-profile breaches in the used public cloud, the developer starts to question this approach. First of all, what is the actual code executing on the cloud – what if

a rootkit attack replaced the website code with a malicious version? Is there a chance that the confidential business data being leaked at the remote server? The developer would like to make sure that she is executing the correct website application on the right platform.

To gain trust in the application execution and protect transactions' confidentiality, the developer decides to work with a cloud provider which provides SGX-capable servers and she starts using an SGX-capable CPU. By using Intel SGX the developer can make sure that the website code cannot be modified. Additionally, through leveraging special SGX cloud services, the developer can also make sure that the original website application executes on the correct remote platform and that only the enclave the developer created has access to the confidential business data.

Key requirements:

- The business application (the developed website) must keep its business logic protected and confidential at all times. Protection of data in transit is guaranteed by SSL/TLS connections when the developer works on the remote public-cloud server via SSH (recall that SSH uses SSL/TLS encryption under the hood). Protection of data at rest is guaranteed by Full Disk Encryption (FDE) implemented by the cloud provider, whereas all code is automatically encrypted on the hard disk. Finally, protection of data in use is satisfied by Intel SGX, because the business logic runs inside the SGX enclave.
- The business application must keep customer data protected and confidential at all times. Protection of data in transit is guaranteed by SSL/TLS connections because the developed website requires HTTPS client connections (recall that HTTPS uses SSL/TLS encryption under the hood). Protection of data at rest is guaranteed by Full Disk Encryption (FDE) implemented by the cloud provider, whereas the website database, where the customer data is stored, is automatically encrypted on the hard disk. Finally, protection of data in use is satisfied by Intel SGX, because all processing of customer data runs inside the SGX enclave.
- The developer must verify the trustworthiness of the running website. In particular, the developer makes sure that the correct application (also the correct, up-to-date version of this application) runs on the remote server, and that the remote server itself is SGX-capable and trustworthy. These requirements are satisfied by Intel SGX remote attestation.

A.1.2 Intel SGX local attestation

Local attestation is used when two SGX enclaves reside on the same hardware platform and want to authenticate each other. Local attestation is straight-forward because both enclaves run on the same Intel CPU, thus they have access to the same shared secret burnt inside the CPU.

Below is a complete, though simplified, local attestation flow:

1. The target enclave (Enclave 1) wants to share a secret with the source enclave (Enclave 2), but before entrusting secrets, it must make sure that the source enclave is what it expects it to be.
2. The target enclave sends a plaintext message with its "target info" to the source enclave. This target info contains the measurement of the target enclave (so that the source enclave can verify it, if it wants to). Target info also contains additional information that the target

enclave wants to communicate to the source enclave (for example, which cipher suite it wants to use for secret communication).

3. Upon receiving this “target info” message, the source enclave derives a new key that will be used to prove it executes on a real Intel CPU. This key is derived from the CPU-specific root key. This key is also derived from the target info (which contains the measurement of the target enclave) and the measurement of the source enclave. Note that the newly derived key is based on attributes from both enclaves, which will help in establishing the shared secret later.

4. This derived key can be used to sign a “report” of a source enclave. The report is a blob that contains the new key, the measurement of the source enclave (so that the target enclave can verify its authenticity), and some other fields. This signed report is then sent to the target enclave. It contains all the information required by the target enclave to produce the same key and verify the source enclave.

5. Upon receiving the report message, the target enclave tries to derive the same key based on its own target info and the source enclave measurement from the report. It uses the same key derivation and same root key. Because the target enclave derives the key with the same function, same root key from the same Intel CPU, and the same parameters, this process must yield the exact same key that was derived at the source enclave.

6. The target enclave verifies the received report data with the newly derived key. If this verification fails, it implies that the keys derived at the source and target enclaves are different, which in turn implies that these enclaves do not execute on the same correct Intel CPU. The target enclave must also compare the source enclave’s measurement against the expected one. If these two verification steps succeed, the target enclave can be sure that the source enclave is valid and executes on the same genuine Intel CPU.

A.1.3 Intel SGX remote attestation

Remote attestation is used when two SGX enclaves residing on two *different* platforms want to authenticate each other. Remote attestation is also used when a remote non-enclave user wants to authenticate the remote SGX enclave.

Remote attestation is considerably more complex than local attestation, and involves special Intel CPU features and the support of an internet-accessible Intel Provisioning Certification service. Ultimately, an internet connection to this Intel service is required to fully gain trust in the remote SGX enclave (recall that Intel is the root of trust in SGX environments).

Since remote attestation is very complicated, in the following part we give a high-level intuition and overview how it works and omit the non-essential details.

The first important building block of SGX remote attestation is the set of *architectural enclaves*. Each Intel SGX-enabled platform has these architectural enclaves, namely a *Provisioning Certification Enclave* and a *Quoting Enclave*. These enclaves are developed by Intel, designed to be very secure, and they have hard-coded secrets shared with the “root of trust” (Intel remote services) that can be used for attestation.

These architectural enclaves execute on the same Intel CPU as the to-be-attested application enclave. Thus, the application enclave can perform local attestation to the architectural enclaves.

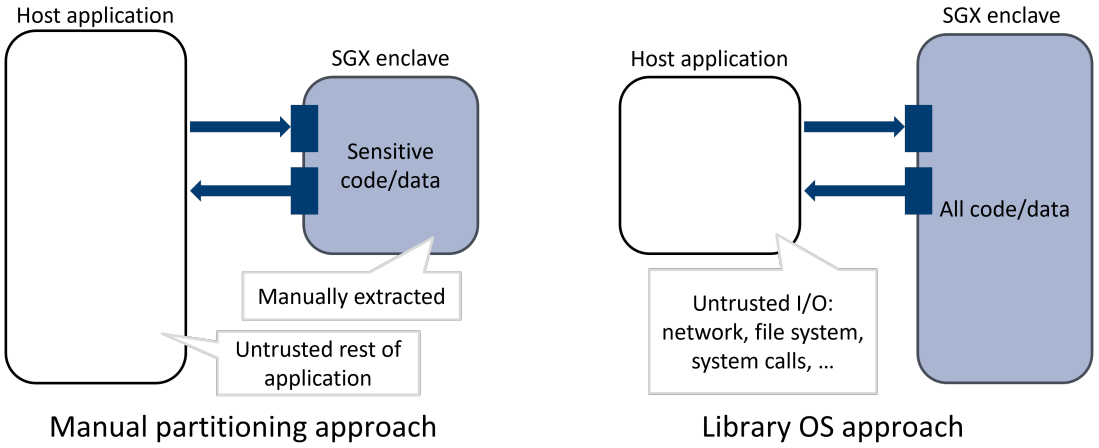

Figure A.1: Two approaches to developing SGX applications: manual partitioning and using a Library OS.

The architectural enclaves can then convert and transcribe the result of local attestation into a remotely verifiable proof by means of secrets they have.

The purpose of the Provisioning Certification Enclave is to provision a unique signing key to the second architectural enclave, the Quoting Enclave. The application enclave communicates only with the Quoting Enclave and not with the Provisioning Certification Enclave. The Provisioning Certification Enclave bootstraps a shared remote attestation secret that will be used when application enclaves actually request SGX remote attestation.

The Quoting Enclave is used by the application enclave. The application enclave prepares a local attestation report first, and then sends this report to the Quoting Enclave using the local attestation mechanism. The Quoting Enclave verifies the attestation report (in particular, it verifies that the enclave runs on the same Intel CPU). If the report is correct, the Quoting Enclave converts it into a special quote (a special format that Intel services understand). The quote is signed by the key obtained from the Provisioning Certification Enclave. This quote is ultimately sent to the remote user.

To gain trust in the application enclave, the remote user receives the quote and communicates with the Datacenter Caching Service (which may communicate with the Intel Provisioning Certification Service, to receive up-to-date information about the SGX platform) to verify the validity of the quote. If the service replies positively, the remote user compares the application enclave's measurement with the one the user expects. If the result of this comparison is also positive, then the user gains trust in the application enclave and can start sending secret data to it.

In summary, the architectural enclaves serve as proxies with some opaque logic to transform the attestation report from the application enclave to a special remotely verifiable proof. They hide the complexity of dealing with security versions, expiring certificates, internet connectivity to a caching service and to the Intel Provisioning Certification Service, and so on.

A.1.4 Intel SGX component: application development tools

There are two main approaches to develop SGX applications. They are illustrated in Figure A.1.

First, the developer can choose to partition the application into the trusted and untrusted parts

[47, 30]. For this, a set of sensitive logic and data must be identified and extracted from the application into the trusted SGX enclave.

In case of legacy applications, such manual partitioning may be too complex, error-prone, or time-consuming. Therefore, the developer may choose to protect the whole application with all its data and logic. In this approach, the whole application is moved inside the SGX enclave, and its communication with the outside world is mediated through a thin layer of a helper *Library OS*.

Intel provides the Intel SGX Software Development Kit (SDK) – a framework that allows to write manually-partitioned applications for SGX [19]. However, a significant disadvantage of programming with Intel SGX SDK is that it requires a lot of manual effort to partition the application into trusted and untrusted parts. First, the developer must manually design and program the host-to-enclave interface. Depending on the size of the application and on the richness of this interface, this may be time-consuming and error-prone. Second, the developer must be very careful to bring all sensitive data and logic inside the enclave. If the developer fails to do so and leaves some sensitive data outside, in the host application, then all security guarantees are broken. Finally, the developer must also be careful to bring only the absolutely necessary data and logic inside the enclave. Otherwise, if unnecessarily many computations are performed inside the enclave, the whole application will exhibit subpar performance.

By putting the whole application inside the enclave, there is no need to perform any partitioning. Instead, Library OS frameworks like *Gramine* automatically generate ECALL/OCALL interfaces to execute only the specific logic required in the untrusted host: system calls, CPUID instructions, etc. Thus, the untrusted code of a Library OS only deals with untrusted input/output.

At startup, the Library OS framework loads the application enclave and immediately switches to enclave mode. Thus, the user application executes inside the enclave almost all of the time, except for I/O requests. At run-time, the Library OS framework only performs this minimal I/O so that the enclave can communicate with the outside world through the network, file system interface, or other system calls.

A.2 *Gramine* - A Library OS for Seamless Protection of CCAM Applications

As outlined in [10], Trusted Execution Environment is a secure area of the server that can protect confidentiality and integrity of enclosed, loaded code and data. In other words, TEEs provide a confined, isolated domain in which the application runs, and this domain appears completely opaque to other software running on the same server. We now dive deeper into the details, namely the *Gramine* Library OS that allows user-friendly trusted execution of Linux applications and the Intel *SGX* hardware security features of Intel CPUs that allow trusted execution of user-space processes in a so-called *Enclave*.

Gramine is a TEE run-time to run unmodified Linux applications on different platforms in different environments [43, 45]. For example, *Gramine* can take a Redis database Linux-x86-64 based binary and its dependent libraries, without modification or recompilation, and let it run in another environment. The currently available and most widely used configuration is running applications inside an Intel SGX enclave on top of the untrusted Linux kernel.

As discussed in the previous section, the Intel SGX technology provides powerful building blocks for application development. Software developers can port their applications to Intel SGX by

putting only the security-critical part of the application into the Intel SGX enclave and leaving the non-critical parts outside of the enclave. Several development kits can help ease the task of writing such code; Intel SGX SDK and Open Enclave SDK are two prominent examples. However, in many real-world scenarios, it is infeasible to write a new application from scratch or to port an existing application manually.

Gramine can help ease this porting burden for developers: *Gramine* supports the "lift and shift" paradigm for Linux applications, where the whole application is secured in a "push-button" approach, without source-code modification or recompilation. Instead of manually selecting a security-critical part of the application, users can take the whole original application and run it completely inside the Intel SGX enclave with the help of *Gramine*.

Gramine not only runs Linux applications out of the box, but also provides several tools and infrastructure components for developing end-to-end protected solutions with Intel SGX:

- Support for both local and remote Intel SGX attestation, with the help of RA-TLS and Secret Provisioning components.
- Transparent encryption and integrity protection of files; in particular, the Encrypted Files feature allows security-critical files to be automatically encrypted and decrypted inside the enclave.
- Optional feature of asynchronous (exitless) transitions for performance-critical applications because transitions between the enclave and the untrusted environment can be rather slow in Intel SGX.
- Full support of multi-process applications, by providing complete fork/clone/execve implementations.

Gramine currently supports many programming languages and frameworks, as well as many kinds of workloads. *Gramine* supports C/C++, Rust, Google Go, Java, Python, R and other languages, as well as database, AI/ML, web-server and other workloads. The typical performance overhead observed is around 5-20% depending on the workload.

A.2.1 Gramine architecture

Gramine is a Library Operating System with modular design, depicted in Figure A.2.

Library Operating System (LibOS) means that *Gramine* can be thought of as a re-implementation of a Linux kernel that is very small in size, somewhat limited in the functionality, and is tailored to run only one application (this application can spawn multiple threads and processes, but *Gramine* cannot run *several distrusting* applications at once as normal operating systems would do). Since *Gramine* is a Library OS, *Gramine* itself runs as a normal user-level process on the host OS. Note that *Gramine* does not use hardware-assisted virtualization and thus is not a Virtual Machine (VM).

Gramine has *modular design*. In other words, unlike Linux, *Gramine* is *not* a monolithic application. It runs as two tightly interacting components: the LibOS and the backend. Each of these two components can be switched to another implementation, independently of the other. To allow for switching between different backends *Gramine* specifies a standard API/ABI interface between the LibOS and the backend.

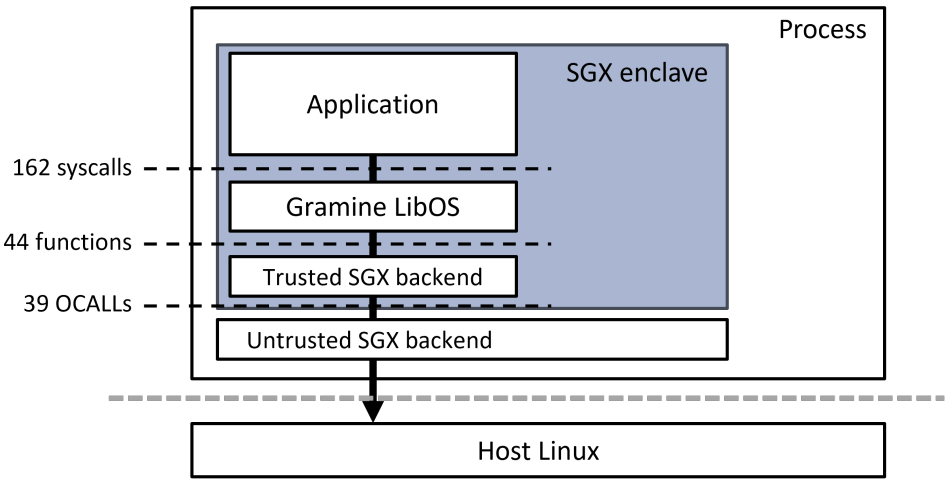

Figure A.2: Gramine architecture (with SGX backend).

Let's look at the involved interfaces at each layer. First, the application running inside *Gramine* must be a Linux application that uses a classic Linux system call (syscall) interface to request functionality from the host kernel and to communicate with the outside world. *Gramine* currently implements 162 most widely used syscalls out of 325 possible Linux syscalls. The unimplemented syscalls are mainly deprecated ones or sysadmin ones – neither of these are needed to run real-world standalone applications that *Gramine* primarily targets.

Gramine intercepts application system calls and invokes their corresponding emulations in its LibOS component. The LibOS component tries to resolve most system calls by itself, but in some cases it must forward the request to the host OS. One example of syscalls resolved inside LibOS itself are "user permission checks" – *Gramine*'s LibOS maintains metadata (shadow state) on opened files, network connections, created threads and spawned sub-processes, and consults this metadata on each syscall. An example of syscalls that are forwarded to the host OS are I/O syscalls like `recv()` and `send()`, because *Gramine* cannot communicate with the outside world without the help of the host platform. Note that all forwarded syscalls also keep relevant metadata (shadow state), and LibOS verifies this metadata for consistency and integrity.

When LibOS needs to ask the host OS for some resources or functionality, the LibOS does not simply issue a direct syscall to the host kernel. Instead, LibOS goes through one level of indirection – the environment-specific backend. The backend is needed to "adjust" the LibOS request to the capabilities of the underlying platform/environment. The backend is kept minimal and stateless, and LibOS calls into it via 44 APIs (classic C functions). The backend encapsulates all host-platform-specific and environment-specific code of *Gramine* in one small component that can be easily changed. Thus, the LibOS codebase of *Gramine* stays the same for different host platforms, and only the backend component is replaced.

The SGX backend is currently the primarily used backend of *Gramine* (the other backend is "direct" which simply forwards all requests from application to the host and is used as a debugging backend). Because the Intel SGX technology dictates separation of a process into trusted and untrusted parts, *Gramine*'s SGX backend consists of two parts: the trusted SGX backend that runs inside the SGX enclave, and the untrusted SGX backend that runs outside. The trusted backend performs OCALLs to exit the enclave and to pass control to the untrusted backend; there are 39 OCALLs in total. The untrusted backend forwards requests to the host kernel, gets back the results from the host, and re-enters the enclave.

It is important to note that these 39 SGX OCALLs are the only attack surface in *Gramine*[1]. All these OCALLs were manually inspected and verified, adding checks and validations to them, to make sure attacks from the untrusted host are thwarted.

A.2.2 Gramine Trusted Computing Base (TCB)

Having 39 SGX OCALLs (controlled enclave entry points) between the enclave and the untrusted host is a reasonable compromise between security, usability, and performance. It is a sufficiently small number of interfaces to be able to manually audit and gain trust in.

The level of security of Trusted Execution Environments (TEEs) is also frequently judged by the metric of *Lines of Code* in the Trusted Computing Base (TCB). The LibOS component of *Gramine* is about 27,000 Lines of C Code (*Gramine* is written in modern C). The SGX backend has around 21,000 Lines in the trusted part and another 4,000 in untrusted part. Note that only the LibOS component and the trusted part of SGX backend are run inside the SGX enclave, therefore the actual *Gramine* TCB for is around 48,000 Lines of Code. This is considered a small TCB for a TEE runtime that is able to run a wide range of applications.

For comparison, the codebase of the open-source version of Redis (without additional modules) is around 144,000 lines, and the tiny configuration of the Linux kernel is at least 270,000 lines of code.

A.2.3 Gramine component: manifest file

To run an application in an SGX enclave with *Gramine*, the application must be accompanied by a *manifest file* – a simple plain text configuration file. Upon startup, *Gramine* parses the manifest file and extracts all the necessary information about the application including its external dependencies and Intel SGX enclave properties. The manifest is the single most important file when porting applications inside *Gramine*. Ultimately, the security and correct functioning of the application depends on how its manifest file is written.

Below is an example how the manifest can look like for a Python application (abridged for readability):

```
libos.entrypoint = "python"
loader.env.LD_LIBRARY_PATH = "/lib"
fs.mounts = [
  { path = "/lib", uri = "file:/usr/local/lib/Gramine" },
  { type = "tmpfs", path = "/tmp" }
]
sgx.enclave_size = "1024M"
sgx.max_threads = 32
sgx.trusted_files = [ "file:/usr/local/lib/Gramine/libc.so" ]
```

Typically, the application developer writes a manifest file as a Jinja-style template with TOML formatting, and *Gramine* provides a tool to render this template into the final manifest. The manifest file contains not only the description of the application itself (its properties, its executables and dependencies, etc.) but also the description of *Gramine* (its executables and dependencies). In this

[1] This is a slight oversimplification, because in addition to OCALLs, CPUIDs and signals are also routed through the possibly malicious host and thus need to be carefully sanitized.

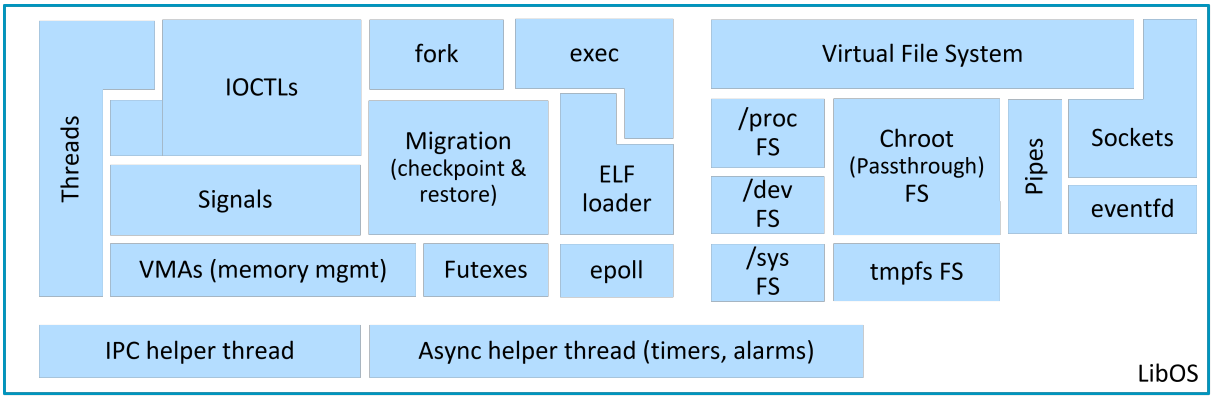

Figure A.3: Features of the Gramine LibOS component.

way, the manifest anchors a specific version of the application as well as the specific version of *Gramine* to be run inside the SGX enclave, for ease of deployment, attestation and maintenance.

Let us analyze the small example above. First, the manifest must specify the executable to load and run – in this case, the Python interpreter. Next, the manifest may overwrite the required environment variables. The manifest should also specify a subset of host-OS directories to be mounted so that only they are visible to the enclavized application (possibly under a different name as shown in the example – here the standard libraries are silently replaced with *Gramine*-patched libraries). In addition to host-OS directories, virtual in-enclave-memory mount points can be specified, e.g. "tmpfs" file system mounts. Next, the manifest contains SGX-architecture-specific variables like the maximum enclave size and the maximum number of simultaneous enclave threads. Finally, this example shows how files to be consumed by the SGX enclave must be marked as trusted – in this case, *Gramine* calculates the files' secure hashes during build time, appends these hashes to the manifest, and during run-time *Gramine* will verify that the files were not modified.

There are several more manifest options in *Gramine*, to enable some additional functionality. For a complete list, please refer to *Gramine* documentation at `https://Gramine.readthedocs.io/en/stable/manifest-syntax.html`.

A.2.4 Gramine component: LibOS

The LibOS component may be thought of as a tiny re-implementation of the Linux kernel (see Figure A.3). But with one significant difference – the Linux kernel relies on and communicates only with the underlying hardware, whereas *Gramine* LibOS relies on 44 functions from the backend component and communicates with the outside world through these 44 functions. This explains why the implementation of *Gramine*'s LibOS is only one tenth of the modern Linux kernel.

Gramine intercepts all application requests to the host OS in its LibOS component. Some of these requests are processed entirely inside the LibOS, and some are funneled through a thin API to the backend and ultimately to the host OS. Either way, each application's request and each host's reply are verified for correctness and consistency. For these verifications, *Gramine* maintains internal, "shadow" state inside the LibOS. This way, *Gramine* defends against Iago attacks.

Gramine's LibOS strives to be 100% compatible with the Linux kernel, even when it deviates from standards like POSIX ("bug-for-bug compatibility"). At the same time, *Gramine* is minimalistic,

and implements only the most important subset of Linux functionality, enough to run portable, hardware-independent applications.

LibOS implements 162 system calls out of around 360 system calls available on Linux. Many system calls are implemented only partially, typically because real world workloads do not use the unimplemented functionality. Some system calls are not implemented because (i) they are deprecated in Linux, (ii) they are unused by real world applications or (iii) they do not fit the purpose of *Gramine* of virtualizing only a single application [44, 4]. The list of implemented system calls grows with time, as *Gramine* adds functionality required by real world workloads.

The LibOS implements the usual operating systems' concepts, objects, and flows. There is a sub-component to spawn new threads and manage them, abstractions for memory management via Virtual Memory Areas (VMAs), Linux-style multi-processing support of `fork()` and `execve()`, code to handle synchronous and asynchronous signals, code for process migration, code for loading ELF binaries, implementation of futexes, `poll()` and `epoll()` on file descriptors, a virtual file system, etc.

Below is a quick overview of most important supported features in *Gramine*'s LibOS component:

- *Gramine*'s LibOS supports multi-processing. A *Gramine* instance starts the first (main) process, as specified in the entrypoint of the manifest. The first process can spawn child processes, which belong to the same *Gramine* instance. *Gramine* can execute ELF binaries (executables and libraries) and executable scripts. *Gramine* supports executing them as entrypoints and via the *execve()* system call. In case of SGX backend, *execve()* execution replaces a calling program with a new program in the same SGX enclave. *Gramine* also supports creating child processes using *fork()*, *vfork()* and *clone()* system calls.
- *Gramine*'s LibOS implements multi-threading. In case of SGX backend, all threads of one *Gramine* process run in the same SGX enclave.
- *Gramine*'s LibOS does *not* perform scheduling of threads, instead it relies on the host OS to perform scheduling. In case of SGX backend, trying to perform or control scheduling would be futile because SGX threat model has no means of control or verification of scheduling decisions of the host OS.
- *Gramine*'s LibOS implements most of the Linux IPC mechanisms. LibOS implements pipes, FIFOs and UNIX domain sockets (UDSes) via host-OS pipes. For SGX backend, all pipe, FIFO and UDS communication is transparently encrypted. For all other IPC mechanisms (signals, process state changes, file locks) LibOS emulates them via internal message passing (in case of SGX, all messages are encrypted).
- *Gramine*'s LibOS partially implements signals. For local signals (*Gramine* process signals itself, e.g. `SIGABRT`) and signals from the host OS (e.g. host sends `SIGTERM`), message passing is not involved. For process-to-process signals (e.g. child process sends `SIGCHLD` to the parent), message passing is used.
- *Gramine*'s LibOS supports the most important networking protocols: TCP/IP and UDP/IP.

A.2.5 Gramine component: File systems

Gramine implements file system operations, but with several peculiarities and limitations described below. *Gramine* does not implement full file system stack by design. *Gramine* relies

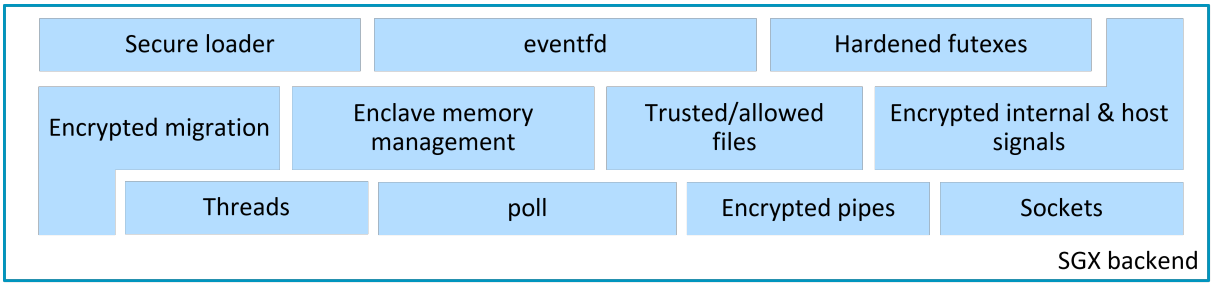

Figure A.4: Features of the Gramine SGX backend component.

on the host file system for most operations. The only exceptions are the “tmpfs” file system and the pseudo-file systems, which are implemented entirely inside *Gramine*.

The most important peculiarity is that *Gramine* does not simply mirror the host OS’s directory hierarchy. Instead, *Gramine* constructs its own view on the selected subset of host’s directories and files: this is controlled by the manifest’s FS mount points. This feature is similar to the “volumes” concept in Docker. This *Gramine* feature is introduced for security.

Another peculiarity is that *Gramine* provides several types of filesystem mounts:

- passthrough mounts (contain unencrypted files, see below),
- encrypted mounts (contain files that are automatically encrypted and integrity-protected).

Passthrough mounts must be of one of two kinds:

- containing allowed files (not encrypted or cryptographically hashed),
- containing trusted files (cryptographically hashed for integrity protection).

Additionally, mounts may be hosted in one of two ways:

- on the host OS (in passthrough mounts),
- inside the *Gramine* process (in tmpfs mounts).

All files potentially used by the application must be specified in the manifest file. Instead of single files, whole directories can be specified.

Gramine also provides a subset of pseudo-files that can be found in a Linux kernel. In particular, *Gramine* automatically populates `/proc`, `/dev` and `/sys` pseudo-file systems with most widely used pseudo-files.

A.2.6 Gramine component: SGX backend

As mentioned before, the LibOS component relies on the 44 functions implemented in the underlying backend.

There are several backends available in *Gramine*, one for each supported environment. The most basic backend is the Linux one (also called “direct”) – it allows *Gramine* to run on a normal Linux kernel on x86-64 machines. This Linux backend implements 44 functions which serve as

a straight-forward "glue code" between what LibOS wants from the underlying platform and what the underlying platform offers. In particular, the direct backend contains a simple ELF loader, functions to facilitate process migration, memory management through Linux's `mmap()`/`munmap()`, thread management via Linux's `clone()`/`exit()`, etc.

To run an application inside an SGX enclave, it is required to swap the direct backend for an SGX backend. The SGX backend exposes the same 44 functions to the LibOS, but all these functions are hardened to secure the application inside the enclave. All subsystems of the SGX backend have additional checks, verifications, and, if needed, transparent encryptions.

For example, the migration of the checkpointed state from the parent process to the child process is transparently encrypted (to prevent confidential data leakage). Similarly, pipes, messages and signals between *Gramine* processes are encrypted. Also, files mounted as "protected" in the manifest, are transparently encrypted and integrity-protected. Furthermore, additional checks are added to prevent Iago attacks.

Note that network-socket communication is *not* protected by *Gramine*. This is because all modern applications already enable the SSL/TLS protocol to protect their TCP/IP communications, so there is no need to additionally re-encrypt network traffic.

A few subsystems in the SGX backend are only lightly protected. This is because there is not much harm that can be done by a malicious host OS in e.g. thread management. The only issues that can arise are application starvation or Denial of Service (DoS) attacks because the malicious OS refuses to schedule enclave threads or give network data to it. But this is out of the threat model of Intel SGX, and thus not protected against in *Gramine*.

A.2.7 Gramine component: SGX attestation

An important aspect of any Trusted Execution Environment is local and remote attestation. Local attestation is used when two TEEs run on the same physical machine and remote attestation is used when a user attests a TEE running on a remote physical machine.

Remote attestation in Intel SGX comes in two flavours: EPID [46] and DCAP [32]. The former is used in client machines whereas the latter is used in data center environments. *Gramine* supports both EPID and DCAP attestation schemes.

Gramine provides support for three levels of attestation flows:

- Local and remote SGX attestation – low-level interface that is exposed to the application via the `/dev/attestation` pseudo-file system. It exposes the low-level abstractions of SGX report and SGX quote through a set of pseudo-files. On the one hand, this interface is flexible and allows to construct arbitrary attestation flows. On the other hand, this interface requires modifications to the application.
- Secure channel – mid-level interface provides the *RA-TLS library* shipped together with *Gramine*. This library offers an abstraction of a secure TLS channel, but with specially-crafted X.509 certificates that embed the SGX quote. RA-TLS uses raw `/dev/attestation` pseudo-files under the hood.
- Secret provisioning – high-level interface is the *Secret Provisioning library* that is also shipped together with *Gramine*. This library can be pre-loaded to move the secrets (such as private keys and passwords) into the SGX enclave, without any modifications to the application itself. Secret Provisioning library uses RA-TLS under the hood.

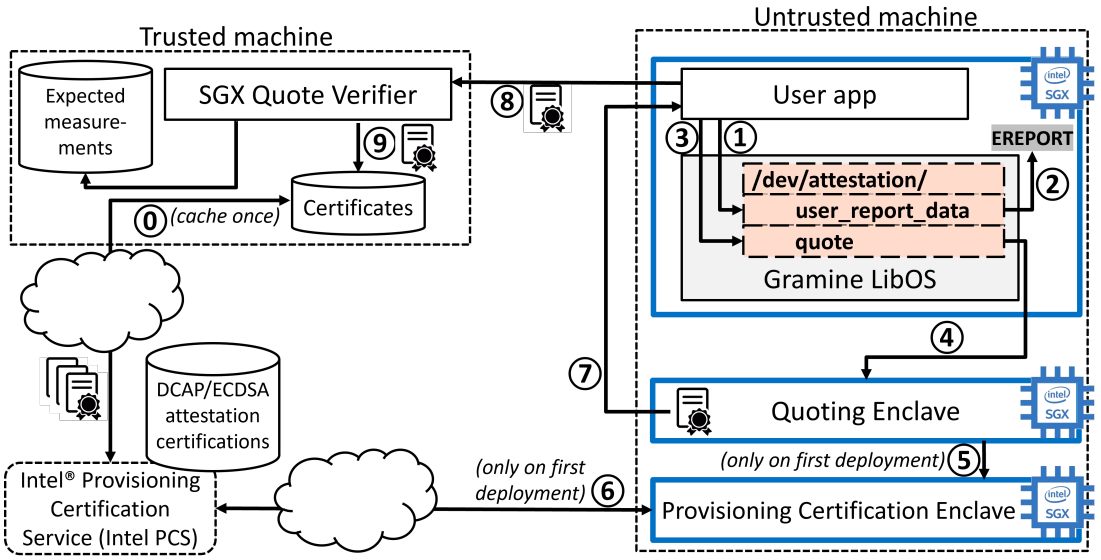

Figure A.5: *Gramine* SGX remote attestation using DCAP flows.

Applications running under *Gramine* can use each of the above three levels to build their attestation flows. Each next level builds on the previous one and exposes a simpler API to the application (but also is more restricted in its functionality).

For the sake of brevity, we will omit the details of EPID SGX attestation and concentrate on the DCAP SGX attestation scheme.

Figure A.5 shows DCAP based remote attestation. The user application runs in an SGX enclave on a remote untrusted machine, whereas the end user waits for the attestation evidence from this enclave on a trusted machine.

DCAP attestation starts with the enclavized user application verifying that it runs inside *Gramine* and opening the special file `/dev/attestation/user_report_data` for write (step 1). Under the hood, *Gramine* uses the `EREPORT` hardware instruction to generate an SGX Report (step 2). After the SGX report is generated, the application opens another special file `/dev/attestation/quote` for read (step 3). Under the hood, *Gramine* communicates with the Quoting Enclave to receive the SGX Quote (step 4). The Quoting Enclave communicates with the architectural Provisioning Certification Enclave (PCE) (step 5). The PCE connects to the Intel service called Intel Provisioning Certification Service (PCS) to obtain the attestation collateral: attestation certificates and certificate revocation lists for the SGX machine (step 6). After obtaining all relevant collateral, the Quoting Enclave generates the SGX quote from the provided-by-application SGX report and sends it back to the enclavized user application (step 7). The application stores this SGX quote in its enclave memory and can later send it to the remote user (verifier) upon request. When the remote user wants to validate the SGX enclave, it requests remote attestation with it, and the enclavized application forwards the SGX quote to the remote trusted machine (step 8). Finally, the remote user consults the cached DCAP attestation certificates that the user is supposed to periodically fetch from the Intel Provisioning Certification Service (PCS) (preliminary step 0). The user compares the certificates embedded in the received SGX quote against these cached certificates (step 9). Finally, the remote user also verifies the enclave measurements embedded in the SGX quote against the expected ones. After this verification procedure, the remote user can trust the SGX enclave on the untrusted machine and start sending inputs/receiving enclave outputs.

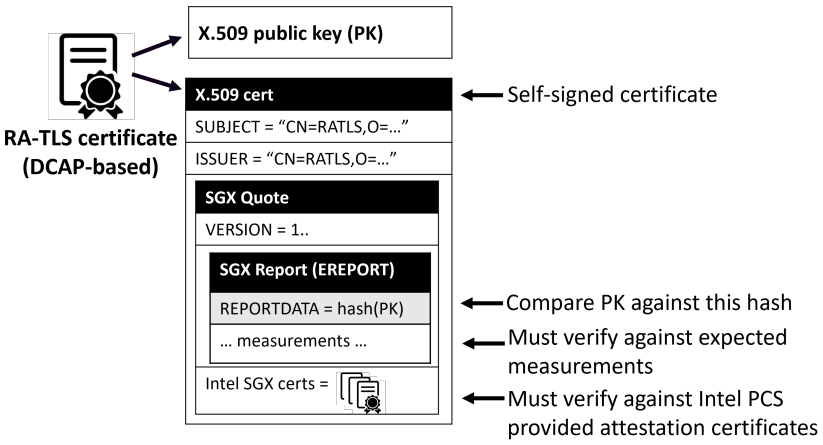

Figure A.6: Gramine SGX remote attestation: RA-TLS X.509 certificate.

RA-TLS interface hides the complexity of the low-level `/dev/attestation` flows and provides a simple and powerful abstraction of a TLS connection between the TEE and the remote user (enhanced with remote-attestation flows) [27]. Using RA-TLS, the application can securely send and receive arbitrary data to/from the remote user. RA-TLS is currently tied to Intel SGX but can be adapted for other TEEs.

RA-TLS integrates Intel SGX remote attestation into the TLS connection setup. Conceptually, it extends the standard X.509 certificate with SGX-related information (SGX quote). The additional information allows the remote user (verifier) of the certificate to verify that it is indeed communicating with an SGX enclave (attester).

Figure A.6 shows the standard X.509 certificate generated by RA-TLS (the figure shows the DCAP based RA-TLS certificate, but the EPID based RA-TLS certificate is conceptually similar). This certificate is self-signed because the actual chain of trust is stored in the Intel SGX certificates embedded in the SGX quote. The most important concept behind the RA-TLS certificate is that it embeds the SGX quote (in one of the unused X.509 extension fields), which in turn embeds the SGX report and the complete Intel SGX certificate chain. Therefore, the RA-TLS certificate contains all the SGX-relevant information. Note how the SGX report's `REPORTDATA` field contains the secure hash of the ephemeral public key generated by the enclavized application – this is how the RA-TLS certificate is tied to the enclavized application that generated it.

Appendix B

Glossary and User Roles

A-ECU An *A-ECU* is an *ECU* with a *TEE* providing secure storage for keys and other data. it is able to do asymmetric and symmetric cryptography.

AIV Attestation and Integrity Verification.

Application Developer The *Application Developer* is responsible for designing and implementing applications for the CONNECT framework.

ATL The Actual Trust Level (ATL) reflects the result of an evaluation of a trust proposition, for a specific CCAM actor, as defined in a trust model managed by the CONNECT Trust Assessment Framework [9]. It quantifies the extent to which a certain node or data can be considered trustworthy based on the available evidence.

C-ACC Co-operative Adaptive Cruise Control.

CCAM The European Commission has on 30th of November 2016 adopted a European Strategy on Cooperative Intelligent Transport Systems (C-ITS), a milestone initiative towards cooperative, connected and automated mobility. The objective of the C-ITS Strategy is to facilitate the convergence of investments and regulatory frameworks across the EU, in order to see deployment of mature C-ITS services in 2019 and beyond [6].

Cloud Administrator The *Cloud Administrator* is responsible for managing the mobile edge cloud *MEC*. This includes managing capacity and keeping the software up-to-date..

CoCo Is a new project of the Cloud Native Computing Foundation (CNCF) that enables cloud-native confidential computing by taking advantage of a variety of hardware platforms and technologies, such as the hardware-based Trusted Execution Environment (HW-TEE). The broader scope is to secure data in use at a pod-level, removing trust assumptions on the cloud side by introducing confidentiality and integrity protection during runtime, through the use of a trusted hardware. They provide resource isolation, data protection, and remote attestation. The Confidential Containers project offers an approach for protecting cloud native applications in a HW-TEE, without requiring further modifications to the container image during the development process.

DLT Distributed Ledger Technology.

ECU An electronic control unit (ECU), also known as an electronic control module (ECM). In automotive electronics it is an embedded system that controls one or more of the electrical systems or subsystems in a car or other motor vehicle.

Enclave Intel SGX is a TEE provided by Intel CPUs that allows to execute a user-space process within a hardware-protected execution environment that is called *enclave*.

ETSI European Telecommunications Standards Institute.

IAM Identity and Authentication Management.

IMA Intersection Movement Assistance.

KBS The Key Broker Service (KBS) helps a source enclave to establish a trusted relationship with a enclave that is the target of a migration or upgrade. It verifies that the target is authorited and trusted and then provides a key to source and target enclave that can be used for state migration during upgrade/migrate..

KRPE The Key Restriction Usage Policy Engine (KRPE) is a *CONNECT* newly developed concept for enabling the vision of local attestation. This, essentially, allows for the verification of an extended set of device characteristics (i.e., device integrity, safety, etc. depending on the type of trust properties considered as part of the respective trust model) without the need to disclose the actual attestation evidence. The verification ofushc values that actually represent the current state of the device, is checked against a policy that binds the usage of a signing Attestation key only in the case where the device is at an expected state. Thus, a Verifier can check the validity of the transmitted signature to assert on the correct state of the host device.

Manifest File The Manifest File of SGX that specifies the hash value and policies of an application to be executed within an *SGX* enclave. For integrity-protection, the manifest is signed by the *Application Developer*.

MBD The Mis-behaviour Detector (MBD) component monitors the data from the vehicle and from elsewhere (from CPM/CAM messages) and looks for anomalies. If these are detected is sends mis-behaviour reports to the TAF and outside of the vehicle. Reports for the TAF will be ‘normally’ signed, while those being sent outside will be anonymously signed.

MEC The MEC serves a number of functions. It makes more powerful computing resources available to vehicles. These resources are provided close to the edge of the network so that calculations can be ‘outsourced’ by the vehicles and still meet the necessary low latency requirements. It can also combine information from vehicles in its vicinity to produce a more detailed map of their positions and trajectories and feed this back to them together with its assessment of their trustworthiness.

OEM An *Original Equipment Manufacturer*. In the context of CONNECT the OEM is the vehicle manufacturer who, often in association with a *Tier 1* supplier, designs, assembles, markets and sells the vehicle.

PKI Public Key Infrastructure.

RoT The (Hardware) Root of Trust (RoT) is the minimal set of security guarantees usually provided by the hardware that is sufficient to guarantee the security of a larger TCB.

S-ECU An *S-ECU* is an *ECU* with secure storage for keys and other data, possibly a *System on Chip (SoC)* with an HSM. It can only do symmetric crypto.

SGX *Intel SGX* is a hardware feature of Intel CPUs that provides a *TEE* for user-space applications on Intel CPUs. The goal is to protect an application from unauthorized access or modification by any component outside the TEE. I.e. neither the operating system nor other untrusted applications should be able to breach the confidentiality or integrity of the protected application.

SoC A *system on a chip or system-on-chip (SoC)* is an integrated circuit that integrates most or all components of a computer or other electronic system. These components almost always include on-chip central processing unit (CPU), memory interfaces, input/output devices and interfaces, and secondary storage interfaces, often alongside other components such as radio modems and a graphics processing unit (GPU) – all on a single substrate or microchip.

TAF The TAF component does the trust assessments and forms trust opinions on the vehicle and data. The trust opinion on the data is sent outside the vehicle and needs to be anonymously signed.

TCB The *Trusted Computing Base (TCB)* of a computer system is the set of all hardware, firmware, and/or software components that are critical to its security, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system. By contrast, parts of a computer system that lie outside the TCB must not be able to misbehave in a way that would leak any more privileges than are granted to them in accordance to the system's security policy.

TCH The Trustworthiness Claims Handler (TCH) is the component responsible for sharing all trust-related information outside the Vehicle in a privacy-preserving manner. This data bundle (encoded in the context of a VP) comprises Trustworthiness Claims (TCs), the Trust Opinion (produced by the TAF) and the Misbehavior Report (produced by the MBD). The TC is usually produced (by the Attester) so as to provide trustworthiness evidence ("Trust Source") that can be used for appraising the trustworthiness level of the Attester in a **measurable** and **verifiable** manner. Measurable reflects the ability of the TAF to assess an attribute of the Attester against a pre-defined metric (e.g., RTL) while verifiability highlights the need for all claims to have integrity, freshness and to be provably & non-reputably bound to the identity of the original Attester. Examples sets of TCs might include (among other attributes) evidence on system properties including: (i) integrity in the context that all transited devices (e.g., ECUs) have booted with known hardware and firmware; (ii) safety meaning that all transited devices are from a set of vendors and are running certified software applications containing the latest patches and (iii) communication integrity.

TEE A *Trusted Execution Environment* allows to execute applications while enforcing well-defined security policies for a given application. An example is *SGX*.

TEE-GSE The TEE Guard Security Extensions (TEE-GSE) is the set of security controls, developed within CONNECT, for supporting the secure life-cycle management of a CCAM actor: from the **secure on-boarding and enrollment** of all CCAM applications/services, instantiated in the vehicle and/or *MEC*, and *CONNECT*-related security components including

the establishment of the necessary cryptographic primitives (for their later interactions with other *CCAM* actors via secure and authenticated communication channels) to the **run-time monitoring and extraction of system measurements/properties**, serving as trustworthiness evidence, and **reaction policy enforcement mechanisms to any indication of risks and changes in the trust state of a device** (state migration of a device).

Tier 2 A *Tier 2 supplier* provides components to the *Tier 1* suppliers and is the next level in the supply chain. Tier 2 suppliers may not just provide components for the automotive industry, but other industries as well. For CONNECT we focus on the suppliers of ECUs (micro-controllers) used in the vehicle and their role in providing identity keys for them.

Tier 1 A *Tier 1 supplier* directly supplies *OEMs* with components that are ready for installation into the vehicle. They work closely with the *OEM* at all stages of a vehicle's development. The Tier 1 supplier may well work with several manufacturers on the development of their vehicles. The Tier 1 supplier will obtain the components that they need from *Tier 2* suppliers.

TMC The Trusted Monotonic Counter (TMC) allows a TEE to create, increase, test, or destroy a given counter with a unique id. The key security objectives are (a) that all stakeholders see the same value and (b) once increased, the counter cannot be decreased at all. This service can be used to implement rollback-protection or guarantee that only one instance of a service is running at any point in time..

VC Vehicle Communication (V2X) This provides communication facilities for the vehicle. Connectivity – automotive ethernet, 5G, V2X.

VP The Verifiable Presentation (VP) is the data structure used for disclosing only a subset of the trust-related information needed for the receiving entity to evaluate the trust level of the originator. This allows the *TCH* to construct data bundles that hold the Trust Opinion, Misbehavior Report and "abstracted" attestation assertions, as described in D5.1 [11].

Zonal controller (ZC) The *Zonal controller (ZC)* is an *A-ECU* that acts as a gateway between the ECUs and the vehicle computer. As an *A-ECU* they will have a *TEE* providing secure storage for keys and other data and will be able to do asymmetric and symmetric cryptography.

References

[1] ISO/SAE 21434:2021. Road vehicles Cybersecurity engineering. *ISO/TC 22/SC 32 Technical Standard*, 2021. https://www.iso.org/standard/70918.html.

[2] 5GAA Automotive Association. Cross Working Group Work Item gMEC4AUTO: Global MEC Technology to support automotive services - Cybersecurity for Edge Computing. Technical report, 5GAA Automotive Association, 2023.

[3] Brewer Eric A. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*. Association for Computing Machinery, 2015.

[4] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: the old, the new, and the missing. In *EuroSys*, 2016.

[5] Enclave cc Community. enclave-cc: Process-based confidential container runtime. https://github.com/confidential-containers/enclave-cc, 2023. [Online; accessed Aug-2023].

[6] Cooperative, connected and automated mobility (CCAM). https://transport.ec.europa.eu/transport-themes/intelligent-transport-systems/cooperative-connected-and-automated-mobility-ccam_en.

[7] Confidential Containers Community. Confidential containers. https://github.com/confidential-containers/confidential-containers/, 2023. [Online; accessed Aug-2023].

[8] Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A Wood. Two-round threshold schnorr signatures with frost. Technical report, Internet-Draft draft-irtf-cfrg-frost-15, Internet Engineering Task Force, 2022.

[9] The CONNECT Consortium. Architectural specification of connect trust assessment framework, operation and interaction. Deliverable D3.1, Project 101069688 within HORIZON-CL5-2021-D6-01, Jun. 2023.

[10] The CONNECT Consortium. Conceptual architecture of customisable tee & attestations. Deliverable D4.1, Project 101069688 within HORIZON-CL5-2021-D6-01, Dec. 2023.

[11] The CONNECT Consortium. Distributed processing and CCAM trust functions offloading & data space modelling. Deliverable D5.1, Project 101069688 within HORIZON-CL5-2021-D6-01, Nov. 2023.

[12] The CONNECT Consortium. Operational landscape, requirements and reference architecture - initial version. Deliverable D2.1, Project 101069688 within HORIZON-CL5-2021-D6-01, Nov. 2023.

[13] The CONNECT Consortium. Distributed processing, fast offloading and MEC-enabled orchestrator. Deliverable D5.2, Project 101069688 within HORIZON-CL5-2021-D6-01, Mar. 2024.

[14] The CONNECT Consortium. Integrated framework (first release) and use case analysis. Deliverable D6.1, Project 101069688 within HORIZON-CL5-2021-D6-01, May 2024.

[15] The CONNECT Consortium. Trust & risk assessment and CAD twinning framework (initial version). Deliverable D3.2, Project 101069688 within HORIZON-CL5-2021-D6-01, February 2024.

[16] The CONNECT Consortium. Virtualization- and edge-based security and trust extensions (first release). Deliverable D4.2, Project 101069688 within HORIZON-CL5-2021-D6-01, Jan. 2024.

[17] The CONNECT Consortium. MEC-enabled orchestrator, continuous authorization, trust management and DLT-based secure information exchange. Deliverable D5.3, Project 101069688 within HORIZON-CL5-2021-D6-01, Mar. 2025.

[18] The CONNECT Consortium. Virtualization- and edge-based security and trust extensions (final release). Deliverable D4.3, Project 101069688 within HORIZON-CL5-2021-D6-01, Mar. 2025.

[19] Intel Corporation. Intel SGX for Linux. https://github.com/intel/linux-sgx, 2023. [Online; accessed Aug-2023].

[20] Heini Bergsson Debes, Edlira Dushku, Thanassis Giannetsos, and Ali Marandi. Zekra: Zero-knowledge control-flow attestation. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '23, page 357–371, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Heini Bergsson Debes and Thanassis Giannetsos. Zekro: Zero-knowledge proof of integrity conformance. In *ARES '22*, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Heini Bergsson Debes and Thanassis Giannetsos. Retract: Expressive designated verifier anonymous credentials. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1084–1101. IEEE, 2019.

[24] European Telecommunications Standards Institute (ETSI). Network Functions Virtualisation (NFV) Trust; Report on Attestation Technologies and Practices for Secure Deployments. ETSI Group Report GR NFV-SEC 007 V1.1.1, ETSI, October 2017. https://www.etsi.org/deliver/etsi_gr/NFV-SEC/001_099/007/01.01.01_60/gr_nfv-sec007v010101p.pdf.

[25] IETF Remote Attestation Working Group. Tee device interface security protocol (tdisp). RFC 18268, August 2022.

[26] Andreas Reuter Jim Gray. *Transaction Processing*. Morgan Kaufmann, 1992.

[27] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating Remote Attestation with Transport Layer Security, 2019.

[28] Chelsea Komlo and Ian Goldberg. Frost: Flexible round-optimized schnorr threshold signatures. In *Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020, Revised Selected Papers 27*, pages 34–65. Springer, 2021.

[29] Dimitrii Kuwaiskii. [State migration] Dumping Docker containers · Issue #1 · dimakuv/gramine-connect. https://github.com/dimakuv/gramine-connect/issues/1.

[30] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rudiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[31] Liangkai Liu, Sidi Lu, Ren Zhong, Baofu Wu, Yongtao Yao, Qingyang Zhang, and Weisong Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2021.

[32] John P Mechalas. Quote Generation, Verification, and Attestation with Intel® Software Guard Extensions Data Center Attestation Primitives (Intel® SGX DCAP). https://www.intel.com/content/www/us/en/developer/articles/technical/quote-verification-attestation-with-intel-sgx-dcap.html, 2021. [Online; accessed Aug-2023].

[33] Microsoft. Confidential consortium framework. http://ccf.microsoft.com/.

[34] Microsoft. Confidential consortium framework - glossary. https://microsoft.github.io/CCF/main/overview/glossary.html#term-CFT.

[35] Microsoft. Confidential consortium framework - governance. https://microsoft.github.io/CCF/main/governance/index.html.

[36] Microsoft. Confidential consortium framework - recovery. https://microsoft.github.io/CCF/main/operations/recovery.html.

[37] Aaron Miller, Kyungzun Rim, Parth Chopra, Paritosh Kelkar, and Maxim Likhachev. Cooperative perception and localization for cooperative driving. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1256–1262, 2020.

[38] Jianyu Niu, Wei Peng, Xiaokuan Zhang, and Yinqian Zhang. NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 2385–2399, New York, NY, USA, November 2022. Association for Computing Machinery.

[39] Kubernetes Projekt. Pod Lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/.

[40] The raft consensus algorithm. https://raft.github.io/.

[41] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, STC '06, pages 27–42, New York, NY, USA, November 2006. Association for Computing Machinery.

[42] Stefano Tessaro and Chenzhi Zhu. Revisiting bbs signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 691–721. Springer, 2023.

[43] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *EuroSys*, 2014.

[44] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald Porter. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. In *EuroSys*, 2016.

[45] Chia-Che Tsai, Mona Vij, and Donald Porter. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC*, 2017.

[46] Sardar Muhammad Usama, Quoc Do Le, and Fetzer Christof. Towards Formalization of Enhanced Privacy ID (EPID)-based Remote Attestation in Intel SGX. In *Euromicro Conference on Digital System Design (DSD)*, 2020.

[47] N. van Ginkel, R. Strackx, and F. Piessens. Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In *Asian Symposium on Programming Languages and Systems*, 2017.