CCAM TRUST & RESILIENCE

D4.1:Conceptual Architecture of Customizable TEE and Attestation Models Specifications

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 / D4.1 / 01.02 - April 4, 2024
Workpackage:	WP 4
Due Date:	M15 - 30th November, 2023
Actual Submission Date:	14th December 2023
Responsible Organisation:	INTEL
Editor:	Matthias Schunter
Dissemination Level:	PU – Public
Revision:	01.02 - April 4, 2024

Abstract:

Deliverable D4.1 documents the initial results of Workpackage 4. It specifies the core security architecture of CONNECT revolving around the use of HW-based trust anchors for supporting both the continuous trust assessment of the CCAM-wide continuum as well as the secure lifecylce management of all comprised devices. D4.1 refines the architecture (components and flows) presented in D2.1 and puts forth a deteiled description of all interactions and sequence of actions with the underlying Trusted Computing Base for providing node- and data-centric trustworthiness evidence. Furthermore, it documents user stories that specify desired features, and finally outlines the key hierarchy and formalized requirements. This overarching TEE architecture will guide the design of all security protocols and cryptographic mechanisms to be documented in D4.2.

Keywords: Trusted Execution Environment, 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, Nikolaos Fotos, Stefanos Vasileiadis, Thanassis Giannetsos (UBITECH)
Dimitrios Stavrakakis, Dmitrii Kuvaiskii, Matthias Schunter, Sergej Schumilo (INTEL)
Christopher Newton (SURREY)

Disclaimer

Executive Summary

Deliverable D4.1 documents the initial results of the *CONNECT* Workpackage WP4. It specifies the user stories and core security architecture of *CONNECT*. The questions that this deliverable answers are ”How can hardware security and a hardware root of trust be used to protect critical *CONNECT* components?”, ”How can the trustworthiness of *CONNECT* components and services be validated?”, and ”What keys are required and how can them be used?”.

The deliverable is structured as follows: We first survey state of the art and basic concepts of ”Trusted Computing”. We then provide additional details on the high-level architecture that was outlined in Deliverable D2.1: We provide more detail on architectural components, the key management and the underlying Trusted Execution Environment with a focus on Intel SGX that is provided by many of today’s off-the-shelf CPUs. To document the requirements of different user groups, we then provide *User Stories*. These User Stories specify the desired behavior and bring the architecture to live. Each user story describes a desired usage by certain user groups or user roles. This includes user stories from Task 4.1, 4.2, 4.3, and 4.5. We then outline our hardware-based trusted execution architecture based on Intel SGX in more detail. This provides design and implementation details underpinning the corresponding user stories that were documented earlier. It includes outputs of Tasks 4.1 and 4.2. A focus is on improving usability of Intel SGX by introducing and extending the Gramine Library OS. This Library OS enables developers to seamlessly transform Linux-style application into applications that run within the protected space (”enclave”) that is provided by Intel SGX. We conclude the deliverable with a first attempt at formalizing the security requirements for our key management. This documents results of Tasks 4.3, 4.4, and 4.5.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Overview

1.1 Scope and Purpose

This deliverable constitutes the first outcome of *CONNECT* WP4. It provides an overview over the security architecture of *CONNECT* with a focus on concepts for hardware-protected execution in so-called Trusted Execution Environments (TEE).

One goal of *CONNECT* is to define a security architecture for the vehicle. This deliverable provides a high-level overview over this architecture. The security architecture will address multiple key objectives:

- Define and introduce key concepts of Trusted Computing.
- Identify the key security components of the architecture that constitute the Trusted Computing Base of *CONNECT* and identify, key management, flows, and document the Trusted Execution Environment that will be used as the Root of Trust for *CONNECT*.
- Document user stories that describe how users can use this architecture and what security requirements we aim to guarantee for those users.
- Provide details on the Trusted Execution Environment and its required extensions for *CONNECT*.
- Outline formalized security and operational assurance requirements.

This document will be the foundation for implementing our Trusted Execution Environment that is the foundation of the project and will also be used as a starting point of future refinement that will be published as D4.2 in Project Month 18.

1.2 Relationship with other WPs & Deliverables

With the documentation of the detailed internal architecture of the *CONNECT* Trusted Execution Environment (TEE) and the envisioned extensions, as the core building block for safeguarding all operations supporting the **continuous trust assessment of any actor and/or data object**, this

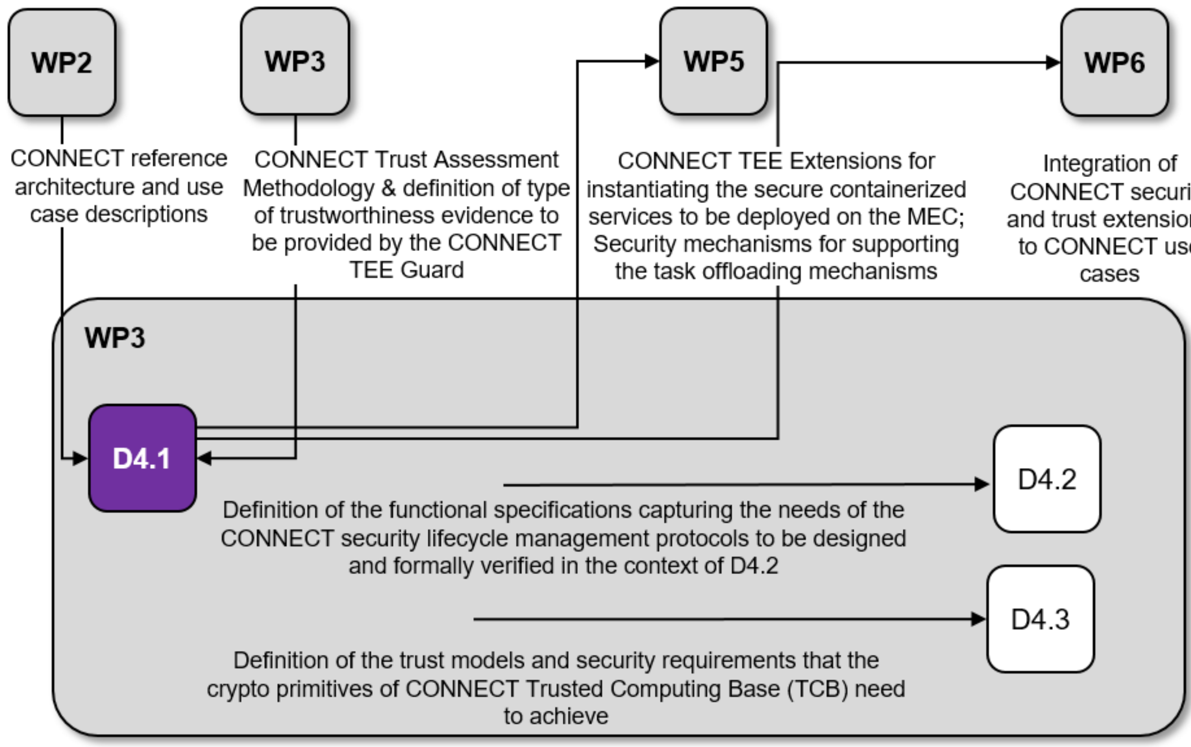
WP2
WP3
WP5
WP6
CONNECT reference architecture and use case descriptions
CONNECT Trust Assessment Methodology & definition of type of trustworthiness evidence to be provided by the CONNECT TEE Guard
CONNECT TEE Extensions for instantiating the secure containerized services to be deployed on the MEC; Security mechanisms for supporting the task offloading mechanisms
Integration of CONNECT security and trust extensions to CONNECT use cases
WP3
D4.1
D4.2
Definition of the functional specifications capturing the needs of the CONNECT security lifecycle management protocols to be designed and formally verified in the context of D4.2
D4.3
Definition of the trust models and security requirements that the crypto primitives of CONNECT Trusted Computing Base (TCB) need to achieve

Figure 1.1: Relation of D4.1 with other WPs and Deliverables.

Deliverable (D4.1) sets the scene for the design of all respective security protocols and mechanisms that will constitute the core innovation of WP4 in the context of the provided Trusted Computing Base (TCB). More specifically, it provides a detailed documentation of the functionalities (**through an extended list of engineering stories depicting the functional specifications of each envisioned trust enabler**) that need to be supported for covering all secure life-cycle management protocols and interfaces (that will be described in detail in the context of D4.2 [14]): from the **secure on-boarding and enrollment** of all applications/services, 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).

All these operations will be wrapped within the *CONNECT* TCB that is responsible for exposing the appropriate set of well defined TEE Device Interfaces (TDIs) linked to the run-time monitoring of an extended set of device characteristics, serving as evidence to the *CONNECT* Trust Assessment Framework (D3.1 [11]). On top of that, *CONNECT* will be offering novel TEE Guard Extensions (*TEE-GSE*) for enabling the functionality of such advanced secure life-cycle control mechanisms in the context of of secure containers. The are built on top of the TCB (each container will be equipped with its own *CONNECT* TCB) for continuously interacting with the trust anchor so as to then be able to provide the necessary evidence to all other requesting components for been able to perform any of the operations described in the next section.

In this context, Figure 1.1 depicts the direct and indirect relationships of D4.1 to the other Tasks and Work Packages (WPs). First, it commences with a detailed State-of-the-Art analysis on the various types of secure elements that can serve as the underlying Root-of-Trust (RoT) for supporting the secure execution and enforcement of various trust policies (as deployed by the *CONNECT* TAF). As it will be elaborated, while *CONNECT* remain **agnostic** to the type of RoT used, all protocols will be instantiated over the Grammine technology (under the umbrella of the Intel SGX TEE). Intel's Gramine technology was selected due to its capability to allow for the transformation of a software binary to its trusted-equivalent, executing in an isolated environment, with no update requirements in the code and without affecting its inter-dependencies with other parts of the loaded software stack. Essentially, it allows the instantiation of any binary to be launched as part of an isolation environment called an enclave. This is a rather new technology, developed under the umbrella of the well-established SGX TEE technology, where *CONNECT* envisions to provide concrete trust extensions for allowing for the secure communication between the "trusted" and "untrusted" worlds of a host device (TEE-I/O) when sharing the monitored device traces serving as trustworthiness evidence to be securely forwarded to the *CONNECT* TAF. Based on this, D4.1 then puts forth a detailed description of the *CONNECT* overarching TEE architecture that will govern the secure operation of all deployed components. This will set the scene for both subsequent WP4 deliverables where the details mechanisms will be designed but also for WP5 where the orchestration of services to be instantiated on the MEC will be described. Apropriate security mechanisms for supporting one of the core visions of *CONNECT* towards **secure task offloading** will be designed in the context of WP4.

In continuation to these activities, D4.1 also documents the required behaviors/operations that need to be supported by the designed *CONNECT TCB* and *TEE-GSE*. These illustrate functionalities that will be used to provide trustworthiness evidence, for the enabling the continuous trust assessment activities (designed in WP3), and how this evidence is securely shared with other CCAM actors in a privacy-preserving manner. It is these set of functional specifications that will guide the subsequent activities of the protocol designs of WP4.

1.3 Open Challenges and Outlook

The focus of this Deliverable has been on the underpinnings of the *CONNECT* security architecture designed for supporting the in-vehicle operations and trust calculations. In D4.2, the consortium will be expanding this architecture to also capture the security controls considered when interacting with the CCAM services and *CONNECT*-related security services that are instantiated on the MEC.

1.4 Deliverable Structure

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

Chapter 2 gives an overview over the state-of-the-art for Trusted Computing. This includes concepts like "Trusted Computing Base", "Root of Trust", and "Trusted Execution Environments". In particular TEEs are a core hardware feature that is required by the *CONNECT* architecture. A Trusted Execution Environment (TEE) allows users to execute certain components with well-defined security guarantees that are enforced by the hardware, which acts as a Trusted Computing Base (TCB).

Chapter 3 provides a high-level specification of the *CONNECT* trusted execution architecture. We start by describing the *CONNECT* TEE Guard Security Extensions (TEE-GSE), the other devices, such as Electronic Control Units (ECUs), in the vehicle and the information flows between them. To complete this chapter we then focus on key management, an important part of ensuring that data in the vehicle (trustworthiness evidence, data from sensors and control signals to actuators) can be checked and verified. We also rationalize the TEE choices made in the *CONNECT* project.

Chapter 4 specifies important required behaviors of our architecture using so-called ”user stories”. 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 now detail the Intel SGX hardware Trusted Execution Environment (TEE) to be used in *CONNECT*. While Chapter2 surveyed the state of the art and gave a high-level overview over the Intel SGX hardware TEE, we now focus on the software stack to be used in *CONNECT*. A focus is the Gramine Library OS that allows seamless migration of Linux applications into a run-time environment that is protected by the Intel SGX TEE.

In **Chapter** 6 we extend the requirements outlined earlier by formalizing the requirements on our key management in a formal model that will later allow us to formally verify a subset of our security objectives.

Chapter 2

Introduction to Trusted Computing

Chapter 2 gives an overview over the state-of-the-art for one of the foundational pillars of the *CONNECT* architecture, namely "*Trusted Execution Environments (TEEs)*". TEEs provide the trusted computing capabilities necessary for verifying that calculations/operations (as part of a service graph chain) are performed by trustworthy systems. A TEE may also be able to verify other (remote) components and therefore help implementing the zero-trust principle of '*never trust, always verify*' prior to establishing a trust relationships between CCAM actors.

The goal is to use the Trusted Computing concepts to establish trust – layer by layer – into the complete architecture. This starts from the hardware built-in capabilities providing a TEE (as the underlying Root-of-Trust) and expands by verifying the software layers and essential (remote) services (see Section 3.2 describing the *CONNECT* *Trusted Computing Base (TCB)*). This stack can then produce evidence to *CONNECT*'s vehicle-wide trust assessment and quantification process that can be verified by other (external) stakeholders.

One core driving factor behind *CONNECT*'s overarching architecture (as described in D2.1 [13]) is to enable a *Connected, Co-operative and Automated Mobility (CCAM)* ecosystem where services are decomposed over the far edge, edge and cloud and still operate in a zero-trust manner [41]. The goal of such a zero-trust architectures is to minimize the required trust and permit stakeholders to validate that their security requirements are met. The concept of trusted computing allows the provision of strong guarantees towards the secure execution of selected safety-critical components, thus, guaranteeing and simplifying the trust relationships between all layers in the system run-time stack. This chapter introduces key concepts of trusted computing that we will later use when outlining the *CONNECT* architecture and surveys the state of the art.

2.1 Basic Concepts of Trusted Computing

2.1.1 What is Trusted Computing Base (TCB)?

End users of a secure system often rely on critical services of a system. Even more so, in automotive systems, availability and safety of the vehicle may depend on correctness and availability of certain services. To correctly provide such services, the architecture must ensure that certain critical components always behave as expected by the users or else fail safely. Because misbehavior of services that guarantee given requirements of the user often cannot be detected by the users themselves, this is called the *TCB* [40].

In general, the TCB may include communication, storage, and computation. Software developers may use encryption to protect *data in transit* and *data at rest*. For example, remote users can connect to the application server via secure network channels using SSL/TLS and VPN. All data travelling through these network channels are encrypted while in transit, such that a potential attacker cannot learn any secrets. As another example, software typically encrypts data before writing it to the hard disk, so that even if attackers gain access to private files, they cannot discern the stored secrets.

Formally, we define a **"*TCB* for a given requirement of a service or a function" as the set of hardware, firmware, services, and/or software components that are required to function correctly in order to guarantee that this requirement is met.**

It follows from this definition that different requirements on a service may require different TCBs. For instance, while the TCB for confidentiality of a real-time mapping service may include the components used for end-to-end encryption, the TCB for availability of the same service may include a hot-spare system that allows fail-over at run-time without interrupting the service.

Note that it is best practice to consider the TCB at design-time. TCBs can be designed to be static or extensible. A static TCB is designed to protect a fixed set of requirements and cannot extended. E.g. a *Trusted Platform Module (TPM)* chip provides fixed functions to allow integrity protection by means of attestation.

In *CONNECT* we aim to implement an extensible TCB that allows the TCB to be dynamically extended. We achieve this by using a modern TEE where new components can be added at run-time, are protected by the TEE, and as a consequence can extend the TCB.

2.1.2 Minimizing the TCB

The TCB can grow to be very large. It is a well-known empirical fact that the larger a component is, the more bugs lurk in it [34]. As a consequence, there is a higher risk of bugs and defects. Furthermore, by definition, a single bug in the TCB can compromise the given requirement that it protects. This constitutes a significant risk for critical services.

To mitigate this risks, designers aim to *minimize the TCB*. In particular for (security) critical services, designers usually try to minimize the size of the TCB. For instance, a small hardware security module for storing keys usually has a lower risk of compromise compared to a large cloud service providing the same function.

2.1.3 What is a Root of Trust (RoT)?

The (hardware) Root of Trust (RoT) is the minimal set of security guarantees – usually provided as built-in hardware capabilities – that is sufficient to protect a TCBs. The design goal is to ensure protection of a (usually software-based) TCB under the assumption that a well-defined set of security functions are provided by the (usually hardware) RoT.

Again, the RoT can be large, such as a whole micro-controller in a *Hardware Security Module (HSM)*, or can be smaller, like an encryption engine with loaded keys. One advantage of a hardware Root-of-Trust (RoT) is that it usually cannot be modified by its owner. As a consequence, it can provide a mechanism to build trust with a remote user; i.e., the remote user must gain trust that her application running in an untrusted environment is executed as expected and that the

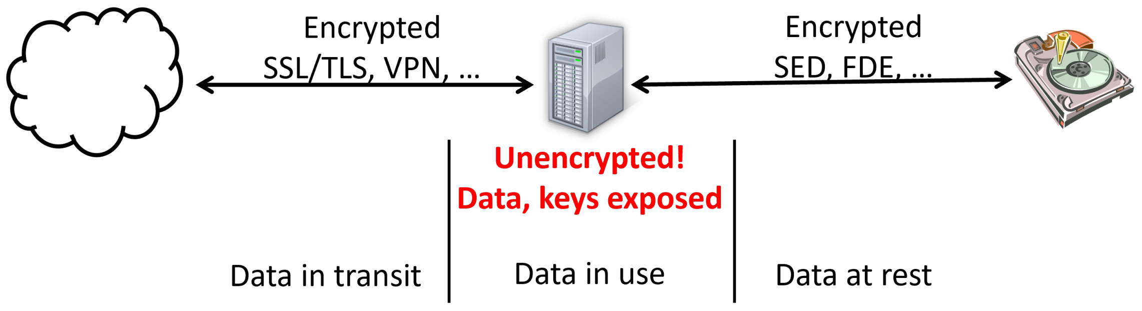

Figure 2.1: Compromised OS in an untrusted environment may leak sensitive data while it is in use.

application secrets are protected. To this end, hardware-based TEEs provide *remote attestation* capabilities.

2.1.4 What is a Trusted Execution Environment (TEE)?

A *Trusted Execution Environment (TEE)* provides a secure and isolated environment for performing critical activities, such as computations, that must be executed with a high degree of assurance. The concept based on which the TEE is built on, is the distinction between the "*trusted*" and the "*untrusted*" world of the host where the TEE is instantiated. By separating the two worlds, a TEE manages to create a safe environment that protects against unauthorised access as well as disclosure or tampering of confidential information. A Trusted Execution Environment 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. TEEs are one type of technology that can serve as a Root-of-Trust for supporting the secure execution of safety critical binaries.

One goal of a TEE is to remove the untrusted and large operating system from the *TCB*: Once the size of the TCB has been minimized and separates the software TCB from the underlying hardware RoT, the TCB often still includes the operating system and most of the hardware (e.g. memory and storage). To further enhance the security of the TCB, designers then *(a)* minimize the required trust into the hardware, and *(b)* further minimize the software that is required to be trusted by removing the operating system out of the TCB. *But why is it important to protect an application from a compromised operating system?*

Typically, software needs to manage various types of secrets, such as encryption keys, authentication credentials, financial information, and so on. These days, software developers offload their software to run on public clouds or in other untrusted environments. Second, a minimal TCB with only the trusted application and underlying hardware is enough to protect the application from a compromised operating system.

However, when data is used in the actual computation happening on the untrusted server (*data in use*), it is processed unencrypted within the CPU. In other words, data is completely in the clear as soon as it moves into the memory of the server. If the attacker controls the server – either by installing malware or simply by the virtue of having physical access to the server – the attacker can steal application secrets. Figure 2.1 illustrates this issue.

As such, *there is a need to protect and isolate sensitive application data in CPU memory during active computations*. For instance, the *CONNECT TCB* provides the necessary capabilities for the runtime monitoring of device characteristics (e.g., device configuration, integrity of the loaded software stack, control-flow integrity, etc.), constituting **security claims**, that need to be shared between entities that wish to establish a trust relationship. Validation properties may range from static properties such as integrity measurements of the host *CCAM* actor (e.g., vehicle), enabling the generation of static evidence of the vehicle's components correct configuration, to dynamic properties for verifying that *Actual Trust Level (ATL)* calculations are performed by a trustworthy software. All these security claims, serving as trustworthiness evidence for the vehicle-wide trust appraisal, need to be provided in a verifiable manner so as to not compromise the trust assessment process. Hence, as it will be described in Section 3.2, it is imperative to consider the secure execution of all processes required for the construction of such security claims to be protected by the SW/HW trusted computing co-design - the *CONNECT* *TEE Guard Security Extensions (TEE-GSE)* and *TCB*.

2.2 State of the Art on Trusted Computing

2.2.1 State of the Art on Hardware-based Roots-of-Trust

To protect a software TCB, well-defined hardware security guarantees are essential. This is where the term **trusted computing** has been introduced providing technologies and proposals for resolving computer security problems through hardware enhancements. Trusted Computing Group (TCG) is an industrial standards organisation that aims to create *TPM* specifications. Its main objective is to develop, define and promote open, vendor-neutral, global industry specifications and standards, supportive of a hardware-based RoT, for inter-operable trusted computing platforms. To that end, there is a need for trusted hardware. Examples of such hardware solutions include *TPM*, *TEE*, *Device Identifier Composition Engine (DICE)* (Device Identifier Composition Engine) and *Physically Unclonable Functions (PUFs)*, which will be further discussed in the following subsections.

In the *CONNECT* envisioned use cases (as documented in D2.1 [13]), these hardware devices can be used as a **RoT where we can store keys, ensure authentication, support attestations, and perform many other tasks in a secure manner**. In what follows, we will introduce these examples with more details while in Section 3.5 we elaborate on the selection of Grammine SGX-enabled technology as the trust anchor for instantiating all newly developed *CONNECT* secure life-cycle management controls. *However, all these protocols (that will be presented in detail in D4.2 [14]) are agnostic to the type of secure element used as long as the underlying RoT can provide the core properties of RoT for storage, reporting and measurement [13]. Grammine technology was selected due to its capability to allow for the transformation of a software binary to its trusted-equivalent, executing in an isolated environment, with no update requirements in the code and without affecting its inter-dependencies with other parts of the loaded software stack. Essentially, it allows the instantiation of any binary to be launched as part of an isolation environment called an enclave*. This is a rather new technology, developed under the umbrella of the well-established SGX TEE technology, where *CONNECT* envisions to provide concrete trust extensions for allowing for the secure communication between the "*trusted*" and "*untrusted*" worlds of a host device (TEE-I/O) when sharing the monitored device traces serving as trustworthiness evidence to be securely forwarded to the *CONNECT* Trust Assessment Framework. ***CONNECT* defines a TEE Device Interface Protocol (TDISP), extending the TDISP [25] as defined by**

the IETF RATS WG [1], for supporting secure TEE-I/O virtualization towards the provision of specific TEE Device Interfaces (TDIs) allowing the secure extraction of run-time system measurements depicting the trust state of the target device.

Trusted Platform Module (TPM): The TPM standard defines a hardware chip. When a TPM is embedded in its host platform, it serves as the Root of Trust for measurement and report. The TPM is also used as a cryptographic engine, which supports general cryptographic functionalities, such as digital signatures, encryption and message authentication. It also supports secure and flexible key management solutions, for example, it can implement light-weight multiple-layers key hierarchy with a very small amount of internal memory. With the key hierarchy architecture, the TPM can securely store cryptographic key material, which helps build security solutions in IoT. The nature of hardware-based cryptography ensures that the information stored in the TPM is better protected than software-preserved data. The TPM serves as a trust anchor for a host platform it is embedded in. It creates proof attestations about the state of the host system, e.g., certifying the boot sequence the host is running on. These attestations are called *Direct Anonymous Attestation (DAA)* ([7]; see Section 3.2.1). A *DAA* allows to sign attestation values anonymously while demonstrating that the TPM issuing the signature was part of a group of "valid TPMs". This is achieved by so-called *Group Signatures*.

Hardware Security Modules: The HSM is a hardware device that supports cryptographic processes such as key generation, sign and encryption, as well as key management and key storage within a computing environment. It may act as a RoT due to its cryptographic abilities and tamper-resistant environment. For the purposes of *CONNECT*, *electronic control units (ECUs)* with *HSMs* will be able to attest to the results of the secure boot process and measurement of the device's executables. However, these will be the only security guarantees that can be provided. The measured software, forming the *TCB*, will run in 'normal' un-protected memory and this will need to be taken into consideration when defining the trust model for the vehicle.

Device Identifier Composition Engine (DICE): DICE stands for Device Identifier Composition Engine. The engine is embedded in hardware in the first immutable boot loader that loads the first component off-chip. DICE is suited for embedded devices and is lightweight solution compared to TPM and Intel SGX. At the same time, the DICE is embedded within the same chip, thus, unlike TPMs allows for a threat model that include the chip boundary only. The Device Identifier Composition Engine is based on the first level of mutable code and *Unique Device Secret* that can be embed by the owner of the device (or the manufacturer). The assumption in the DICE is that the first mutable code is a boot level that does not change and thus, the engine will always deduce the keys based on *1)* Unique Device Secret UDS or *2)* hash of first boot level. DICE enables a solution that does not rely on strict hardware but allow family of hardware and software techniques that rooted back in hardware. The DICE using the hardware and software techniques provide Root of Trust for reporting (attestations), Root of Trust for storage (data encryption), and other deterministic seed for cryptographic functions. DICE security standard is created by the Trusted Computing Group (TCG) and supported by Microsoft's open-source project RIoT core [2], which acts as the first boot level security extension that does not change during runtime. DICE Architecture is a simple and new security approach suit for *Internet of Things (IoT)* and embedded devices that does not increase silicon requirements and provide the ability to owners to embed their own secret keys and own the entire attestation solution and identity of the device. The DICE

[1] https://datatracker.ietf.org/wg/rats/documents/

[2] "**RIoT** - A Foundation for Trust in Internet of Things", [Available Online]: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/06/RIoT20Paper-1.1-1.pdf, [Github]: https://github.com/microsoft/RIoT

architecture, with its hardware Root of Trust for measurement, breaks the software stack up into layers and creates unique secrets and a measure of integrity for each layer which means that any point in time, it can create comformity certificates of integrity for each software layer/process. Thus, DICE architecture allows a safe deployment and verification of software updates, which often are a source of malware and other attacks.

2.2.2 State of the Art on Trusted Execution Environments (TEEs)

A wide range of TEEs has been proposed for the automotive industry [31] - from simple HSMs like AUTOSAR SHE[3] to complete shielded and tamper-proof subsystems (e.g. the IBM4765 cryptographic co-processor). According to the *CONNECT* specification from D2.1 we aim to ”run *CONNECT* software within a hardware protected environment”. As such we focus on TEE functionalities provided by commercial off-the-shelf CPUs. The rationale is that we intend to pursue an integrated approach where non-security and security parts co-exist on the same hardware platform. TEEs typically consist of several components:

- **Secure Bootstrapping** to ensure the system starts at a secure initial state.
- **Secure Input/Output** to provide an end-to-end path for secure communication with network, storage, and other peripheral devices.
- **Isolated Execution** to protect data in use.
- **Remote Attestation** to prove the trustworthiness of TEE to a remote party.
- **Secure provisioning to obtain secrets from a remote party**.

ARM TrustZone

ARM TrustZone [4] is a feature that provides TEE support for ARM-based CPUs. It originally was launched as a feature of mobile devices in 2004 [33]. The core idea of TrustZone is to virtually partition the CPU into an untrusted world and a trusted world. The untrusted world then runs the commercial mobile OS (e.g. Android) while the trusted world (running a special secure OS) then only hosts security-critical services [4].

Intel SGX

Intel SGX [16] is a feature of Intel CPUs. Unlike TrustZone, it does not partition the complete CPU into trusted/untrusted but allows each user-space application to declare so-called ”enclaves” that protect a subset of an application. The CPU then provides dedicated protections of integrity and confidentiality of this enclave.

Intel SGX enables user-level code to allocate exclusive memory locations, known as enclaves, that are specifically designed to be protected from programs operating at higher privilege levels. Applications can communicate with the enclave by invoking a trusted function, which can

[3]https://www.autosar.org/fileadmin/standards/R21-11/FO/AUTOSAR_TR_SecureHardwareExtensions.pdf

[4]https://github.com/enovella/TEE-reversing provides a curated survey of TrustZone-related resources.

be run within a secure enclave. The enclave only allows authorized functions to execute, and any attempts to access enclave data are denied by the processor. Hardware-level encryption protects against software-based attacks, ensuring sensitive data remains undisclosed even if a hacker gains control over the operating system and BIOS. Enclaves may autonomously produce and store their own exclusive signing/attestation keys, thereby excluding any external party from accessing them. Data may exclusively be signed by utilizing keys that are linked to certain instruction sets, operating within each enclave.

A simple and illustrative example is a Browser that can partition-out its built-in password manager into a dedicated enclave. The CPU then ensures that the password manager is confidentiality and integrity protected. I.e. the memory and storage of the password manager is encrypted. The OS and applications cannot read the passwords, except the access provided by the API that is implemented inside the enclave. Furthermore, if an attacker then tries to launch a modified password manager to export the passwords, the CPU will refuse to launch it or will not allow this modified password manager to gain access to the encrypted passwords.

Note that one challenge when using SGX is that the *Operating System (OS)* still has strong influence over the enclave and thus writing code that remains secure even if attacked by the OS is challenging. Attack frameworks like SGXstep [39] allow the adversary outside the enclave to excercise fine-grained control which may result in leakage of secrets.

Securing Virtual Machines (ARM SEV, Intel TDX)

While Intel SGX protects a portion of a user-space application as an enclave within a TEE, the more recent proposals *AMD Secure Encrypted Virtualization (AMD SEV)* [1] and *Intel Trusted Domain Extensions (Intel TDX)* [27] aim at protecting complete virtual machines. They aim at a cloud environment where one server with a hypervisor runs multiple virtual machines. The goal is to protect the contents of each virtual machine against malicious behavior of other VMs and against the hypervisor. This is achieved in *AMD SEV* by *(a)* associating a key with each VM that is managed by the CPU and not exportable, *(b)* to use this key to encrypt the memory whenever it leaves the CPU, and (c) to encrypt all registers whenever the VM starts running. For *AMD SEV*, a few attacks have been published (e.g. [29]), *Intel TDX* has only been release in 2023.

2.2.3 Run-time Environments for TEEs

TEEs by itself are usually hardware features to support trusted execution. Since the hardware features of a CPU are often complex and difficult to use for application developers, a wide range of software frameworks have been proposed to simplify development, provisioning, and debugging of TEE applications. The open source community provides other frameworks that follow the Library OS approach – for example, the Gramine Library OS project [37, 38], Haven [6], SCONE [5], Occlum [35] or Panoply [36]. These frameworks put the whole application inside the SGX enclave. Only a small chunk of original logic stays untrusted, namely, input/output (I/O) functionality such as networking, file system, other system calls, and so on.

2.3 Intel Software Guard Extensions (Intel SGX)

Within *CONNECT* we will use the lightweight Grammine variant of Intel SGX TEE that is summarized in Section 2.3.1 and described in more detail in Section 5.1, where we will also provide details on the Gramine Library OS. The main reason is that the goal of the project is to perform research and rapid prototyping of technologies to be protected by a *TEE*. As a consequence, the main requirements we had were as follows:

Availability: The TEE should be widely available to all partners in the consortium.

Open: The TEE should be usable by any developer without additional (expensive) tools or restrictive licenses.

Tools: A wide range of tools should be available - ideally as open source.

Ease of use: Developing applications for this TEE should be easy and similar to developing traditional applications.

Extensibility: The TEE should support an extensible TCB, i.e. new software components can be added to the TCB at run-time to meet emerging security requirements.

No lock-in: The developed applications should not be entangled with the TEE and should be portable to other TEEs (if they are powerful enough).

Overall, we believe that the Intel SGX technology meets these requirements: It is widely available (in many off-the-shelf CPUs), supported by tools, open, and does not imply a vendor lock-in. Furthermore, the TCB is extensible since additional enclaves can be started any time and added to a running application.

Note that Intel SGX technology provides some CPU instructions and usually is not used without additional tools. It only provides the building blocks, but the actual SGX applications require the use of a software development framework that supports SGX. We already mentioned several such frameworks – Intel SGX SDK, Gramine, etc. In the following sections, we will concentrate on Gramine and its usage, as well as on the tools to run SGX applications in secure containers. The Gramine Library OS that is summarized summarized in Section 3.5.1 and described in more detail in Chapter 5.

2.3.1 A High-level Overview of Intel SGX

In *CONNECT* we aim at using the Intel SGX Trusted Execution Environment in an automotive context. The *Intel Software Guard Extensions (Intel SGX)* technology was introduced in 2015 with the Intel Skylake architecture [2, 18]. Intel SGX provides software developers the environment to secure their code and data even on untrusted and potentially compromised platforms. Developers can partition their code into CPU-hardened "enclaves" and place the security sensitive parts of their applications inside these SGX enclaves. Intel SGX-enabled CPUs protect selected code and data from disclosure or modification. Secrets that are used by the application are stored and operated on inside the enclave while the rest of the system cannot access this memory area.

Hardware-protected and isolated execution inside SGX enclaves provides a trusted execution environment at the application layer (*process-based TEE*). Intel SGX overcomes the shortcomings

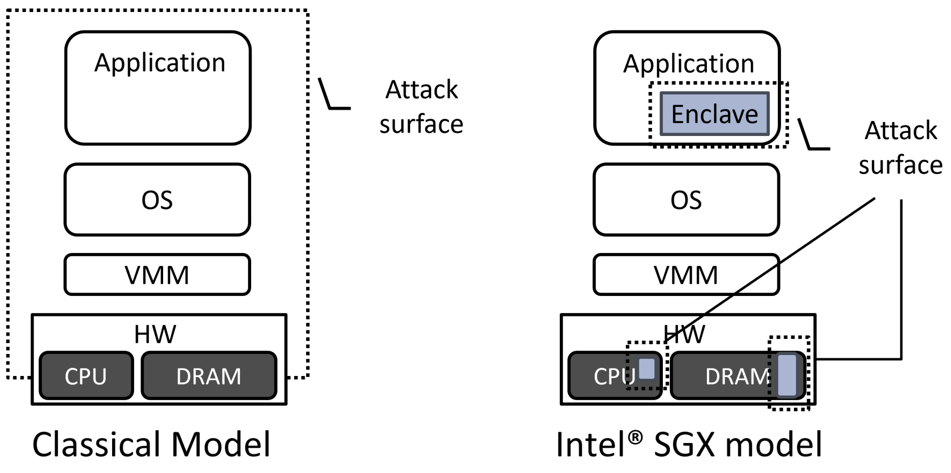

Figure 2.2: Classic application execution model vs Intel SGX protected execution model: In Intel SGX, the CPU provides an isolated Enclave while other parts no longer need to be trusted.

of the traditional protection model where the privileged software has full access to all data of any application. This is illustrated in Figure 2.2.

In a classical model, the application developer has to trust the whole hardware/software stack since she does not have control over the entirety of its components. As explained before, protection is provided by the *TCB*, which requires the developer to trust all of the underlying software and hardware. Any vulnerability of any component in the software/hardware stack becomes a vulnerability of the whole system, resulting in a system-wide attack surface.

With Intel SGX, the developer places sensitive code and data inside a so-called SGX enclave at the application layer. Code and data inside the enclave is protected by the CPU and inside main memory (DRAM). CPU limits access to the protected area to only the enclave code. No other software can gain access inside the enclave. Thus, the attack surface is reduced to the enclave code/data and Intel SGX hardware.

Intel provides the Intel SGX Software Development Kit (SDK) – a framework that allows to write manually-partitioned applications for SGX [17]. However, a significant challenge when programming with Intel SGX *Software Development Kit (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.

2.3.2 A History of Trusted Execution Environments at Intel

The history of confidential computing solutions at Intel is rich and long. First, Intel introduced the Intel VT-x (also called simply Intel virtualization) technology around 2005, that introduced the “virtual execution mode” to Intel processors, which could be entered and exited using special new instructions. This allowed to create hardware-assisted virtual machines, or VMs, that could run isolated from each other and from the virtual machine monitor (or VMM, also called hypervisor).

Around the same time, Intel introduced a technology called Intel Trusted Execution Technology

(Intel TXT). This technology uses a Trusted Platform Module (TPM) to measure the code and initial state of the computer and to build a chain of trust during boot. In the context of confidential computing and virtualization, Intel TXT allows to gain trust in the VMM, since it is hardware-measured by the CPU and is thus guaranteed to be a known good VMM implementation in a known good initial state.

But these technologies provide no confidentiality guarantees. To help improve confidentiality of customer workloads, in 2021 Intel presented Total Memory Encryption (Intel TME for short) that allows to encrypt all memory on the platform, to prevent software attacks as well as hardware physical attacks on RAM. A refinement of this technology called Multi-Key TME (MK-TME) allows to encrypt not just with one encryption key, but with multiple encryption keys; thus it becomes possible to encrypt each VM with its own key, such that each VM becomes completely opaque to any other collocated VM.

In the meantime, Intel introduced Software Guard Extensions (Intel SGX) around 2014. Intel SGX is unique in this list of technologies because it does **not** use virtualization technologies. Instead, Intel SGX is able to create opaque “enclaves” on the platform, and enter and exit these SGX enclaves using special new instructions. Intel SGX was the first confidential computing technology from Intel that allowed to completely remove all software from the *TCB*, including the Operating System, *Virtual Machine Monitor (VMM)*, and any other software, privileged or unprivileged. Intel SGX is sufficiently easy to deploy: it requires only a driver in the OS, and a software framework to run applications inside SGX enclaves: either an SDK, that requires re-writing and re-factoring applications to use SGX-specific SDK APIs, or a Library OS, that requires no modification or recompilation of the app.

Finally, currently Intel is focusing on a new technology called Intel Trust Domain Execution (or Intel TDX). Intel TDX is similar to Intel SGX in that it’s a full fledged Confidential Computing solution – it creates opaque “Trust Domains” (basically, encrypted and integrity-protected VMs) that are not accessible even by the privileged Cloud Service Provider software, such as OS or VMM. In contrast to Intel SGX, Intel TDX is based on the virtualization technology and thus operates on a more familiar level of complete virtual machines (rather than single applications or parts of applications). In contrast to Intel MK-TME, Intel TDX not only encrypts the VM but also stops the hypervisor from accessing the VM. The enablement of Intel TDX is a bit more complicated than that of Intel SGX: both the VMM and the guest Operating System require some modifications; on the other hand, no modifications to the applications that run inside the TD are required.

Chapter 3

High-Level Overview over the *CONNECT* Trusted Execution Architecture

After having described the core building block based on which the entire *CONNECT* framework is built upon – *integration of a Trusted Execution Environment for safeguarding all operations supporting the continuous trust assessment of any CCAM actor and/or data object*, in what follows we delve into the architectural details of *CONNECT*'s trust extensions. These essentially cover all secure life-cycle management protocols and interfaces (that will be described in detail in the context of D4.2 [14]): from the **secure on-boarding and enrollment** of all CCAM applications/services, instantiated in the vehicle and/or *Mobile Edge Cloud (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). All these operations are supported by *CONNECT*'s two main trusted computing pillars:

- ***CONNECT TCB*:** This constitutes the palette of security mechanisms and ancillary 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. Part of *CONNECT*'s *TCB* (as will be detailed in Section 3.2) comprises the: (i) **tracing capabilities**, based on the newly developed TEE Device Interfaces (TDIs), to be exposed for capturing the continuous extraction of an extended set of device characteristics (as required by the *Trust Assessment Framework (TAF)*) in a verifiable manner, (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 on-boarding of a device (Section 3.3), and (iii) **key usage restriction policy engine** for checking the validity of deployed policies protecting the use of the created (signing) keys only when the state of a device is at an expected state. The *TCB* must always behave as expected, otherwise, there is a risk for the entire trust assessment process to get exploited. Therefore, the entire *TCB* software stack configuration is monitored, stored and continuously verified by the underlying *Root of Trust (RoT)* so as to make sure that it is not altered. As described in Section 2.1.1, the *TCB* is minimized and constitutes the trust anchor for all applications/services that are instantiated in secure environments and for which we want to guarantee high operational assurance.

- ***CONNECT* TEE Guard Extensions (*TEE-GSE*):** These comprise the secure containers offering the necessary functionality of the secure life-cycle management processes. The are built on top of the *TCB* (each container will be equipped with its own *CONNECT TCB*) for continuously interacting with the trust anchor so as to then be able to provide the necessary evidence to all other requesting components for been able to perform any of the operations described in the next section.

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 [13].*

3.1 *CONNECT* High-Level Security Architecture Including TEE Guard Extensions

The *TEE Guard Security Extensions* (*TEE-GSE*) are comprised by a set of *CONNECT* system components. These components play a crucial role since they provide the necessary run-time security guarantees which ensure the correct usage of all cryptographic primitives, and consequently facilitate the provision of trustworthiness evidence in a verifiable manner. These evidence may be consumed either internally (i.e., within the boundaries of the vehicle) by the *TAF*, among other information, to derive to the Actual Trust Level (ATL), or they could be sent in the form of *Verifiable Presentation (VP)s* outside the vehicle (i.e., to other vehicles or the infrastructure), enabling the *CCAM* vision as introduced in D2.1 [13].

The need for this continuous and real-time extraction of evidence arises from the dynamic nature of trust states, associated with both nodes and data. In a dynamic environment, where trustworthiness can evolve rapidly, real-time evidence extraction becomes crucial to adapt to changing conditions and promptly assess, thus respond to potential security threats or alterations in the trust landscape. The continuous evidence extraction process ensures that the trust states of nodes and data are properly reflected, enabling the system to maintain an up-to-date and accurate understanding of the trust levels across the *CONNECT* ecosystem. Delving into the contents of these evidence, trust properties may be included such as integrity, robustness, availability, security, etc. More information regarding the trust properties is available at D3.1 [11].

The *CONNECT TEE-GSE* is built on top of the underlying *RoT*, while as a framework it remains agnostic to the employed trusted computing technology. Overall, *CONNECT* aligns with the zero-trust principle, which is based on the notion that no entity can be inherently trusted, even when it operates within the vehicle. Hence, all nodes should be able to provide evidence regarding their trustworthiness. This notion further applies to the MEC side, enabling the Chip-to-Cloud vision of *CONNECT* by providing strong security enablers across the entirety of the *CCAM* ecosystem from the vehicle (far edge) to the *MEC* and Cloud. In the present deliverable, we focus the descriptions on CONNECT's security and trust extensions deployed at the vehicle side. More details on the *MEC* configuration will be presented in D4.2 [14] - although the same trusted computing principles apply as for the services deployed within a Vehicle with the core differentiating factor been on the type of trustworthiness evidence monitored and used for the trust assessment of the two types of operational environments. The following paragraphs analyse the functionalities of the *TEE-GSE* components (refer also to Figure 3.1).

Towards this direction of continuous *CCAM* trust assessment, the *TEE-GSE* is proposed by *CONNECT* as a core building block, offering the necessary security enablers. In *CONNECT* architec-

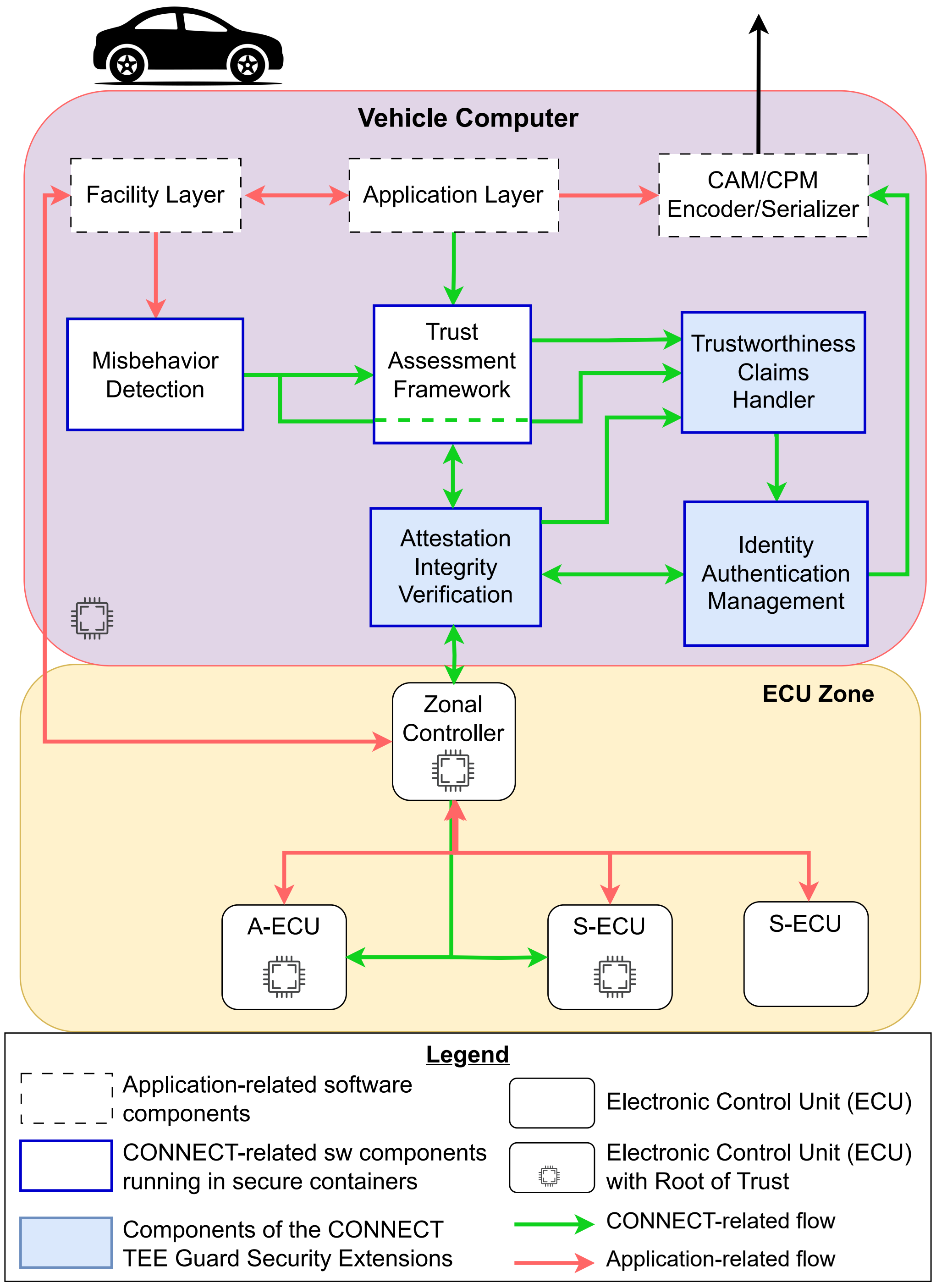
Vehicle Computer
Facility Layer
Application Layer
CAM/CPM Encoder/Serializer
Misbehavior Detection
Trust Assessment Framework
Trustworthiness Claims Handler
Attestation Integrity Verification
Identity Authentication Management
ECU Zone
Zonal Controller
A-ECU
S-ECU
S-ECU
Legend
Application-related software components
CONNECT-related sw components running in secure containers
Components of the CONNECT TEE Guard Security Extensions
Electronic Control Unit (ECU)
Electronic Control Unit (ECU) with Root of Trust
CONNECT-related flow
Application-related flow

Figure 3.1: *CONNECT* Vehicle Security Architecture.

ture, all components that execute trust-related tasks are build on top of the *TCB*; thus, are able to attest their own state (i.e., through key restriction usage policies). Nevertheless, the ones that are considered as part of the *TEE-GSE* further provide support in terms of verification of attestation evidence of other components. More information regarding the exact functionality supported by each of the *TEE-GSE* components, is provided in the following paragraphs.

- ***Identity and Authentication Management (IAM)***: The *IAM* handles the secure on-boarding of the devices (i.e., including *ECUs* and Zonal Controllers) to the in-vehicle Computer. To successfully perform this task the IAM leverages nominal reference values, which are acquired by the *Original Equipment manufacturer (OEM)* and reflect the expected configuration of the *ECU*. After the validation of the device's state, the *IAM* establishes the cryptographic keys and key restriction usage policies, acting as a key manager for the components of the vehicle. It manages firmware and software updates as well as component replacements, while it further maintains a list of configuration information (i.e., serial numbers, software versions, attestation reference values, etc.). This configuration information is stored by the *IAM* as a *Verifiable Credential (VC)* and is shared with the *Attestation and Integrity Verification (AIV)* for (run-time) verification of attestation evidence. The *VC* is also sent to the *Trustworthiness Claims Handler (TCH)* to be used for disclosing design-time integrity attributes, when needed as part of a trust assessment request, for all devices comprising a service graph chain. For instance, whether secure boot mechanism is available for all in-vehicle sensors which was executed correctly during the on-boarding phase of an ECU. The IAM component further manages the V2X communication key (i.e., PKI-issued pseudonym credentials as defined by ETSI) for the secure and privacy-reserving communication with other vehicles or the MEC, while it constructs the VPs containing the harmonised attributes (i.e., anonymised attestation reports), the *TAF* and the *Mis-behaviour Detector (MBD)* reports.

- **Attestation and Integrity Verification (*AIV*)**: The *AIV* manages the attestation of the devices within the vehicle, including both asymmetric (*A-ECUs*) and symmetric capable (*S-ECU*). A key distinction between the two lies in their attestation capabilities. *A-ECUs* are capable of supporting local attestation, while *S-ECUs* requires the *AIV* to act as the Verifier, due to limited resources. Therefore, in the local attestation case, the *A-ECU* provides a signature leveraging the key restriction usage policies, and if the device's state is correct, then it is able to share the signature with the *AIV* for verification. In the remote attestation approach, the *S-ECUs* transmits a quote (i.e., attestation evidence) which is verified by the *AIV*, against a reference value that is stored by the *IAM*. The process of evidence collection, which includes attestation evidence (i.e., traces) is instantiated upon the receipt of a Request for Evidence (RFE) message by the *TAF*. This evidence is consumed both by the *TAF* and the *TCH*. The *TAF* leverages the attestation report to calculate the ATL, while the TCH utilises the attestation report for the generation of the harmonised attributes, which are later disseminated in the form of a VP to other vehicles in the vicinity of the MEC. Upon the occurrence of attestation failures, the *AIV* transmits the encrypted raw evidence to the *Distributed Ledger Technology (DLT)* for subsequent scrutiny by the relevant stakeholders, such as *OEM* or regulatory authorities. 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.

- **Trustworthiness Claims Handler (*TCH*)**: The *TCH* is responsible for the attribute harmonisation/obfuscation of of all attestation evidence received by the AIV so as to be grouped together in trust-centric properties that enable the vehicle-wide trust assessment without breaching the privacy of each individual device; e.g., sharing of detailed attestation information per ECU can lead to vehicle fingerprinting which, in turn, can allow an attacker to target known vulnerabilities of the software stack loaded to the ECU. In essence, the harmonisation represents the same group of evidence (i.e., group-based evidence from multiple *ECUs*), that can be used by a receiving entity to assess trust levels for another entity, in a zero knowledge manner. The *TCH* further receives, thus verifies, the signed

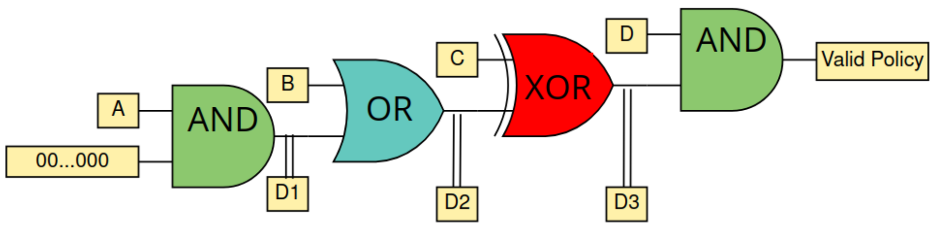

Figure 3.2: Key Restriction Usage Policy Engine.

reports (i.e., in the form of *VCs*) from the *TAF* and the *MBD*, which are combined with the harmonised attributes and shared with the *IAM*. The output of the TCH results in the generation of the Verifiable Presentation (*VP*), signed with the TCH key and to assert its correct computation. This VP encapsulates the Trustworthiness Claims (TCs) introduced by CONNECT [12]. and subsequently, it is incorporated into the T-CAM and T-CPM messages, along with the kinematic information. This inclusion of the VP in CAM and CPM messages ensures the dissemination of trust-related information within the CCAM landscape. The *TCH*'s role is equivalent on the *MEC* side, with the difference that the harmonisation of the *AIV*'s output is not needed to provide anonymity. Consequently, the trustworthiness claims (i.e., including the attestation report, the *TAF* report and the *MBD* report) are signed by the *MEC*-*TCH* and sent to the vehicles which can obtain, directly, the correct state of the infrastructure hosting the MEC services.

3.2 *CONNECT* Trusted Computing Base & Building Blocks

The *CONNECT TCB* serves as the foundation for the execution of critical tasks as it pertains to security and trust assessment. While functionalities related to the verification of attestation evidence, acquired by the *ECUs*, are considered as part of the *TEE-GSE*, other equally critical operations are supported by the *CONNECT TCB*.

The following paragraphs delve into the exact building blocks that form the *CONNECT TCB* along with their functionalities. It is worth mentioning that the *CONNECT TCB* may refer both to the vehicle computer and the *ECUs*. However for the case of the *ECUs*, there is a distinction between different *ECUs* (i.e., *A-ECUs*, *S-ECUs*) based on their cryptographic capabilities. The following descriptions are mainly focusing on the *A-ECUs* that have the capacity to support such operations.

- **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 Usage Restriction Policy Engine:** Enhanced key authorization mechanisms need to be adopted by the *CONNECT* TCB. More specifically, 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 labeled as ”completed”. These actions refer to the usage of an attestation key and are called Key Restriction Usage Policies. Within this context, assertions play a pivotal role, representing statements that must be true for a policy to be satisfied. The Key Restriction Usage Policy Engine (KRPE), a proposed TCB component in *CONNECT*, supports a collection of logical equations constructed from these assertions. Figure 3.2 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 authorization, and the KRPE itself.

- **(Attestation) Tracer:** Another core component of *CONNECT*'s TCB is the Tracer (Figure 3.3), which is responsible for continuously monitoring processes and routines that are executed in 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. These security measurements are signed by the Tracer in the trusted world and are sent to the Key Manager to perform the required operations. 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. In the trusted part, 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, are taking place.

  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 TDISP (Trusted Device Interface for Security Protocols) protocol. The Attestation Agent's role in providing authentic traces and ensuring the secure communication of these measurements is fundamental to the overall security and trustworthiness of the system.

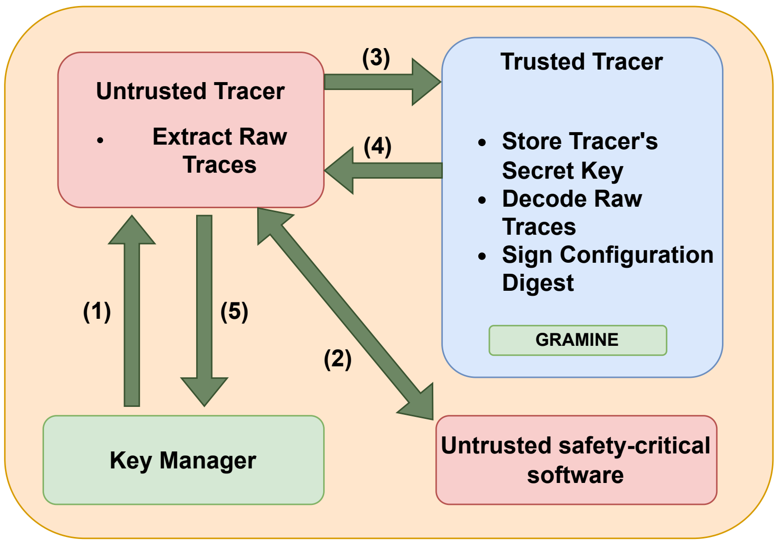

Figure 3.3: *CONNECT*'s Attestation Tracer (i.e., trusted vs untrusted world).

3.2.1 *CONNECT* Attestation Enablers

The zero-trust principle is a core characteristic of the *CONNECT* project and all critical components within the framework need to be able to provide continuous verification mechanisms that enable the validation of various attributes of an entity. This section focuses on the attestation capabilities that are tailored to the security requirements where a prover may want to validate not only the correct state of a node in the topology but also the correct execution of the functionality in real-time, where possible. Given the complex and diverse environment of the *CONNECT* ecosystem, the attestation capabilities of one powerful device — e. g., a TEE-enabled Zonal Controller device — might be completely different compared to the ones of a resource-limited edge device - e.g. a S-ECU. In parallel, depending on the criticality of a CCAM functionality, a prover entity (i.e., the AIV component) may request to take advantage of only a subset of the available attestation capabilities of a device. The higher the required amount of trustworthiness required for a device to participate in a function, the more attestation evidence need to be collected - both in quantity and diversity - by a prover entity to gain a deeper level of insight for the device under investigation.

Static Attestation capabilities

The common attestation scheme for all *CONNECT* devices will support is static attestation. Static Attestation is well defined in the academic literature and is used in many Remote Attestation schemes [24] as it provides a straightforward verification over the integrity of the attested object. Typically the algorithm computes a hash digest of the device's memory in order to create a fingerprint of the entire software stack of the device and stores it in a trusted register. The main flaw of such a static attestation algorithm is that it does not apply to dynamic objects of the attested software stack. Using the hash digest of such dynamic entities on a device is ineffective , because they change in unpredictable ways, making it impossible for a verifier to keep track of their states. For example, many applications use dynamic memory allocation and library invocations, creating code spaces that cannot be mapped statically and can create a perception to the verifier of malicious behavior even if the application is running correctly. To handle such vulnerabilities, *CONNECT* plans to leverage run-time attestation, which will be explained in the next section.

Configuration Integrity Verification (CIV)

There are many proposals in the literature that focus on verifying the correct configuration of a platform with related specifications providing the foundational concepts such as measured boot and remote attestation. However, many of the existing families of attestation solutions have strong assumptions on the verifying entity's trustworthiness, thus not allowing for privacy-preserving integrity correctness. Furthermore, they suffer from scalability and efficiency issues. It should be difficult for any (possibly compromised) Verifier to infer any meaningful information on the state or configuration of any of the devices comprising the service graph chain or wishing to enroll to a network. In this context, it is essential to ensure not only the security of the underlying host and loaded software processes but also their privacy and confidentiality - an attacker should not be able to infer any information on the configuration of any of the binaries loaded in the same node. This is identified through an oblivious challenge where an entity - such as the *IAM* component - needs to attest to all of its components in a zero-touch manner without the need to reveal specific configuration details of its software stack [21, 22]. This is a prerequisite to allow the *IAM* to successfully perform the secure on-boarding of an ECU device. The trust assumptions and the security requirements that enable the CIV to be launched for a particular *ECU* device are presented in tables 6.6 and 6.7 respectively. The complete process of the secure enrollment of an *ECU* is captured in the User Story V.

Control-Flow Attestation (CFA)

A core limitation of CIV-based solutions is that they do not ensure the integrity of the software's execution during run-time and, therefore, cannot capture attacks that target the program's control flow. These types of attacks are considered the most devastating since they try to exploit memory-related and data related vulnerabilities for altering the execution path of the underlying system processes. Such vulnerabilities are able to bypass the security offered by static attestation techniques since the measurement of a binary can remain unchanged even though the software's behavior has been altered. To address this issue, advanced dynamic control-flow attestation solutions propose capturing the software throughout its execution in order to identify whether it is running as expected by verifying the integrity of the entire control flow [19, 32]. To address scalability and performance issues we should carefully identify the parts of the software base that are essential to be attested to. We do not need the attestation of the entire application or device but only the execution properties of the security sensitive functionalities running. The identification of which functionalities should be monitored is implementation dependent and depicted in the deployed attestation policies. The aim of this procedure is to check both behavioral properties and low-level concrete properties about the entity's configuration and execution, such as the current firmware version it is running, the version of its configuration file or presence of certain hardware properties, integrity of sensor measurements, execution paths to specific memory regions, ports and network interfaces, etc. To enable a run-time attestation process like CFA, Tables 5.7 and 5.8 summarize the set of assumptions and requirements that are needed with respect to the characteristics of the tracer software that is responsible for collecting the run-time traces of the application under investigation. User story VII describes the need for collecting (run-time) attestation evidence from the vehicle devices.

Verifiable Policy Enforcer (VPE)

As aforementioned, the Identity Authentication Management component handles the Secure Onboarding of all the devices of the in-vehicle topology, setting the appropriate key restriction usage policies for each TEE-enabled device. In *CONNECT*, we differentiate the between two types of key restriction usage policies: *static* and *dynamic*.

- **Static policies:** These cover the design space, encompassing system properties that typically remain unchanged over time. Examples of such properties are secure boot identity, firmware version, and the configuration of the software that instantiates the *CONNECT*'s TCB. Frequent modifications in these properties could result in a modification of the device's state, hence safeguarding against frequent alterations is critical. *CONNECT* opts to allocate these properties to the constant memory space of the device's TCB. This placement ensures that static policies cannot be changed without disrupting the normal operation of the underlying TCB, mandating a device reboot to implement state changes.
- **Dynamic policies:** These capture the full spectrum of the design space, and include system properties that may change frequently during the operational lifecycle of the device. These policies depict the level of trust in the correctness of the operational state of a device and can be attested utilising various attestation enablers, such as Configuration Integrity Verification (CIV) and Control-Flow Attestation (as mentioned in subsections 3.2.1, and 3.2.1).

In the context of *CONNECT*, this distinction is crucial to provide trusted software updates. When a device undergoes a software update, it necessitates a corresponding key restriction usage policy update. This introduces challenges regarding the auditability of the updated device, as we need to ensure that an old policy is no longer in use. To address these concerns, *CONNECT* introduces a new mechanism referred to as **Verifiable Policy Enforcer (VPE)**. The VPE consists of a key bound with the issued static policy and a set of assertions, that will attest the software stack of the TCB that is running in the untrusted/normal world. Operating exclusively within the Trusted world, the VPE assumes the critical role of auditing the key restriction usage policy bound to the attestation key of the device. Notably, all attestation keys are bound with dynamically issued policies, representing the updatable software stack.

Swarm Attestation and Composable Attestation

As an alternative to the attestation of a single device, *CONNECT* also considers the parallel attestation of a swarm of devices featuring the same processes to be verified. The ultimate goal of the swarm attestation schemes is to provide scalable solutions that verify the trustworthiness of a large-scale network in a more efficient way than attesting devices individually. The existing swarm remote attestation protocols differ from each other from various design parameters such as network topology, adversary model, attested memory regions, verification of exchanged communication data, number of verifiers, etc. For instance, consider the case of the *AIV* wanting to attest all of the A-ECU devices within the same vehicle coming from a single *OEM*. Instead of initiating remote attestation processes with each one of the deployed edge devices, a single process can be triggered for attesting the same properties of all devices in a privacy-preserving manner.

One important limitation of swarm attestation techniques is that they focus on attesting to a particular trust property within a designated network. However, depending on the request for evidence sent by the *TAF* to the *AIV* component, it is possible that multiple trust properties are required to be assessed for a set of devices. This leads to the exploration of Composable Attestation schemes, an important aspect of *CONNECT* which enable the collection of multiple attestation results for subsets of a swarm of devices in an efficient and scalable manner.

Direct Anonymous Attestation (DAA)

As mentioned in In *CONNECT*deliverable D5.1, CONNECT envisions to use a newly developed variant of Direct Anonymous Attestation (DAA) scheme [28, 3], which scopes to enhance the privacy guarantees of complex ecosystems such as *CONNECT* 's. The goal of the DAA scheme is to shift trust assurance from the infrastructure to each vehicle, thus creating a decentralized approach in the trustworthiness establishment process, in a privacy preserving manner. To this end, DAA aims to provide authentication of data in terms of their origin, meaning that a vehicle's component should be able to provide verifiable evidence that it represents a valid and enrolled computer, while simultaneously ensuring that this evidence cannot be linked to the vehicle's privacy critical information, such as vehicle's location or other attributes that can lead to the vehicles fingerprinting. Due to the privacy preserving nature of the DAA, we consider it as a suitable candidate for Story-X, as the *TCH* wishes to create an anonymous signature over the harmonized attributes of the vehicle.

Link Tokens

Expanding our research for suitable attestation schemes for *CONNECT*, another important requirement that was considered was the **interplay between the provision of verified attestation results in a zero-knowledge manner vs. the possibility to trace back to the origin of a failed attestation result.** This is rather important as a failed attestation process in an indication of risk which might require the deployment and enforcement of the appropriate set of mitigation strategies. For instance, in *CONNECT*, the detailed evidence that led to a failed attestation result are securely recorded on the Blockchain infrastructure so that the OEM/Security Administrator can further process them and identify the exact point of failure. In the case of zero-day vulnerabilities this, will allow the CONNECT Risk Assessment engine to produce a new risk graph for the entire CCAM continuum and provide an updated version of the graph (with the risk quantification of all identified vulnerabilities) to the CONNECT Trust Management Framework for calculating the new Required Trust Level (RTL) that needs to be exhibited by an entity to be characterized as trustworthy (see D2.1 [13] for more details). This RTL will be deployed (together with a possible update of the respective trust model includng the type of attestation evidence to be monitored) to all Trust Assessment agents against which the current trust level will be compared.

In this context, *CONNECT* will add a traceability requirement to the DAA scheme to be leveraged [8] to obtain a novel Traceable DAA protocol based on the use of link tokens for the internal Schnorr signatures. More specifically, Schnorr signatures provide a secure and efficient way of proving possession of certain information without revealing the actual data. As will be detailed in Stories XI and XII, *CONNECT* 's components *AIV*, *TAF* and *MBD* will be providing their output (Trust Opinion and Misbehavior Report, respectively) through (anonymized) Verifiable Credential (*VCs*) structures, in order for both the *TCH* but also a CCAM message recipient to be able to verify the included attributes without been able to extract any information about the identity of the signature provided. In this context, Schnorr signatures can be used to attest to the validity of the credential without revealing the actual details. The *CONNECT* components could sign a message containing the relevant information with their private key, which of course is bound with a key restriction usage policy dictating the correct configuration of this component and others could verify the signature, not by using the associated public key, but by using bi-linear pairings over the public key of the credential issuer, which is a Trusted Third Party.

In order to be able to trace back to the identity of an *AIV* component, that possibly produced failed attestation results (based on the evidence it received from one of the in-vehicle sensors), the signatures will also be linked to a *tracing key* (of an authenticated entity (e.g., OEM) that is authorized to trace back a failed attestation result) that allows the de-anonymization of the signature. This will allow the tracing back of a VC to its origin (i.e., AIV) which will then be able to map to the in-vehicle sensor that provided the evidence that produced a failed attestation result.

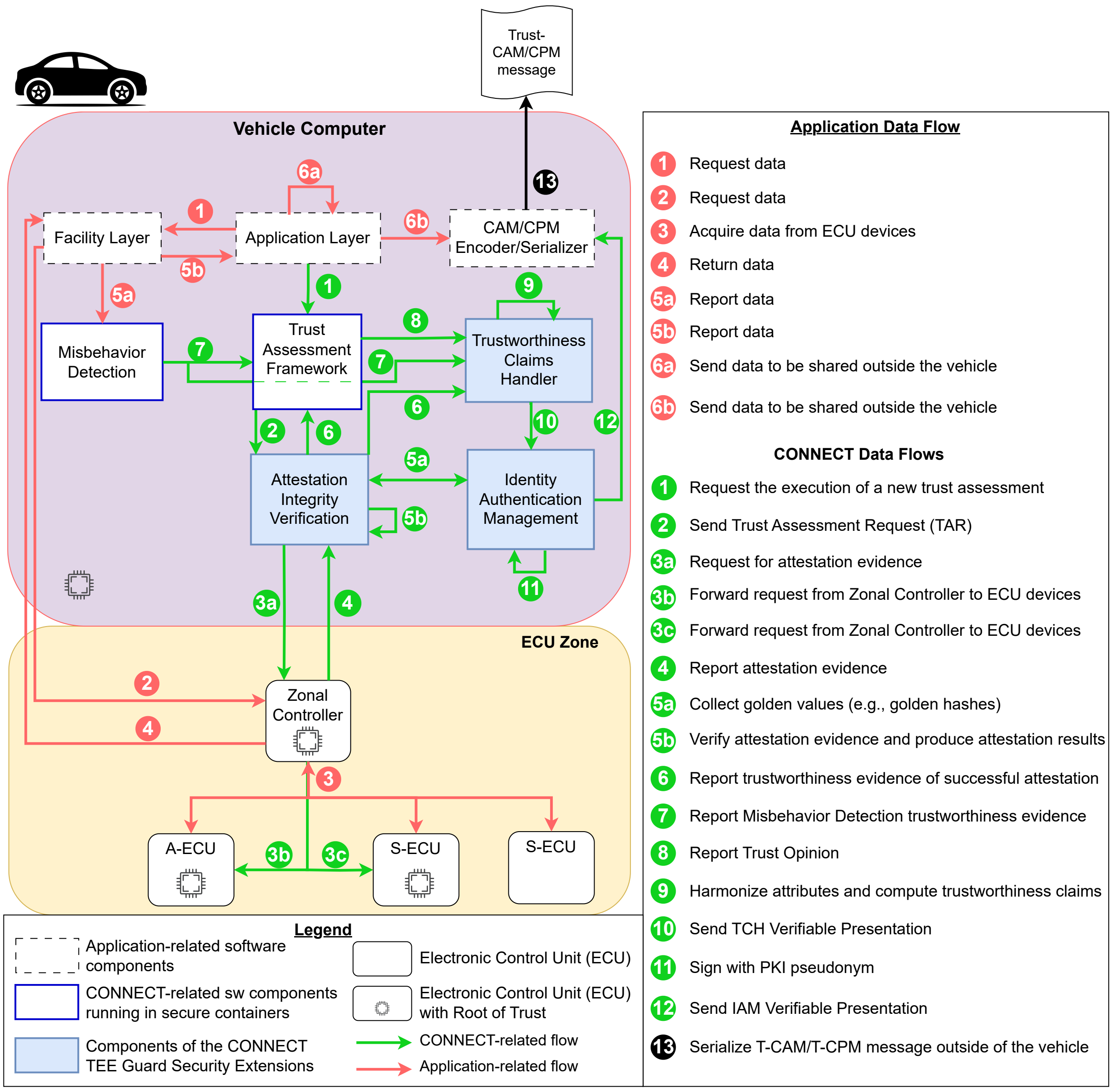
Trust-CAM/CPM message
Vehicle Computer
Facility Layer
Application Layer
CAM/CPM Encoder/Serializer
Misbehavior Detection
Trust Assessment Framework
Trustworthiness Claims Handler
Attestation Integrity Verification
Identity Authentication Management
ECU Zone
Zonal Controller
A-ECU
S-ECU
S-ECU
Legend
Application-related software components
CONNECT-related sw components running in secure containers
Components of the CONNECT TEE Guard Security Extensions
Electronic Control Unit (ECU)
Electronic Control Unit (ECU) with Root of Trust
CONNECT-related flow
Application-related flow
Application Data Flow
1 Request data
2 Request data
3 Acquire data from ECU devices
4 Return data
5a Report data
5b Report data
6a Send data to be shared outside the vehicle
6b Send data to be shared outside the vehicle
CONNECT Data Flows
1 Request the execution of a new trust assessment
2 Send Trust Assessment Request (TAR)
3a Request for attestation evidence
3b Forward request from Zonal Controller to ECU devices
3c Forward request from Zonal Controller to ECU devices
4 Report attestation evidence
5a Collect golden values (e.g., golden hashes)
5b Verify attestation evidence and produce attestation results
6 Report trustworthiness evidence of successful attestation
7 Report Misbehavior Detection trustworthiness evidence
8 Report Trust Opinion
9 Harmonize attributes and compute trustworthiness claims
10 Send TCH Verifiable Presentation
11 Sign with PKI pseudonym
12 Send IAM Verifiable Presentation
13 Serialize T-CAM/T-CPM message outside of the vehicle

Figure 3.4: *CONNECT* Vehicle Information Flows.

3.3 *CONNECT* Key Management

The architecture of *CONNECT* is depicted in Figure 3.4, providing a comprehensive overview of the components within the vehicle, differentiating the in-vehicle computer with the ECUs, while illustrating various information flows. This section delves into the dedicated functions of each component within the vehicle, emphasizing their role in generating attestation evidence. A detailed exploration of the information flows is presented in Section 3.4.

The vehicle computer is in essence a system of systems. The vehicle computer in the *CONNECT* framework possesses its dedicated *RoT* and *TCB*, enabling it to conduct local attestation. While the in-vehicle computer supports local attestation, all software components used *CONNECT* for trustworthiness assessments further provide their own attestation evidence (i.e., TAF, MD). It shall be noted that the *CONNECT* *TCB* components performing trust-related tasks, are all executed within secure containers.

For the *ECUs* in the vehicle there needs be verifiable evidence that they are correctly configured, that they are using the appropriate keys and that the software being used has not been tampered with. The *IAM* manages the configuration and use of keys, while the devices themselves need to provide attestation evidence to confirm the status of their software and where possible to provide run-time attestation evidence as well.

This section builds on the description in Deliverable D2.1 of the keys that will be used in a *CONNECT* vehicle. While giving more information about the keys, details of the protocols that will be used will be given in Deliverable D4.2. First of all there are some general requirements:

- All processing should be as efficient as possible and meet the timing constraints of the applications being supported.
- The source and processing of data should be guaranteed – it should be possible to know that the communication of data was handled by trustworthy entities in the system and further that any processing was carried out by authorised entities.
- All communications between entities (devices, containers and applications) should be integrity checked and where appropriate maintain confidentiality.
- All communications outside of the vehicle should protect the privacy of that vehicle.

The keys used in the *CONNECT* system are essential to enable these requirements to be met.

3.3.1 ECU keys

As mentioned in the previous section, the ECUs provide verifiable evidence regarding the correctness of their configuration. Nevertheless, the different types of *ECU* supported by *CONNECT* have different cryptographic capabilities. Note that despite the diverge capabilities, ideally all types of ECUs should possess a *RoT* providing the basic functionalities listed in Deliverable 2.1, Section 8.1.5. Hence, all ECUs within the vehicle will support a *TCB* and maintain their dedicated local key managers, responsible for securely storing the necessary cryptographic keys. These keys managers, where applicable implement key restriction policies to govern the use of keys. Note that *N-ECUs* have no security features at all. In these (limited) cases, trust assessments are adjusted to consider this absence in the overall evaluation. The remaining ECUs fall into

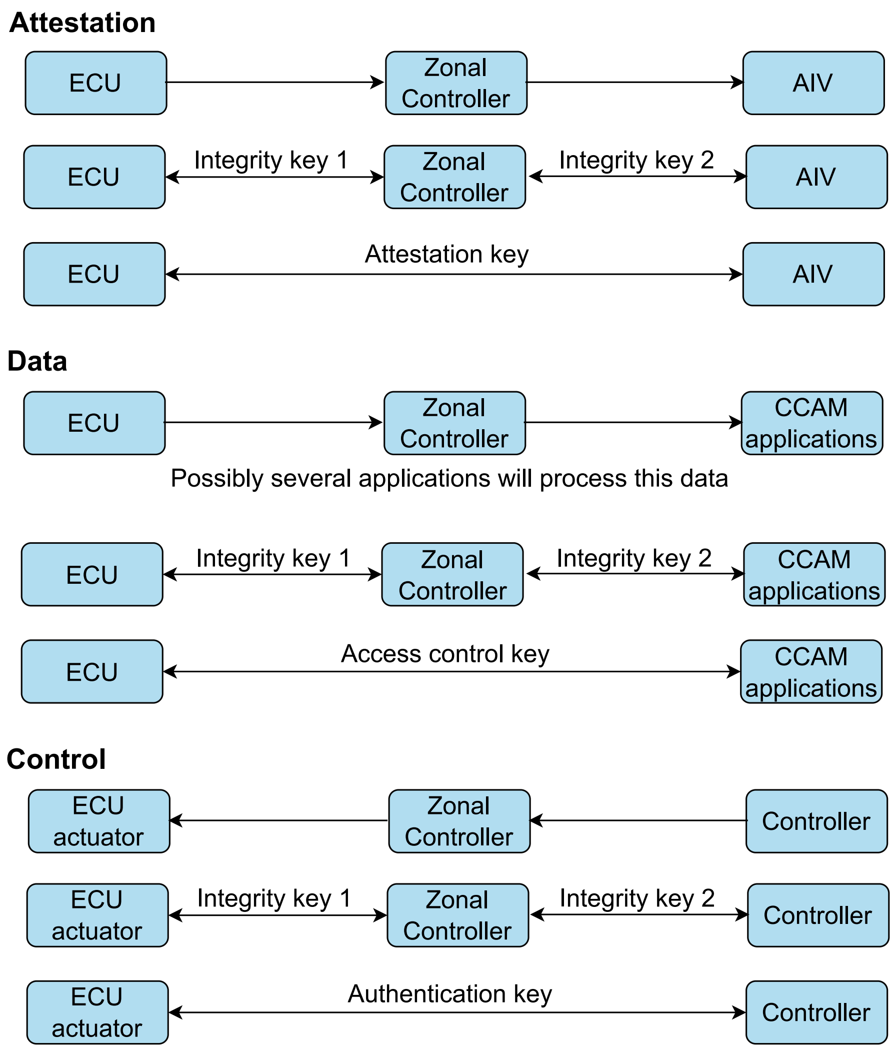

Figure 3.5: *CONNECT* ECU keys.

two categories: those capable of both asymmetric and symmetric cryptography (*A-ECUs*) and lower-powered devices limited to symmetric cryptography (*S-ECUs*).

A-ECU *A-ECUs* will have a *RoT* based upon a *TEE*. For these *ECUs* to enable the *CONNECT* services, the *TCB* should include the CPU, the memory, the firmware, the operating system and a Key Manager that will support the use of key usage restriction policies.

S-ECU *S-ECUs*, can only support symmetric cryptography. In addition, the *RoT* is not provided by a *TEE*, but (where possible) by a Hardware Security Module (HSM). These can provide Key Management capabilities, but without any key usage restriction policies. In addition, because a *TEE* is not an option there will also not be a tracer.

While some form of hardware support is a key requirement for *CONNECT* this may not always be practicable so there might be some *S-ECUs* that do not possess any type of hardware support, this will necessarily be reflected in their treatment by the TAF.

Both types require keys that support very similar sets of requirements, but the details of how they are instantiated and used will be different for *A-ECUs* and *S-ECUs* (see Figure 3.5). The sets of keys required are:

1. **Keys used to ensure the integrity of communication between the *ECU* and its Zonal Controller.** For an *S-ECU* this will be achieved using a MAC (e.g. HMAC) with a pre-shared key. For an *A-ECU* the data will be signed with an asymmetric key for which a certificate is provided to the zonal controller. Alternatively the two asymmetric *ECUs* could first use their asymmetric keys to establish a shared symmetric key and then use it to compute MACs.

2. **Keys used for reporting attestation evidence shared with the *AIV* component in the vehicle computer.** For an *S-ECU* this will be achieved by sending the attestation evidence to the *AIV* protected by using a MAC (e.g. HMAC) with a shared key. The *AIV* will need to obtain the reference values from the *IAM* to check whether the attestation succeeded. For an *A-ECU* the attestation evidence will be signed using an asymmetric key for which the *AIV* has the certificate. In this case a key restriction policy will be used to ensure that the attestation evidence can only be signed if the attestation succeeds, so if the *AIV* receives a correctly signed set of attestation evidence it knows that the attestation succeeded.

3. **Keys used for ensuring the provenance of data exchanged between the *ECU* and the entities that process, or provide it.** For example, a device controlling the accelerator needs to be sure that the commands it receives came from the correct source. For an *S-ECU* the data will be encrypted using a key that is only shared with those entities that are allowed to process, or provide it. For an *A-ECU* the data will also be encrypted, but the symmetric key to be used can be generated using a KEM/DEM protocol. In both cases AES can be used with an appropriate mode (the GCM mode is a good choice as it provides authenticated encryption).

4. **Keys used to encrypt the data from an *ECU* to ensure that only allowed applications can process that data.** The options here are the same as for the previous item.

3.3.2 Zonal Controller Keys

The Zonal Controller plays a crucial role in the secure communication within the vehicle network. It requires keys to protect the integrity of communication with its *ECUs* and the vehicle computer. Additionally, the Zonal Controller requires a key for its own attestation process. The specific keys used for communication with *ECUs* are determined based on the capabilities of each individual *ECU*, as discussed previously. Essentially functioning as an asymmetric cryptography-capable electronic control unit (*A-ECU*), the Zonal Controller follows key options similar to those outlined for this category. In the context of a federated *TAF*, further keys may also be required, this should be borne in mind, but can be left for the moment.

3.3.3 Vehicle Computer Keys

The vehicle computer is, in essence, a “Systems-of-Systems”; hence, it possess its dedicated *RoT* and *TCB*, enabling it to conduct local attestation to ensure the integrity of the underlying firmware and its software components. The collected attestation evidence from the vehicle computer is sent to the *AIV*. In this case, the *TCB* (being statically measured and verified for its configuration integrity) will be the underlying firmware and software used to govern the correct launch and run-time management of the different containers and applications running on the vehicle. As is the case with the other in-vehicle sensors, this attestation evidence will provide another trust source for the (MEC-instantiated) *TAF* towards the creation of a composite view of the trust state of the entire CCAM continuum. We have to note, however, that the **type of attestation evidence to be collected from the vehicle computer will be different focusing mainly on the integrity and operational assurance of the computing environment where the containers are been instantiated.** The vehicle computer can be represented by a single computing device (where multiple containerized services can be instantiated in isolated environments) or a cluster of computing resources where each container will be mapped to its own core. In both cases, container-centric attestation evidence mainly capture assurances on the execution environments rather than guarantees on the correct behavior of an application which is achieved through key restriction usage policies.

Although the in-vehicle computer supports local attestation, all software components used by *CONNECT* for supporting the continuous trust assessment process further provide their own attestation evidence (i.e., TAF, MD). It shall be noted that the *CONNECT* *TCB* components performing trust-related tasks, are all executed within secure containers. In addition to the *CONNECT* TCB, the *IAM*, the *AIV* and the *TCH* collectively form the **TEE-GSE**. These three components are working collaboratively to attest and verify the evidence collected, not solely from the vehicle computer, but also from other in-vehicle devices (i.e., ECUs), forming a *distributed RoT*. The other devices in the vehicle further provide trustworthiness evidence (usually based upon attestation), hence possess their own *RoTs* and *TCBs*. This distributed RoT provides storage, measurement and reporting for the vehicle computer. Storage capabilities in terms of (failed) attestation evidence is supplemented by the *DLT*. The following paragraphs analyse the different applications running on the vehicle computer.

Containers: Many of the components of the *CONNECT* system will be running inside containers. The different containers will need to provide attestation evidence to the *AIV*, although those that use Gramine inside may make use of Gramine's built-in attestation capabilities (see Chapter 5). Asymmetric attestation keys will be required regardless of the employed mechanism, while the attestation evidence from the containers will be leveraged as trust sources by the *TAF*. The components residing within the containers require keys and certificates to establish secure communications with other components (deployed in different containers) as well as other applications running on the vehicle computer. Note that this requirement may also extend to communication with components in containers running at the *MEC*. Regardless of the specifics of each case, during the on-boarding, the components are configured with asymmetric keys. These keys keys can be used directly for signing or, where necessary, to generate shared encryption keys.The next paragraphs focus on the particular requirements for the different components running on the vehicle computer.

- **Identification and Authentication Management (IAM):** This component will hold the vehicle master key as this is used when integrating the Zonal Controllers and *ECUs* into the vehicle. It will also receive information from the OEM to use when on-boarding a device (integration key, device ID, software versions, attestation reference values, etc). This information will be retained and used when reporting on the devices and also when a device is updated. It will hold certificates for the different components providing it with data so that the authenticity of the data can be verified. It will also use (a) an asymmetric key when signing *VCs* used to provide attestation reference values and configuration information to the *AIV* and (b) pseudonym keys to sign *VPs* used for sending trustworthiness data outside of the vehicle.
- **Attestation and Integrity Verification (AIV):** This component *AIV* will require keys, or certificates, enabling it to verify the received attestation evidence. Once verified, the ev-

idence will be used to generate an attestation report *VC*, which is shared with the *TCH* and the *TAF*. The *AIV* requires an asymmetric signing key to perform this task. The *AIV* is further responsible for storing the failed attestation evidence to the *DLT*. The failed attestation evidence is encrypted using ABE, so that strictly entities with the correct attributes can view them and investigate the failure. Therefore, the *AIV* needs to generate the ABE key to-be-used.

- **Trustworthiness Claims Handler (TCH):** In addition to keys or certificates for validating communication with the *TAF*, *AIV* and *IAM* and the application for the CAM/CPM construction, the *TCH* will require a (HW-based) key for signing the harmonised attributes *VC*. The harmonised attributes, including the TAF and MD reports are included in the *VP*, which is sent outside the vehicle. Therefore, to preserve the privacy of the vehicle, a DAA key is leveraged to sign the harmonised attributes *VC*.
- **Trust Assessment Framework (TAF):** The *TAF* will need keys or certificates for its interaction with *AIV*, the *MBD* and the CCAM applications that generate the *Trust Assessment Request (TAR)s*. It further requires a key for the communication with the *TCH* and a DAA key to sign the Trust Opinions *VC* that it provides to the *TCH* to be included in the *VP* to be sent outside of the vehicle.
- **Mis-behaviour Detection (MBD):** The *MBD* requires keys for decrypting the sensor data that it evaluates. It further necessitates keys for the dissemination of its results to the *TAF* and the *TCH*. Additionally, to ensure anonymous signing of the misbehavior report *VC*, provided by the *TCH* and subsequently included in the *VP*, that is sent outside of the vehicle, the MBD requires a DAA key.
- **CCAM Services:** These services require keys to perform their designated tasks, enabling them to process the data needed. Furthermore, these services necessitate keys for communicating with the *TAF* when making *TARs*.

3.4 *CONNECT* Vehicle Information Flows

The *CONNECT* information flows inside the vehicle are shown in Figure 3.4. Two 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 color capture the interactions that take place between all *CONNECT*-related components for enabling both data- and node-centric trust assessments. The figure focuses on the In-Vehicle topology. More information regarding the MEC topology will be discussed in D4.2 [14], hence are considered out of scope for this deliverable.

Starting with the application data flows, a request for data is sent from an application (in the application layer) to the *Facility Layer (FL)* **(step 1)**. This request is forwarded by the latter (i.e., the facility layer) to the *Zonal controller (ZC)* **(step 2)** and then to the *ECUs* which then acquire the data **(step 3)**. The *ECUs* collect data from their sensors, and send them to the *Zonal controller (ZC)*, which forwards it to the *FL* **(step 4)**. One of the responsibilities of the *FL* component is to associate (i.e., tag) the raw data coming from the sensors to the corresponding device identifiers. Subsequently the tagged data is sent both to the *MBD* to be checked for discrepancies **(step 5a)**, as well as to the application that made the request **(step 5b)**. The application processes the information **(step 6a)** and the results are forwarded to the CAM/CPM Encoder/Serialiser, to be sent to the MEC or other vehicles **(step 6b)**.

Moving to the *CONNECT* data flows, these are related to the invocation of the trust assessment process to the in-vehicle topology. There are various ways that such a process can be triggered. In the scope of this deliverable we focus on the cases where an application is requesting for the execution of a new trust assessment through the application layer (i.e., pull scenarios). However, invocation of a new trust assessment can take place once a change in the topology is identified (i.e., push scenarios). For instance, the *AIV* may detect that an attestation process has failed and a trust assessment needs to be executed in order to re-calculate the ATL value for the affected trust relationships.

By focusing on the pull scenario of Figure 3.1, we have an application requesting for a new trust assessment to the *TAF* **(step 1)**. The *TAF* requires certain evidence in order to calculate a Trust Opinion (i.e., ATL), so it sends a *RFE* (Request for Evidence) to the *AIV* Component **(step 2)**. The *RFE* specifies which devices should provide attestation evidence and whether the resulting attestation reports should be sent immediately, or periodically. Upon receipt of the *RFE*, the *AIV* requests the evidence from the *Zonal controller (ZC)* **(step 3a)**, which forwards the requests to the *ECUs* (including both *A-ECUs* **(step 3b)** and *S-ECUs* **(step 3c)**). The *Zonal controller (ZC)* collects this attestation evidence and forwards it to the *AIV* **(step 4)**, that is responsible for verifying correctness of the attested devices.

Some devices, such as the *A-ECUs* and *Zonal controller (ZC)s*, may support local attestation, leveraging the key restriction usage policies, offered by the *CONNECT* *TEE-GSE*. For these devices, the verification takes place locally, while only a signature on the nonce (e.g., attestation challenge) is sent back to the *AIV*. In this case, the *AIV* is able to verify their state correctness simply by validating the correctness of this digital signature (i.e., without disclosing the low-level system traces). The devices that do not support local attestation (i.e., *S-ECUs*), provide their signed attestation evidence (as quotes) to be remotely verified by the *AIV*, since key restriction usage policies cannot be enforced. The *AIV* needs to further acquire the reference values (i.e., golden hashes), against which the runtime measurements will be assessed, from the *IAM* component **(step 5a)**, to act as the Verifier. The *AIV* generates **(step 5b)** and sends the attestation results to both the *TAF* and the *TCH* **(step 6)**. In parallel, the *MBD* sends a misbehavior report *VC* both to the *TAF* and the *TCH* **(step 7)**. Having all the evidence required, the *TAF* performs its assessment and sends the trust opinion *VC* to the *TCH* **(step 8)**.

The *TCH* is responsible for the harmonization of the received evidence. This is a crucial step before building the final *VP* since the trust-related information that accompanies a CAM/CPM message need to be provided in a privacy-preserving manner to address the risk of vehicle fingerprinting. The *TCH* component uses the harmonized attributes to form a Trustworthiness Claim and along with the *VCs* from the *TAF*, the *MBD* it constructs the TCH *VP*. This essentially discloses those attributes needed (from all aforementioned credentials) with the necessary level of abstraction so as to enable vehicle-wide trust assessment (from the receiving CCAM actor) while not impeding on tne privacy profile of the vehicle. The TCH *VP* is sent to the *IAM* **(step 10)**, which signs the *VP* using a PKI-based pseudonym **(step 11)** and sends it to the CAM/CPM Encoder/Serialiser **(step 12)**. The last step before sending this *VP* to the MEC or other vehicles, is to serialise the T-CAM/T-CPM message **(step 13)**.

3.5 *CONNECT* TEE Design Choice

Today, vehicles are increasingly digital and many functionalities are implemented by a large number of diverse ECUs. Due to the increasing focus on software and AI-based algorithms, the required compute power in the vehicle is rapidly increasing. In particular for autonomous driving, the vehicle functionality requires many powerful ECUs. As a consequence, we see a clear trend towards more powerful (central) compute clusters in the vehicle. To address this need in a cost efficient way, the industry trends from small fixed-function ECUs towards larger general-purpose clusters. These larger clusters are often services by deploying commercial off-the-shelf CPUs. To address this trend, we decided to test and validate the commercially available Intel SGX Trusted Execution Environment that is provided and supported by beneficiary Intel.

As outlined in Section 2.3.1, Intel SGX allows to execute largely unchanged Linux applications within a hardware-protected TEE, Due to its broad availability in commercial CPUs, is is supported by a wide range of tools and example applications: While we selected Intel SGX, it is important to note that this does not constitute a lock-in to this technology.

While we will prototype and conduct research using this technology, the resulting architecture is largely TEE agnostic: Our architecture will require that security-critical parts are clearly separated from other parts. We then require that the critical parts can be executed within and protected by a TEE. Since we prototype the protected parts as Linux applications, any TEE that can protect Linux-style applications should be rapidly usable to replace Intel SGX if desired. Furthermore, by refactoring / recompiling the applications for other run-time environments, they should be portable to any other sufficiently powerful TEE.

3.5.1 The Gramine Library OS for Intel SGX

To simplify the burden on an individual developer when using the Intel SGX CPU feature, the Open Source Community has developed multiple frameworks that simplify the life of a developer that intends to design TEE-protected applications. Examples include the Gramine LibOS project [37, 38], Haven [6], SCONE [5], Occlum [35] or Panoply [36]. These frameworks put the whole application inside the SGX enclave. Only a small chunk of original logic stays untrusted, namely, input/output (I/O) functionality such as networking, file system, other system calls, and so on.

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. For the *CONNECT* Project we decided to use and extend the Gramine Library OS that is maintained by Intel, one of our Beneficiaries. Gramine allows to seamlessly port Linux applications that implement security-critical services into a Intel-SGX-protected TEE.

Gramine is a TEE run-time to run unmodified Linux applications on different platforms in different environments [37, 38]. 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 a TEE that is protected by Intel SGX. The currently available and most widely used configuration is running applications inside an Intel SGX enclave on top of the untrusted Linux kernel.

When using InteL SGX, software developers can port their applications to Intel SGX by putting only the security-critical parts 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:

Chapter 4

User Stories for Security-Critical Features of CONNECT

Chapter 4 specifies important required behaviors of our architecture using so-called ”user stories”. 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*.

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 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 Chapter 5) 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

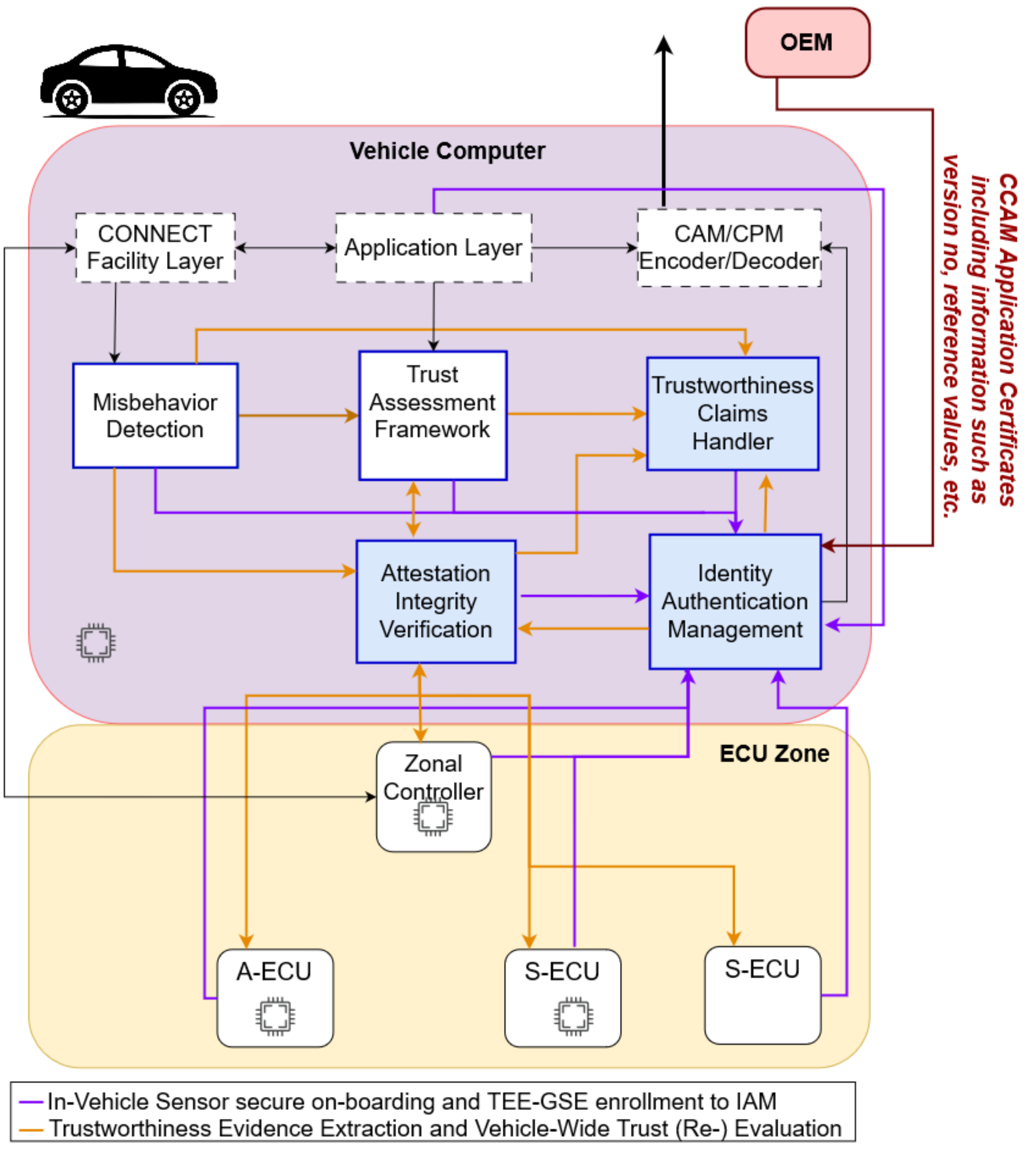
OEM
Vehicle Computer
CONNECT Facility Layer
Application Layer
CAM/CPM Encoder/Decoder
Misbehavior Detection
Trust Assessment Framework
Trustworthiness Claims Handler
Attestation Integrity Verification
Identity Authentication Management
CCAM Application Certificates including information such as version no, reference values, etc.
ECU Zone
Zonal Controller
A-ECU
S-ECU
S-ECU
In-Vehicle Sensor secure on-boarding and TEE-GSE enrollment to IAM
Trustworthiness Evidence Extraction and Vehicle-Wide Trust (Re-) Evaluation

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

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

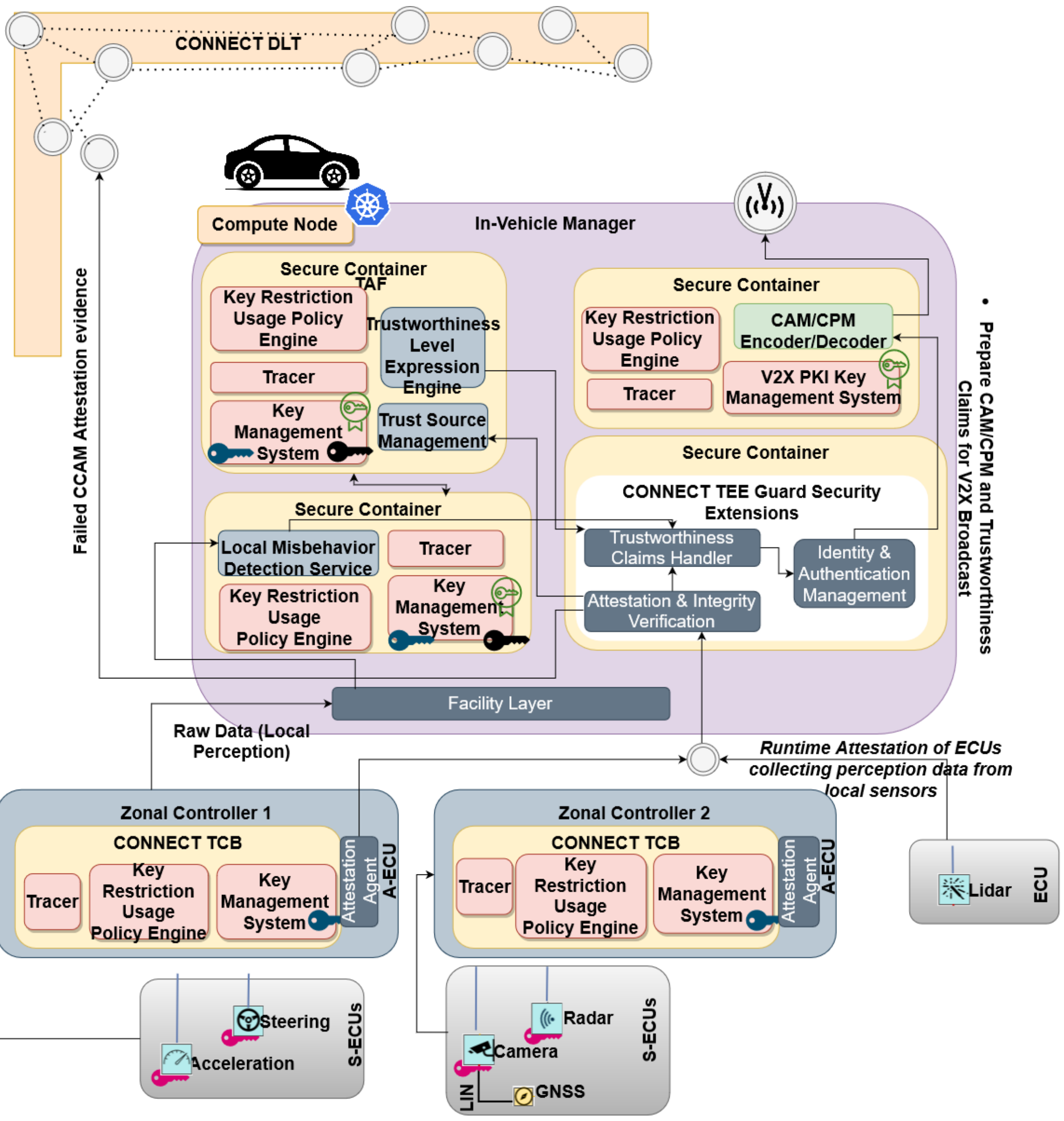
CONNECT DLT
Failed CCAM Attestation evidence
Compute Node
In-Vehicle Manager
Secure Container
TAF
Key Restriction Usage Policy Engine
Trustworthiness Level Expression Engine
Tracer
Key Management System
Trust Source Management
Secure Container
Local Misbehavior Detection Service
Tracer
Key Restriction Usage Policy Engine
Key Management System
Secure Container
Key Restriction Usage Policy Engine
CAM/CPM Encoder/Decoder
Tracer
V2X PKI Key Management System
Secure Container
CONNECT TEE Guard Security Extensions
Trustworthiness Claims Handler
Identity & Authentication Management
Attestation & Integrity Verification
Prepare CAM/CPM and Trustworthiness Claims for V2X Broadcast
Facility Layer
Raw Data (Local Perception)
Runtime Attestation of ECUs collecting perception data from local sensors
Zonal Controller 1
CONNECT TCB
Tracer
Key Restriction Usage Policy Engine
Key Management System
Attestation Agent
A-ECU
Zonal Controller 2
CONNECT TCB
Tracer
Key Restriction Usage Policy Engine
Key Management System
Attestation Agent
A-ECU
Lidar
ECU
Steering
Acceleration
S-ECUs
Radar
Camera
GNSS
LIN
S-ECUs

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 35.

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 *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* supplier.

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 manages the installation and update of the different software components running on the vehicle computer and stores the *VCs* containing their configuration and attestation information. 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 (see Deliverable D4.2 for details).

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 Section 3.3). The details will vary depending on the type of *ECU* and how they will be used. A detailed description for each ECU and application will be given in Deliverable D4.2.

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 authorized 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 *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 harmonization mechanisms, in order to create a harmonization/abstraction of the attributes of the engaged devices and eventually create a VP of such a harmonization. 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 [13] and D5.1 [12], 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. Moreover, as described in deliverable D5.1 [12], we need to employ privacy enhancing technologies for the creation of the verifiable evidence. Thus, *CONNECT* ensures that each *VP* will not breech the privacy of a vehicle but it will allow authorized entities to trace back to the failed attestation evidence producer. This requires the use of advanced crypto primitives like the *DAA* which will be elaborated in D4.2 [14].

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 (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 [12], 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. The harmonization mechanisms and the exact type of *VCs* that are going to be employed, are going to be investigated in the deliverable D4.2 [14].

Motivation: As described in the deliverable D5.1 [12], *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 .

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 [11].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 MD I want to self-issue a valid misbehavior report VC for the data that is being sent.

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

Motivation: A misbehavior 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 Misbehavior Detection component to construct a Misbehavior Report from the evidence collected by the zonal controllers and their underlying ECUs. The constructed Misbehavior Report is sent to the *TCH* and is included to a T-CPM message. Because the Misbehavior report is shared with other components/entities, it needs to be created in a verifiable manner. To be more precise, for the finalization of the Misbehavior Detection report the MD calculates a digital signature over the misbehavior checks for a particular observation and then associates the signature with a link token in order to construct its verifiable credential. Moreover, as the Misbehavior Detection report is included in a T-CPM message, a lot of privacy implications are raised, as any external entity shouldn't be able to extract any information about the identity of the signing *MBD*. To resolve this issue, the *VC* provided by the Misbehavior Detection component needs to be anonymized.

Requirements: For this usage story to be feasible the Misbehavior Detection (*MBD*) component needs to have a mapping of all the devices keys, that corresponds to communication integrity and data integrity. More specifically, the Misbehavior 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 neighboring 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, authorized 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 does not 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 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 unauthorized modification and information leakage by executing it inside a *TEE*. The following requirements are desirable:

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 requirements are desirable:

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

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 requirements are desirable:

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 requirements are desirable:

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 requirements are desirable:

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.
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.
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.

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 requirements are desirable:

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 requirements are desirable:

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.

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.

NOTE: In this user story, I mean a new feature of Gramine: Gramine state relocation. Alternatively, each application can be modified to migrate its own state, instead of a generic Gramine solution.

The following requirements are desirable:

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.

Chapter 5

Hardware-Backed Trusted Execution Extensions for CONNECT

In Chapter 5, we detail the Intel SGX hardware Trusted Execution Environment (TEE) that is used as the underlying trust anchor in *CONNECT*. While Chapter 2 surveyed the state of the art and gave a high-level overview over the Intel SGX TEE technology, we now focus on the software stack to be used (and extended) in *CONNECT*. Particular focus is given to the Gramine Library OS that allows seamless migration of Linux applications into a run-time environment that is protected by the Intel SGX TEE.

5.1 Intel SGX Features in Detail

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 Section 2.3.1. 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 (Story-XVI, Story-XVII) while attestation specifically is required for Story-XVII.

5.1.1 Intel SGX Component: Memory Encryption

At the hardware level, the security guarantees of Intel SGX are enforced using a number of enhanced hardware flows. The first flow is how the enclave code and data is encrypted in the system, and it is illustrated on Figure 5.1.

Enclave code/data leaving the CPU is transparently encrypted, and its integrity and replay protection metadata is calculated and securely stored within the enclave memory. Similarly, when receiving enclave data from system memory, as soon as it reaches the CPU, the data is decrypted and checked for integrity and replay attacks. Enclave data is finally consumed by the CPU only when all these checks pass.

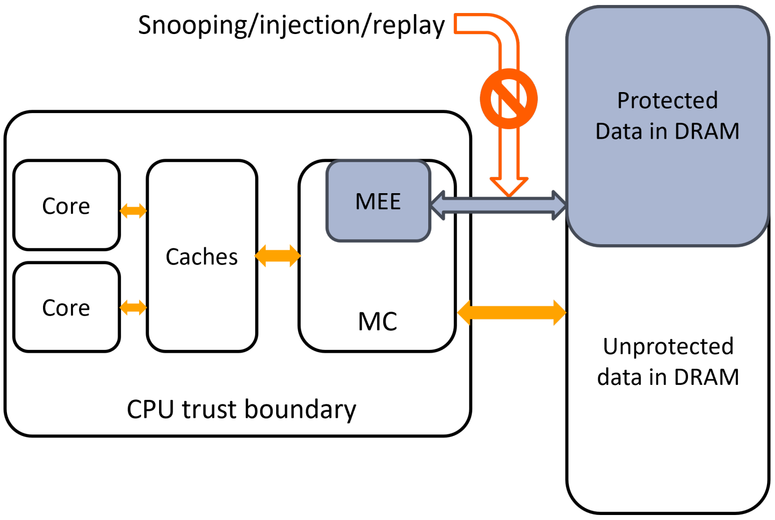

Figure 5.1: Intel SGX memory encryption engine (MEE) ensures that unprotected (orange) and protected (grey) memory accesses are separated. Encryption protects memory from snooping, injection and replay attacks.

Internally, SGX uses a special memory encryption engine for memory protection. This engine is an extension of a memory controller. Every access to protected memory goes through this component, which provides in-flight encryption/decryption for memory confidentiality. This engine also uses an integrity tree to provide memory integrity and replay attack protection.

With this encryption engine mediating all accesses to protected data, the attacker cannot launch an attack. Any attempt by the attacker to snoop enclave data is futile, since protected data is encrypted. Additionally, any attempt by the attacker to inject or replay wrong data is detected by integrity-tree checks.

5.1.2 Intel SGX component: Protected Entry and Exit Points

Another SGX-specific hardware flow contains the enclave entry (EENTER) and enclave exit (EEXIT) controlled points. These points of entry into/exit from an SGX enclave are controlled by the CPU. Entering an enclave is only allowed after passing certain checks. The CPU enforces these access control checks at special enclave entry points. Exiting an enclave is also controlled by the CPU to guarantee no data leakage outside of the enclave. The corresponding diagram can be seen on Figure 5.2.

After the SGX enclave is deployed and initialized on the system, the CPU can be directed to jump in and out of the enclave code via special SGX instructions. The EENTER instruction transfers execution from the host application to the enclave. EENTER checks the metadata of the enclave and makes sure that this enclave is properly initialized. After a successful EENTER, the enclave code is executed until a system event or an exit instruction causes enclave execution to pause. Once the EEXIT instruction is invoked, the execution is transferred back to the regular code of the host application. Afterwards, the application can enter the enclave again via an EENTER instruction. Depending on the enclave workload, it may request many EENTERs and EEXITs, and enclave enter and exit events may be frequent.

Intel SGX dictates a strict, well-defined interface for data transfers to and from the enclave. It establishes a trusted mechanism that controls data transfers. The transition between untrusted

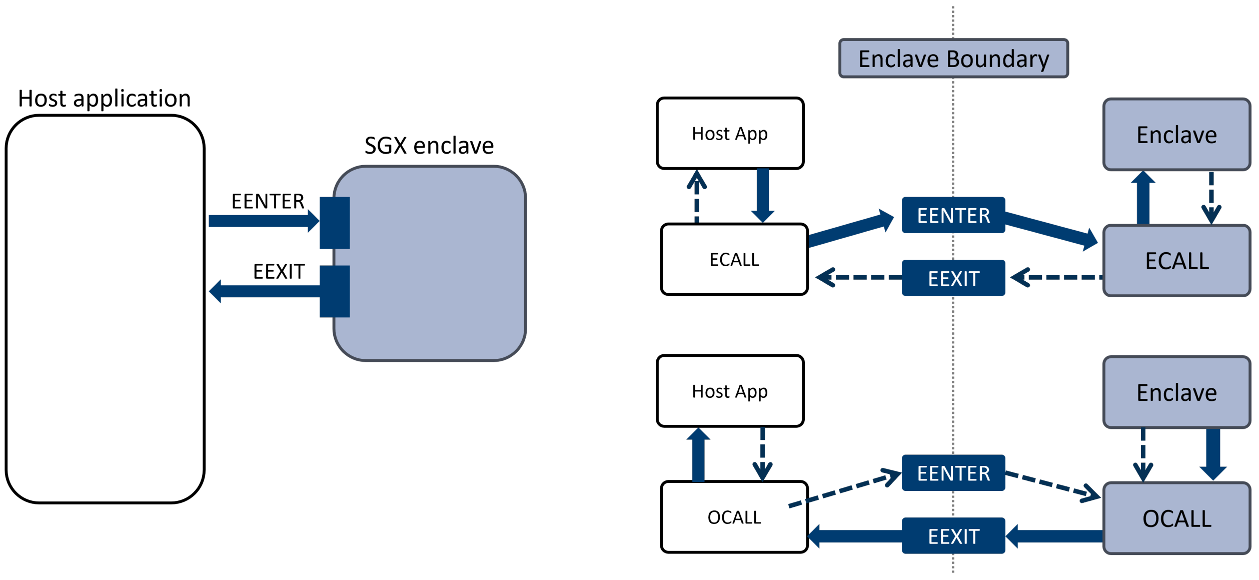

Figure 5.2: Intel SGX controlled entry and exit points (ECALLs and OCALLs).

code and enclave code is done only by EENTER and EEXIT to avoid control flow attacks. Intel SGX allows only a few predefined entry points inside the enclave, so the attacker cannot jump to an arbitrary point in enclave code.

In reality, application developers do not write assembly code with EENTER/EEXIT instructions. Intel SGX provides a programming abstraction that hides the complexity of using EENTER/EEXIT and provides a familiar, C-style function interface.

This programming abstraction operates on *Enclave Calls (ECALLs)* and *Outside Calls (OCALLs)*. As the names imply, ECALLs are used when an untrusted host application calls an interface function of the enclave, and execution is transferred to the enclave. OCALLs are used in the opposite direction, when an enclave calls a function of the host application, and execution is transferred to the untrusted part.

ECALLs and OCALLs are very similar to normal C functions. Under the hood, however, they perform data transfers and SGX-imposed checks, and execute EENTER/EEXIT instructions. In this sense, ECALLs and OCALLs are actually similar to Remote Procedure Calls (RPCs). The diagrams of ECALL and OCALL execution are depicted in Figure 5.2. The ECALL description follows:

1. The developer of the application adds an ECALL to the untrusted code. At run-time, the host application wants to execute a sensitive computation on sensitive data. For this, it needs to enter the enclave via an ECALL.

2. The untrusted part of the ECALL logic marshals ECALL arguments (flattens them so that the enclave can understand and consume them). Then it prepares the CPU to enter the enclave mode and invokes EENTER instruction.

3. Now, inside the enclave, the trusted part of the ECALL logic unmarshalls ECALL arguments passed by the host application and forwards them to the enclave code. The trusted ECALL logic must copy all arguments from untrusted system memory to trusted enclave memory, to prevent Time-of-Check to Time-of-Use (TOCTOU) attacks.

4. The trusted enclave code can now perform sensitive computations over sensitive data. When the enclave is done, it returns to the host application.
5. To return from ECALL, the enclave invokes the trusted part of ECALL, which in turn invokes the EEXIT instruction to exit enclave mode. The untrusted part of ECALL forwards the return values of ECALL to the host application (if there are any return values), and the host application continues its untrusted execution. The trusted ECALL logic must copy all return values from trusted enclave memory to untrusted system memory, because the untrusted part of ECALL has no access to these in-enclave values otherwise.

A similar flow happens for OCALLs, when the enclave's trusted code wants to execute a non-sensitive computation, for example network or file system I/O. Typically, the data intended for enclave consumption is stored encrypted on the hard disk (or received encrypted from the network) and in untrusted host memory, and when brought inside the trusted enclave, it is decrypted to plaintext. Similarly, the results of enclave execution are typically encrypted before leaving the enclave.

Enclaves usually use OCALLs to perform untrusted I/O. For example, the enclave may ask the host application to receive TCP/IP packets from the network. These packets are usually encrypted with SSL/TLS, so the untrusted host cannot peek into them. The untrusted host will put received packets in a special buffer and finish the OCALL. The enclave will get a copy of these packets in the trusted buffer and can safely decrypt and process packets. Similarly, the enclave may ask the host application to send a TCP/IP packet to the network. The enclave then may encrypt the packet with SSL/TLS, put it in a special buffer which will be copied in untrusted host memory, and the corresponding OCALL will push this encrypted packet to the network.

5.1.3 Intel SGX component: Attestation

Intel SGX is intended for remote-server computations. As mentioned before, a remote user must gain trust in an SGX application running remotely, in an untrusted environment. SGX provides two primitives to establish trust: *SGX local attestation* and *SGX remote attestation*.

Since in the SGX threat model the remote platform can be malicious, we cannot trust the platform software to run SGX applications correctly. In particular, remote users must gain trust in the SGX enclave without having to trust the rest of the platform. Otherwise, various attacks may occur. For instance, the user can establish a secure communication channel with a remote SGX enclave. However, this enclave is not the intended one but rather a malicious one that only simulates correct execution (but in reality can potentially forward secret data to third parties). Another potential attack example happens in the case where the user establishes a secure communication channel with the correct remote SGX enclave indeed, but this enclave executes on a malicious platform that only emulates a correct CPU, but in reality "forgets" to switch to enclave mode.

The solution to the problems outlined above is SGX attestation. Attestation is a process in which the attesting entity (the SGX enclave in this case) must produce a proof that it is (or was initially) in some specific state. Having obtained this proof, the remote user will verify it and thus gain trust in the enclave: Before entrusting the secrets to the remote enclave, the remote user needs to make sure that *(1)* this enclave is trustworthy and *(2)* this enclave runs on a trustworthy platform.

How to perform attestation in practice? The formula is simple:

$$Attestation = Signature(Measurement) \tag{5.1}$$

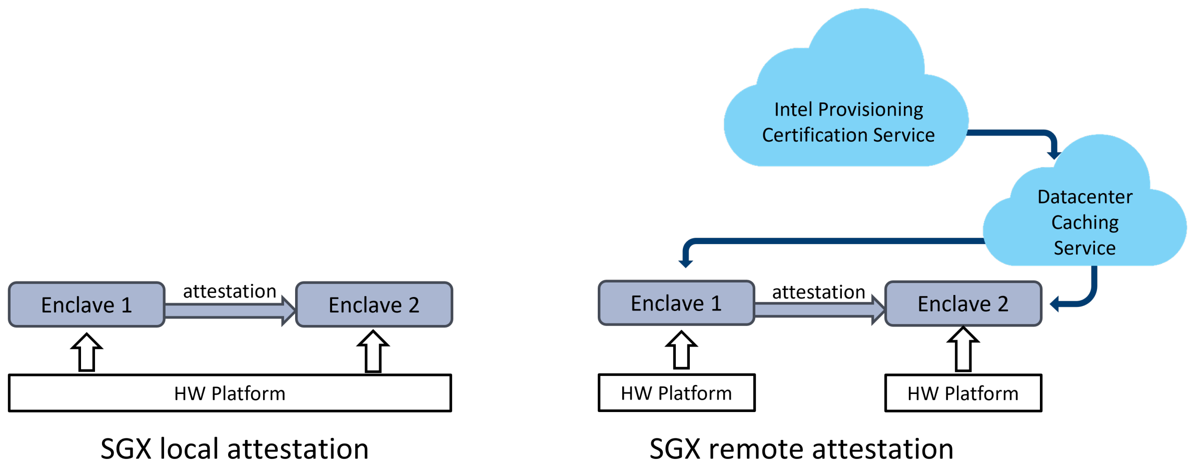

Figure 5.3: Intel SGX local and remote attestation.

When we “attest a system”, we digitally sign the measurement of the system and later verify this signature and the signed measurement. In other words, to “attest an SGX enclave”, we “digitally sign the measurement of the enclave”.

In case of SGX enclaves, the *measurement* is the enclave. The enclave is a set of code and data pages loaded inside trusted memory at startup. Since the startup process is deterministic, and the initial enclave code and data are always the same, we can, during enclave initialization, measure all code and data using a cryptographic hash function. Intel SGX does it automatically: as soon as the SGX enclave is finalized, its measurement is also finalized and securely stored inside enclave memory.

Computing the signature over this measurement brings a number of open questions, e.g., what entity performs this operation, and how remote users anchor their trust. Since remote users must trust Intel CPU to execute the enclave anyway, they can root their trust in this Intel CPU. Each Intel processor possesses a unique, nonforgeable secret stored inside the processor fuses. This unique secret is used to produce a platform-unique signature. Thus, the enclave measurement can be signed by the unique private key of the Intel CPU, and the remote user can verify the signature by comparing it with the known public key (that Intel provides).

To summarize, the enclave measurement assures the remote user that *the correct enclave is executed*. The Intel CPU signature, with which the enclave measurement is signed, assures the remote user that *the correct platform executes this enclave*.

5.2 *Gramine* - A Library OS for Seamless Protection of Linux-style Applications

As outlined in Section 2.2.2, 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 [37, 38]. 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.

5.2.1 Migrating Applications onto *Gramine*

Intel SGX Example: Enclavizing an Application with *Gramine* Tools

The main task of “graminizing” 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. In particular, the manifest template needs to be expanded to the interim

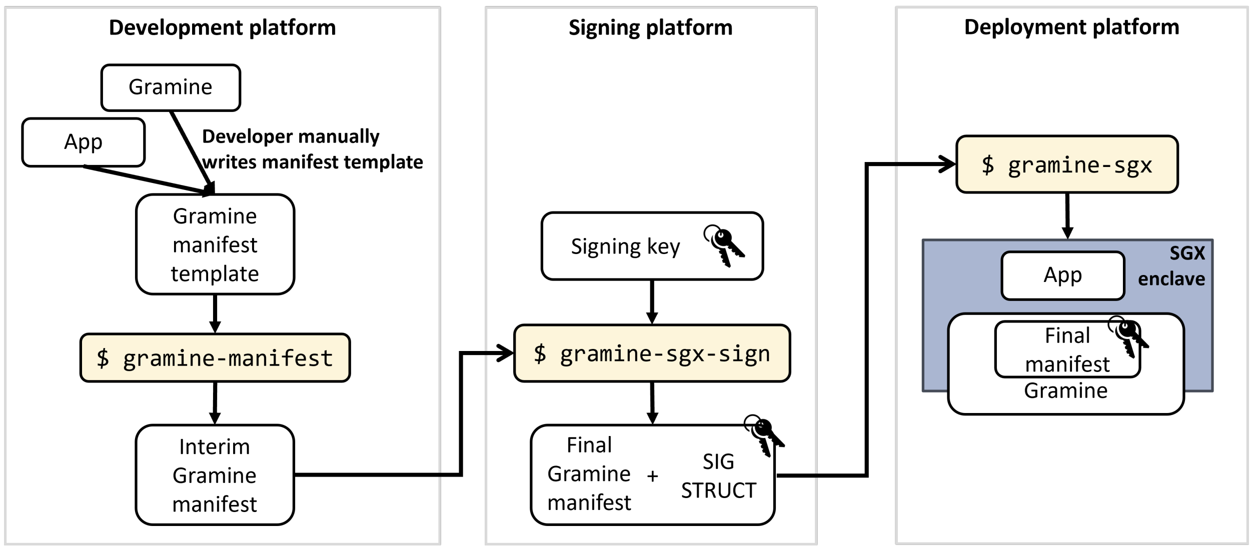

Figure 5.4: *Gramine* story: porting application to *Gramine* with SGX backend.

version (where all paths are replaced with absolute paths) with the `gramine-manifest` tool. Then the interim manifest must be signed with the private enclave-signing key, such that the authenticity of the graminized application can be proved later during SGX attestation. This is achieved with the `gramine-sgx-sign` tool, that produces the final manifest file and the SGX-specific SIGSTRUCT file as outputs. After this step, the bundle consisting of the application, *Gramine*, final manifest file, and the SIGSTRUCT file can be shipped to an SGX-enabled deployment platform and run there via the `gramine-sgx` tool. Below is the command-line snippet on how to “graminize” the application:

```
# prepare the manifest file for your app
$ vim python.manifest.template

# generate Gramine- and SGX-specific files
$ gramine-manifest python.manifest.template python.manifest
$ gramine-sgx-sign --key signing.key --manifest python.manifest \
                   --output python.manifest.sgx

# run Python workload in Gramine
$ gramine-sgx python -c 'print("Hello, world")'
Hello, world
```

This sequence of steps is depicted in Figure 5.4. Note that *Gramine* supports two-phase signing: the development and testing happens on the development platform (with some dummy signing key) and the actual act of signing happens on a separate signing platform. This split is done to protect the signing key from being stolen; ideally only a dedicated person has access to the signing platform (such a platform may not even have network access).

5.3 Initial Architecture of *CONNECT* Extensions to Gramine

Within *CONNECT* we will optimize the features of Gramine and Intel SGX to meet the requirements and functional specifications identified in Chapter 4. In addition, the *CONNECT* architecture requires a range of new features and capabilities that we need to design and implement to close gaps identified by those requirements. In this section, we now outline the high-level architecture for such envisioned extensions to Gramine and Intel SGX.

5.3.1 Motivation and Objectives

Today, and Intel SGX enclave is part of a user process. One design goal is that an operating system can *manage* the user-space process including the enclosed enclave transparently without requiring any change. As a consequence, while the OS cannot inspect or modify the enclave, it can suspend and resume and enclave and roll-back an enclave to an earlier. This would e.g. break the requirement that after an upgrade, the enclave with outdated (potentially buggy) software can no longer be used.

Connection to User Stories: Today's design of Intel SGX and Gramine do not protect against state rollback or unauthorized cloning (Story-XXII, Story-XXIII. We now outline how to meet these requirements by extending Gramine.

The Intel SGX hardware technology does *not* support sharing the state between two SGX enclaves. As a consequence, the classic implementation of forking in Linux is impossible in Gramine. Traditionally in the Linux world, fork is implemented via Copy-on-Write (CoW) technique that initially shares memory pages between two processes and duplicates them only when the state of the two processes starts to diverge. This CoW technique cannot be applied in the SGX environment, because each SGX enclave encrypts its memory pages with a private key, and thus no two SGX enclaves can access the same memory.

This is the reason why Gramine implements the checkpoint-and-restore logic: upon a fork request from the application, Gramine collects all the application state as well as the internal state of Gramine itself, serializes this state and combines it in a single byte sequence. This byte sequence may be sent to another (child-process) SGX enclave via an encrypted pipe, or may be dumped to a file (encrypted with an SGX-platform specific key or with a key-broker-service obtained key). This checkpoint-and-restore logic of Gramine can be re-purposed for several scenarios: (1) parent SGX enclave forking a child SGX enclave, as explained above, (2) performing upgrades of the application or Gramine without losing its state, and (3) migrating the application with Gramine to another platform. In the following subsections, we describe in detail each of these scenarios.

5.3.2 Gramine State Relocation: Forking

Forking is the only currently fully implemented usage of the checkpoint-and-restore technique in Gramine. Figure 5.5 shows this flow. Whenever an application wants to spawn a new child process, it invokes the `fork()` system call. The fork system call is intended to create a new empty process and populate it with the exact same copy of the state as in the parent (forking) process. Therefore, the task of Gramine is to collect all the state of the parent SGX enclave and securely copy it into an empty SGX enclave of the child process.

Initially, the application performs some computations and processing and grows its application state. For example, the application may open some files and establish some network connections. The application's state thus will contain the buffers with data read from the files and buffers with data read from the network. Since Gramine intercepts all file and network I/O requests from the

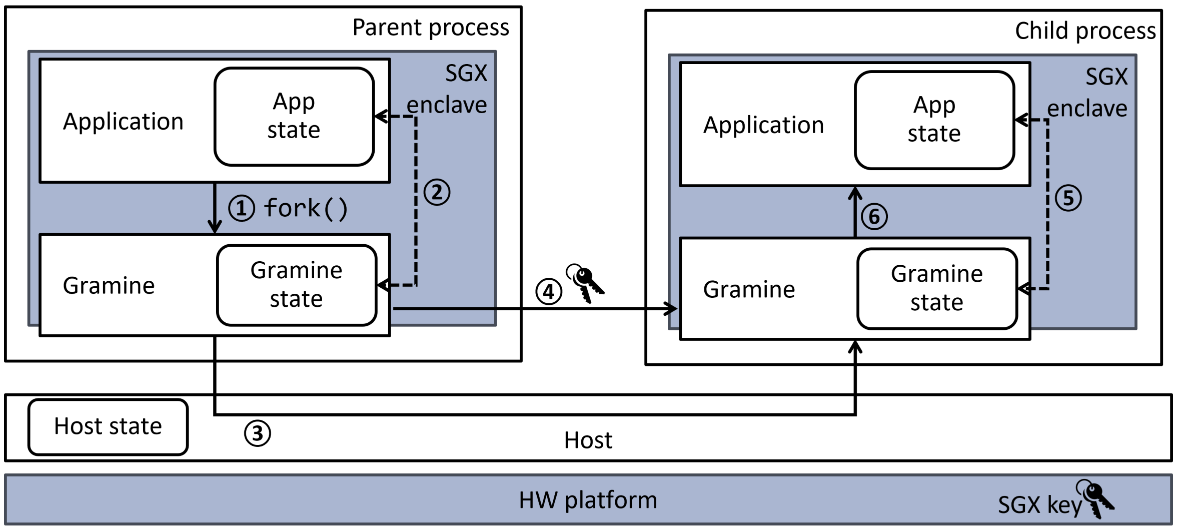

Figure 5.5: Gramine state relocation: fork scenario.

application and keeps the metadata on the opened files and established network connections, Gramine's state grows to include this metadata (e.g., Gramine will memorize file names, file sizes and current offsets in the files, as well as the IP addresses and ports of each network connection).

At some point, the application may decide to fork a sub-process – for example, the application notices a high load and decides to create a worker process to offload some computations to this worker. The application (the parent process) invokes `fork()` at this point (1). This fork request is intercepted by Gramine LibraryOS and processed by Gramine: all application state and all the corresponding Gramine-internal state is collected into one single blob, with all data serialized to get rid of pointers to specific memory locations (2). This single blob is simply a sequence of bytes which can then be sent to the child process. Simultaneously with collecting the state, Gramine requests the host to create a new empty process with a new empty SGX enclave (3). Gramine instance in the parent process connects to the newly created Gramine instance in the child process, performs SGX local attestation so that both the parent and the child SGX enclaves can gain trust in each other, and establishes an encrypted pipe. Note that the established pipe between two SGX enclaves is encrypted with a key known to both SGX enclaves (and only to them), because the two enclaves run on the same SGX hardware platform and thus can derive the same shared SGX-platform-specific key. After this secure pipe is established, Gramine in the parent process sends the collected state (called a checkpoint) on this pipe to Gramine in the child process. The child-enclave Gramine instance receives the state and restores it: Gramine de-serializes the state, splits it back into application state and Gramine-internal state and rewires all data structures to use pointers to corresponding memory locations (5). After this step, the state in the child SGX enclave is the exact replica of the state in the parent SGX enclave, and Gramine can pass control back to the application that will continue from the location right after `fork()` (6).

5.3.3 Gramine State Relocation: Upgrade

Upgrading the Gramine binaries is a not yet implemented usage of the checkpoint-and-restore technique. Figure 5.6 shows this flow. It may be beneficial to periodically upgrade the Gramine version without losing the current application state. This may be important if for example an

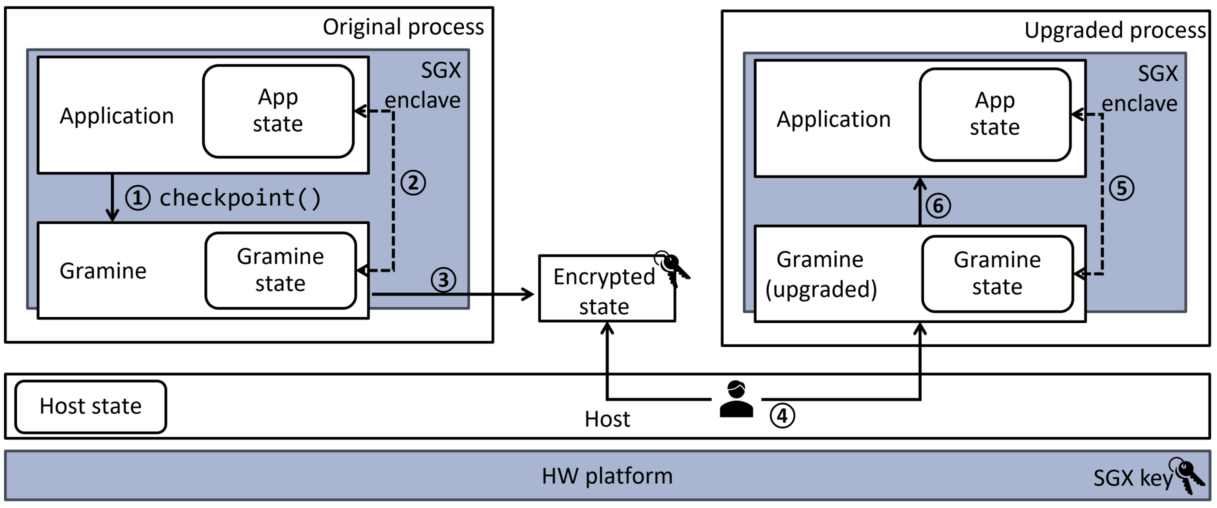

Figure 5.6: Gramine state relocation: upgrade scenario.

application is running on Gramine version 1.4, and that version was found to be vulnerable to some security bug, and it is recommended to run applications under Gramine version 1.5 (which fixed the security bug).

This can be achieved using the following flow. First, the application developer updates the application configuration to the new version of Gramine, by writing a new manifest (if required due to Gramine changes), re-building the required SGX files (e.g. SIGSTRUCT) and verifying that the new application works correctly under the new version. Then, the application developer ships the application configuration bundle together with the new version of Gramine to the same machine where the current version of Gramine with the application is running.

After this preparation, the upgrade process can start. The application developer triggers the special `checkpoint()` system call in the original process, that runs the old version of Gramine (1). The triggering can be carried out using a signal to the process (e.g. `SIGTERM`). Gramine intercepts `checkpoint()` and checkpoints all the state into a single blob (2). In contrast to the previous scenario with `fork()`, Gramine does not create a new process immediately and does not establish a pipe connection. Instead, Gramine dumps the checkpoint blob into a file specified as an argument to `checkpoint()` (3). This file is supposed to be marked as *encrypted* with the SGX-platform-specific key in the Gramine manifest. In this case, only another Gramine SGX enclave on the same platform would be able to decrypt this file and restore the state.

After the state is securely dumped into a file, the developer may stop the previous version of the SGX enclave. Now the developer may start the upgraded Gramine version from the new bundle prepared as described above. The developer starts Gramine in a special “restore” mode, specifying as a command-line argument the file that contains the encrypted state (4). Gramine – which is now the updated version – decrypts this file using the SGX-platform-specific key and restores the resulting checkpoint, similarly to the fork scenario described in the previous subsection (5). After the restore is finished, all Gramine and application data is ready for use, and Gramine passes control to the application, to run from the location right after `checkpoint()` (6).

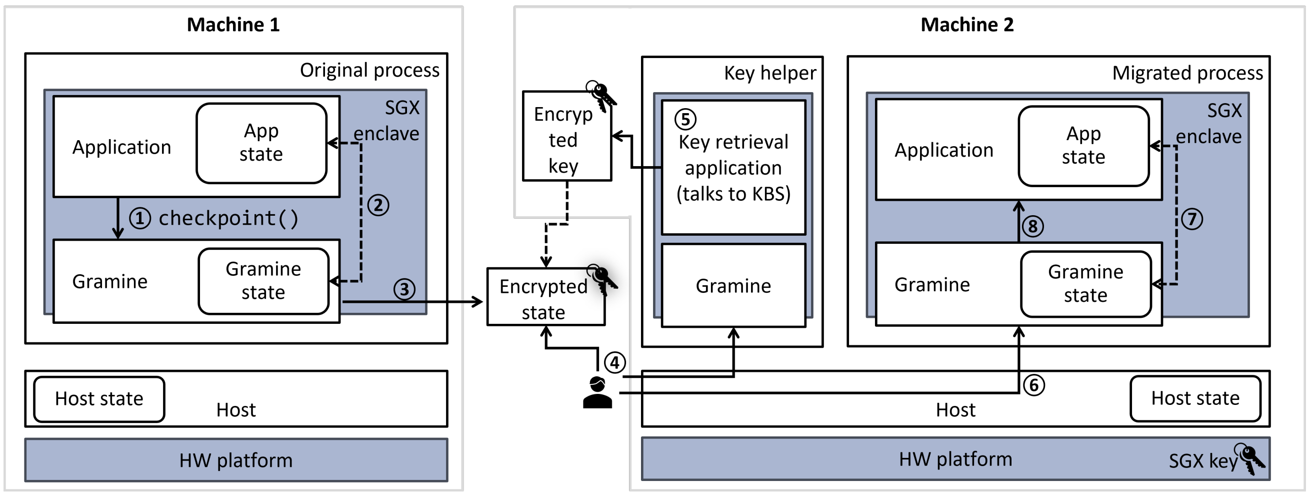

Figure 5.7: Gramine state relocation: migration scenario.

5.3.4 Gramine State Relocation: Migration

Migrating the application with Gramine is another new usage of the checkpoint-and-restore technique. Figure 5.7 shows this flow. Migration is different from the previous two scenarios in that both the application and the Gramine state have to be moved from one SGX platform to another SGX platform. This difference precludes the use of the same SGX-platform-specific key to encrypt the checkpointed state on one machine and to decrypt it on another machine. Therefore, an encryption key needs to come from a third party, which we will call the Key Broker Service (KBS). Fortunately, Gramine manifest syntax supports encrypted files that use not only SGX-platform-specific keys, but arbitrary keys obtained from such third parties as KBS.

With this additional complication of KBS key retrieval, the migration process looks similar to the upgrade process: while the original application and Gramine run in an SGX enclave on Machine 1, the application and Gramine bundle is shipped to a new machine (Machine 2). To start the migration process, the application developer triggers the `checkpoint()` system call in the original process. Gramine intercepts this system call and performs the same sequence of steps as in the upgrade scenario; steps (1) - (3) in the figure. They key with which the checkpointed state is encrypted should be provisioned to the original process beforehand by the Key Broker Service (KBS). KBS – not shown on the figure for simplicity – should have a policy of generating random ephemeral keys for the exact purpose of state migration between two Gramine SGX enclaves; KBS should provision the key to the original Gramine enclave and later to the migrated Gramine enclave, and then destroy this ephemeral key.

The application developer must then terminate the SGX enclave on Machine 1 and copy the encrypted-state file to Machine 2. On Machine 2, the developer must first start the "key helper" Gramine SGX enclave (4). The sole 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.

At this point, Machine 2 has all the components to successfully restore the application in the new Gramine SGX enclave: the application + Gramine bundle, the encrypted-state file and the encrypted-key file. 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 (6). Upon SGX enclave initialization, 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 (7). After the restore is finished, 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).

Chapter 6

Towards Formalized Security & Operational Assurance Requirements

Having defined in detail the functional specifications (Chapter 4), depicting the capabilities that the newly developed *CONNECT* TEE extensions need to offer for supporting the continuous trust assessment of all relationships to be established across the entire CCAM continuum, in what follows we proceed with the **formal definition of the security, trust and operational assurance requirements that need to be achieved when reasoning about the trust level of a CCAM actor**. More specifically, it has become apparent that the foundational pillar of such a complex trust assessment procedure is the use of a highly secure trust anchor that serves as a Root-of-Trust: In the context of *CONNECT*, Gramine has been selected as the Trusted Execution Environment (TEE) that can leverage the hardware built-in RoT capabilities of the target (SGX-enabled) devices. There are several works that conduct formal analysis on the security of such commodity RoTs (or parts of it), and also a few ones devoted to verifying the trusted computing service as a whole. However, most of the existing schemes try to **verify security without differentiating the internal cryptography mechanisms of the underlying hardware token from the client application cryptography.** This approach limits, to some extent, the reasoning that can be made about the level of assurance of the overall system [24].

For instance, when exchanging **trustworthiness claims** from the AIV to the *CONNECT* Trust Assessment Framework (TAF), to be leveraged as a **trust source** based on which a fresh trustworthiness appraisal of the entire service graph chain can occur (based on the service-centric trust model managed in the TAF), such claims are generated by the *CONNECT* TEE Guard and the provided attestation enablers and are based on the validation of specific system properties. This process is composed by cryptography operations that the *CONNECT* TEE Guard is performing for self-consumption (i.e., check authenticity of received run-time trace measurements by verifying that they are signed under the correct Tracer's key, correct setup and storage of HW-based and enclave keys, etc.) but also from security functions (i.e., hashing, asymmetric encryption, (blind) signatures, correct execution of *key restriction usage policies*, etc.) that are intended to guarantee the "*verifiability*" of the exchanged evidence to the "*client*" (TAF) application. When trying to reason about the correctness of these operations, that directly translate to the integrity of the provided trustworthiness evidence (which, in turn, allows for the construction of a validated composable view of the trust state of the entire system topology), it becomes rather complex. This sets the challenge ahead: *Because underlying RoTs are by definition trusted, all internal operations including handling of cryptography data can be idealized. However, this does not directly translate to the overall application security that leverages such RoTs.*

Nonetheless, it limits the number of security requirements that need to be achieved by all the developed attestation, secure exchange and identity management protocols (revolving around the use of all the *CONNECT* TEE Guard extensions) to be described in the context of D4.2 [14]. Thus, in the reminder of this chapter, we **focus on differentiating those RoT capabilities and internal cryptographic operations, that are leveraged by the *CONNECT* TCB, from those security controls that are needed by the *CONNECT* TEE Guard to support the entire life-cycle of the continuous trust assessment process (governed by the *CONNECT* TAF):** From the secure on-boarding of all in-vehicle sensors and the establishment of the appropriate *CONNECT*- and application-related keys to the synchronous/asynchronous communication of trustworthiness evidence in a secure and verifiable manner. The endmost goal is that all internal (RoT-centric) operations form the *idealized security model of* CONNECT which is by default trusted. All these operations have been expressed through the adoption of a **predicate-based language** that allows for the construction of formal statements mapped to specific requirements which when considered as a whole can represent the properties offered by the underlying TCB which are considered trusted. Such predicates can then be glued together to model those security functions that are ancillary to the trust assessment process (and are based on the "*idealized security model*" expressed through the predicates) as **axioms** and these constitute the requirements and assurances that the *CONNECT* attestation and secure communication protocols need to achieve (more details on the underpinnings of the *CONNECT* modelling language can be found in Section 6.1). These will set the scene for the security analysis of these schemes as part of D4.2 [14] where the goal is to prove the soundness of all newly developed protocols against the defined axioms.

Such a granular (security) model will enable the reasoning about the **correctness, completeness and zero-knowledge** nature of the *CONNECT* secure life-cycle management functions that leverage the "*perfectly secure*" cryptographic operations of the underlying *CONNECT* TEE, under various adversarial models and for different security and privacy guarantees excluding any possible implications from the leveraged cryptographic primitives. Idealizing the cryptography used internally in a RoT allows to carry out an analysis of the cryptography to the application itself more concisely: Especially when considering anonymity- and unlinkability-preservation when sharing VPs comprising trustworthiness evidence - hence, the need for the appropriate level of abstraction to take place as depicted in Story-X. This approach simplifies the modelling approach of the complex *CONNECT* security functions and attestation protocols that will be described in detailed in the context of D4.2 [14] and are summarized in the following Table 6.1.

Crypto Primitive	**Related User Story**	**Purpose**
Configuration Integrity Verification (CIV) [21, 22]	Story-V, Story-VII	Allows a Verifier (i.e., the AIV component) to ensure that a Prover device (e.g., ECU, Zonal Controller and any other in-vehicle sensor comprising a service graph chain to be assessed) is at an expected configuration state (Section 3.2.1) This is usually invoked during the **secure on-boarding** of an in-vehicle sensor, with the IAM for establishing the appropriate crypto primitives as well as the necessary key restriction usage policies binded to what is the expected state of the device (as certified by the OEM), or periodically during run-time when device integrity evidence is requested by the TAF. Furthermore, this operation is also executed every time there is an update in the device's configuration state; i.e., *software update/upgrade* or *state migration* from another in-vehicle sensor as a result to an indication of risk (i.e., failed attestation evidence).

Crypto Primitive	Related User Story	Purpose
	Story-V, Story-VII	As with all attestation variants, security properties of interest include completeness and soundness of the protocol so that no malicious Prover can convince a Verifier of the validity of a falsified attestation statement (if there are no guarantees on the authenticity of the attestation evidence) as well as the capability of the Prover to produce attestation assertions in a zero-knowledge manner - *no Verifier is required to learn anything additional about the Prover other than the fact that a shared attestation assertion/statement is true.* This obviates the danger of implementation disclosure attacks that may be exposed when details on the configuration of a device (e.g., OS Version, type of firmware, etc.) is leaked [20].
Control-Flow Attestation [19, 32]	Story-VII	Part of the attestation evidence, requested by the TAF component, may have to do with the run-time behavior/operation of a specific device. The Control-Flow Attestation (CFA) scheme enables the verification of the run-time execution of a device by assessing (in real-time) the control-flow graph of a device's execution against a finite state machine model that captures the nominal behavior of the target device. This is usually triggered for assessing only those safety-critical functions of the device and is based on the construction and reception of authentic traces (to be provided by the underlying Tracer of the *CONNECT* TCB).
Composable Attestation of Swarm of Devices	Story-VII	This attestation variant extends the CIV and CFA capabilities of *CONNECT* from 1-Prover to multiple-Provers. Essentially, it allows the AIV to enact upon a Request for Evidence (RFE), received by the TAF, for the collection of fresh trustworthiness evidence from all devices (in-vehicle sensors) that comprise the service graph chain for which the trust appraisal is targeted. This RFE might require the extraction of different types of evidence (based on the type of trust properties to be assessed) and, thus, necessitates the governance of all TEE Device Interfaces (TDIs), exposed by each *CONNECT* TCB (linked to the run-time monitoring of an extended set of device characteristics), by the AIV. This, in turn, allows the collection of the necessary security claims serving as evidence to the trust quantification model processed by the TAF.
Direct Anonymous Attestation (DAA) [28, 3]	Story-VIII, Story-X, Story-XI, Story-XII	Besides trustworthiness evidence on the operational assurance of a device, one has to cater for a number of privacy-preserving properties including *unlinkability*: No receving entity (vehicle or MEC) should be able to link specific trustworthiness evidence back to the origin vehicle so as to avoid possible "vehicle fingerpriting" (i.e., identification of manufacturing brand of vehicle associated with specific software stack versions) which might allow an attacker to try and manipulate existing vulnerabilities of the deployed software. *CONNECT* leverages Direct Anonymous Attestation (DAA) for the provision of privacy-preserving and accountable authentication services leveraging group signatures: Through the use of DAA signatures, the TCH is able to *anonymize* the produced VP while allowing for authorized entities to link back failed (attestation) evidence to devices for further actions.
Local Attestation vs. Remote Attestation	Story-VIII, Story-X, Story-XI, Story-XII	Multiple attestation schemes - such as the CIV and the CFA - can be executed either locally, within the target device, or with the support a remote Verifier. In the former case, the attestation evidence will be checked locally, by the underlying RoT, in the context of the enforcement of key restriction usage policies. This scheme ensures that the use of a specific cryptographic material can be used *if and only if* the respective policy is satisfied.

Crypto Primitive	Related User Story	Purpose
	Story-VIII, Story-X, Story-XI, Story-XII	By enforcing such a policy within the *TEE-GSE* applications, the TAF and the MD component, we rely on the fact that the generation of the Verifiable Credentials and Presentations is achievable under the condition that each entity is at a correct state. In the latter case, devices without the resource capabilities to host a fully-fledged *CONNECT* TCB but a RoT with limited capabilities (such as the S-ECU), can produce *verifiable quotes*, on their state, that can then be verified by the AIV.
Attribute-Based Encryption (ABE)	Story-XI, Story-XV	It is important to allow for an OEM/Security Administrator or any other certification authority, to check the trust state of a device at any point in time. This will not only enable the auditing/certification of the target device (an important feature for the enhanced user acceptance of such CCAM safety-critical services) but will also allow the further processing of evidence from those devices with a decrease in their trustworthiness state so as to identify the exact point of intrusion (zero-day vulnerability). In this context, *CONNECT* will design a novel Attribute-based Encryption (ABE) scheme for safeguarding the confidentiality of failed attestation evidence (when recorded on the Blockchain infrastructure) so that only authorized entities that can prove the ownershup of specific attributes can create the necessary decryption keys. This (decentralized) ABE will remove the need for a centralized component managing the encryption/decryption keys and will allow the devices to leverage the underlying *CONNECT* TCB to check the validity of device properties based on which the necessary encryption/decryption keys can be created. Such properties can span from identity attributes (such as type of requesting entity ("*Only the respective OEM can get access to such low-level attestation evidence and system traces*")) to device characteristics ("*Only devices with the latest set of security patches can get access to such evidence*") so as to minimize the risk of possible leakge/compromise.
Link Tokens	Story-XI, Story-XII	The use of these tokens aims to allow for an external Verifier (outside a Vehicle component) to trace back evidence to a specific device in the case that the attested system is deemed compromised. Essentially, it enriches the privacy features of the employed DAA (*CONNECT* Traceable DAA) algorithm with linkability and tracebaility capabilities but only for those authorized entities with access to the necessary tracing/linking key associated with the created DAA Key.

Table 6.1: *CONNECT* Secure Life-cycle Management Functions leveraging Cryptographic Operations Provided by the underlying TCB.

All operations that leverage the "*perfectly secure*" cryptographic primitives, provided by the underlying TCB, will be modelled through axioms and will be further formally verified in the context of Deliverables D4.2 [14] and D4.3 [15]. These cover all *CONNECT* functional specifications that target the activities performed for **preparing a vehicle** (Section 4.2) so as to be part of a trustworthy CCAM continuum as well as those for **enabling the continuous trust assessment of the vehicle services** (Section 4.3) and the **re-establishment of trust** (Section 4.4) in case of a possible compromise or misbehavior. This does not include the specific tasks for protecting the execution of tasks and service workloads (instantiated in Gramminized execution environments) (Section 4.5) nor *CONNECT* trust extensions including state migration and software upgrade (Section 4.7) as these are offered directly from the underlying RoT and, thus, are part of the *idealized security model*. As aforementioned, the goal is to converge on a model that can allow for the reasoning about the security and privacy offered by the *CONNECT* secure life-cycle management protocols without being bogged down by the intricacies of various crypto-related primitives of the underlying trust anchor - a capability that may vary in different platforms featuring different type of RoTs. Considering the agnostic nature of *CONNECT* to specific types of secure elements, this type of formal verification provides adequate flexibility without losing the ability to express and verify fine-grained security characteristics. On the downside, such an idealized approach lacks in capturing potential vulnerabilities stemming from the internal cryptography, offered by various secure elements [24], nor it can identify timing attacks, as those presented in. However, this is something well studied in the literature, when it comes to the non-perfect secrecy of the TEE's crypto primitives [30], thus, in what follows we opted for focusing on idealized models capturing the operational characteristics of real-world automotive applications.

6.1 Language for Modelling *CONNECT* Secure Life-cycle Management Protocols with RoT

A number of security requirements need to be met in order to establish and maintain strong guarantees that will set the basis for assessing the trustworthiness of the nodes but also the data flows comprising the *CONNECT* ecosystem. The core security requirement, on top of which we are building the entire *CONNECT* framework, is that each critical software component or device has a **trustworthy RoT**. Inevitably, this requirement implies a set of trust assumptions that need to be made. Based on a valid RoT, we are able to extend those requirements needed for building a valid *TCB* which allows a host to measure and report (i.e., attest to) its state. In addition, we analyze the necessary requirements with respect to the secure enrollment of all the software applications comprising the *CONNECT* architecture.

Concerning the *TEE-GSE*, we identify those requirements needed for the secure issuance of all the verifiable evidence (*VCs* and *VPs*) that will be managed by each component. Specifically, VCs enable a device/component to manage and selectively disclose attributes depicting specific device characteristics. For instance, if an issuing component (e.g., OEM) verifies the integration of secure bootup mechanism in a device, then this constitutes an attribute as part of the issued VC based on which the device at any point in time can generate a proof on whether the output of this secure bootup process was correct or not. Which, in turn, can translate to evidence on the **design-time integrity** of the target device. Furthermore, all attestation enablers (provided by the *TEE-GSE*) need to satisfy the following conditions:

- **Completeness.** If an honest Prover makes a claim (e.g., regarding the correctness of its operational state), an honest Verifier (i.e., one following the zero-knowledge protocol properly) will be convinced of this fact by the Prover.
- **Soundness.** If the statement made by the Prover is false, no malicious Prover can convince an honest Verifier that the statement is true, except for a negligible probability (soundness error).
- **Zero-Knwoledge.** If the statement made by the Prover is true, no Verifier is required to learn anything additional other than the fact that the statement is true. In other words, knowledge of the statement is sufficient.

In parallel, similar requirements are also specified for the *CONNECT* containers that are not part of the *TEE-GSE* (e.g., the *MBD* and the *TAF*). Finally, in the context of the in-vehicle architecture, it is also essential to capture the prerequisites for the secure on-boarding of all the edge devices with the in-vehicle computer.

The formalization of the aforementioned security requirements can be achieved using diverse modeling languages. In *CONNECT*, we mainly employ a combination of **predicate-** and **algebra-based** languages. Specifically, predicate-based languages are used in order to model the assumptions and conditions that have to be valid in order to ensure the integrity of the produced trustworthiness evidence (in a verifiable manned). Essentially, predicates serve the modelling of the internal cryptographic operations of the underlying *CONNECT* TCB which is part of the "*isealized security model*". On the other hand, axioms depict those properties that need to hold for enabling the correct operation of the *CONNECT* TAF based on the trusted execution of *CONNECT*-equipped devices, based on both behavioral properties and low-level concrete properties about the entities' configuration and execution (to be considered as trust sources during the trust quantification of the target device). This is summarized in the following concrete definitions:

- **Security Predicates**, that depict the assumptions in a format that pairs a formally defined word with its meaning in terms of security requirements,
- **Axioms**, that depict the security claims, that should be present for *CONNECT* to guarantee that the security requirements are fulfilled.

Based on the use of such flexible predicate-based languages, the trust models presented in the following sections are split into three components: (i) the predicates which are essentially the "words" of the language, (ii) the core axioms which define how these predicates fit together to produce meaningful security requirements, and (iii) the complex axioms which combine predicates and core axioms showcasing how the *CONNECT* secureity life-cycle management functionalities are derived. For example, when we refer to a complex axiom concerning the secure creation of verifiable evidence for the correctness of a device, this can be achieved by fulfilling the expression consisting of the respective security predicates and core axioms.

The predicates are the dictionary of the trust model that lists each statement as a word and pairs it with its meaning. We split them into multiple categories depending on the layer that each model pertains to: either for an edge device (e.g., *ECU*, *Zonal controller (ZC)*), the underlying *TCB* of a host or an application running in the in-vehicle computer. The first category (Section 6.3) aims to give a set of predicates that represent the characteristics of a valid RoT, regardless of the technology that is used; if one precondition of these predicates fails then the system will be in an untrustworthy state and no guarantees on the security posture can be verified. Other predicates refer to the instantiation of protocols that enable the configuration and run-time attestation of hosts. For instance, in the context of run-time attestation (see Table 6.10), the $Policy_{IAM}(p, tcb, k_{AK})$ predicate translates to the fact the Attestation Key (k_{AK}) of the Prover device needs to have been created correctly during its secure onboarding with the IAM component which in turn fits under the generic $CryptoSafe_{RoT}$(r) system predicate for verifying the presence and validity of the underlying *RoT* of a *TCB* tcb to create the k_{AK} based on the key usage policies provided to the Prover during the secure on-boarding. There are also other layers (e.g., per *TEE-GSE* component) that represent intermediate states of the *CONNECT* data flow. The respective predicates in Tables 6.13 and 6.17 list the assumptions that need to be taken into consideration when designing the necessary cryptographic primitives to produce verifiable evidence to be sent outside the vehicle (e.g., the need to adopt the ETSI TS 102 941 specification [23] for producing IAM verifiable presentations as shown in the final axiom of Table 6.18).

Finally, we have to highlight that the security of the BIOS/Kernel of the system is considered as a prerequisite for a valid TCB. If the kernel is approached as a monolithic system, then it should be assumed that it is trusted in its whole since if even a single component diverges then the entire kernel is deemed untrusted. On the other hand, the kernel in the emerging edge- and cloud-computing applications where everything is considered as a service, can also be seen as a set of micro-services where only a specific set of them should be considered trusted in order for the entire system to be at a correct state. This reduction of the trusted code base of the kernel can introduce a chance for the tracing capabilities of the *CONNECT* to monitor exactly those functionalities. Besides these considerations, we will also assume that the underlying *RoT* itself satisfies memory-safety, type-safety and control-flow safety (see Tables 6.2 and 6.3). In other words: they are correct, secure and tamper-resistant.

6.2 Operational Essentials for Continuous Trust Assessment

6.2.1 Root-of-Trust for Storage, Reporting and Measurement

The security requirements presented in this section need to capture all the different devices and applications comprising the *CONNECT* ecosystem. The first requirement that we focus on pertain to the base characteristics of the underlying RoT (RoT) regardless of whether the device (hosting this RoT) is a powerful on-board unit (OBU), a Multi-access Edge Computing (*MEC*)-enabled service or a resource-limited (*ECU*) device.

By using a trustworthy *RoT*, a host is able to securely measure and attest to the correct state of part of its software stack. For a *RoT* to be considered trustworthy, a set of assumptions need to be made (Table 6.2). Specifically, as stated in Axiom 1 (Table 6.3), the assumptions are the following:

1. **Memory safety** is a crucial and desirable property for any device loaded with various software components. Its absence may lead to software bugs but most importantly exploitable vulnerabilities that will reduce the trust level of the device running the problematic software. In a nutshell, all accesses performed by loaded processes/services in the underlying memory map of the host device need to be “correct” in the sense that they respect the: (i) logical separation of program and data memory spaces, (ii) array boundaries of any data structures (thus, not allowing software-based attacks exploiting possible buffer overflows), and (iii) don’t access the memory region of another running process that they should not have access to. Memory-safety vulnerabilities can be detected in design-time with static code analysis techniques and during run-time with the well known tool Valgrind that is designed to identify memory leaks of an executable binary.

2. **Type safety** is closely related to memory safety as it also specifies a functionality that restricts how memory addresses are accessed in order to protect against common vulnerabilities that try to exploit shared data spaces (i.e., stack, heap, etc.). Type-safety is usually checked during design-time with most programming languages providing some degree of correctness (by default) paired with static code analysis tools that might catch some exceptions not covered by the language compiler (i.e., “fuzzing” tools or concolic execution

engines). However, type-safety can also be checked during run-time with the possibility of identifying issues that the static method did not identify

3. **Operational Correctness** is an intermediate abstraction of control-flow safety. It checks for the static state of the system and relies on the fact that a crucial part of the underlying kernel is in a trusted state. The operational-correctness aims to provide a more holistic view of the system by combining dynamic and static data collected by the *CONNECT* Attestation Toolkit in order to produce guarantees on the operational trust state of the system.

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

5. **Physical Security**: Regardless of the RoT that is in place, it is clear that physical attacks can compromise the functionality of the system. For instance, in the case of TPMs (i.e., discrete hardware chips that interconnect with the Low Pin Count (LPC) bus of a system through) it is possible for an adversary to spoof PCR values and steal sensitive data (e.g., the BitLocker disk encryption key), bypassing critical TPM trust guarantees. Consequently, the physical security of both the device as a whole and the actual pins that connect the TPM on the device motherboard should be carefully designed if the TPM is to be trusted.

Predicate	Description	Used in Axioms
PhySecure$_{RoT}(r)$	RoT r is physically secure.	Axiom 1
CryptoSafe$_{RoT}(r)$	RoT r uses secure cryptographic primitives.	Axiom 1
MemorySafe$_{RoT}(r)$	RoT r has memory safety.	Axiom 1
CorrectExecution(r, c)	RoT r correctly executes command c.	Axiom 1

Table 6.2: Predicates for Valid RoT.

Axiom: RoT

Axiom 1 $r, \forall c_i \ : \ \mathbf{Valid_{RoT}}(r, h) \ \Leftrightarrow \ \mathbf{PhySecure}_{RoT}(r) \wedge \mathbf{CryptoSafe}_{RoT}(d) \wedge \mathbf{MemorySafe}_{RoT}(r) \wedge \mathbf{CorrectExecution}(r, c_i)$

A RoT r of a host h is valid if and only if it correctly executes commands, is physically secure, has memory safety and uses secure cryptographic primitives.

Table 6.3: Axiom for Valid RoT.

6.2.2 Trusted Computing Base

In the context of *CONNECT*, the *TCB* of a host consists of all the firmware, hardware and software capabilities that allow for the enforcement of run-time security policies. As described in Chapter 3, the *CONNECT* TCB essentially comprises the tracing unit (exposing all TEE Device Interfaces) for providing the run-time system measurements capturing the current device's configurational and operational state; the key restriction usage policy enforcement module that interacts with the underlying RoT for checking whether the policies that have been set up for governing the usage of any cryptographic primitives are met or not; and part of the Attestation Agent responsible for the establishment of of a secure and authenticated channel with the host towards the provision of authentic traces. All these operations can be achieved in a trustworthy manner *if and only if* a valid *RoT* is in place (Table 6.3). The *RoT* shall be able to measure, store and report the results collected from the TCB's software components in a secure manner. The necessary predicates for defining a valid *TCB* are defined in Table 6.4. Specifically, as explicitly defined in Axiom 2 (Table 6.5), the *TCB* shall be able to securely measure - through a *RoT* for measurement - any software function (irrespectively of where it is running - "*untrusted*" host or part of the "*TCB*") with the support of the underlying tracing capabilities and TDIS as exposed by the *CONNECT TCB*. In parallel, it is of paramount importance that these measurements are securely stored within a *RoT* for Storage. This ensures that all attestation measurements and all cryptographic keys cannot be tampered with. Finally, the *TCB* shall be able to report on the measurements - through a *RoT* for reporting. The secure reporting of the measurements ranges from the reporting of the secure boot results up to the transmission of run-time traces, reported in a fresh (i.e., avoid replay attacks) and verifiable manner (e.g., digitally signed with an Attestation Key).

Predicate	Description	Used in Axioms
SecureDataExchange$(h1, h2)$	Two hosts h1 and h2 have the ability to securely establish a communication channel while preserving the exchanged data confidentiality and integrity.	Axiom 3, 7, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20
SafeBound$(k1, k2)$	A cryptographic key k_1 is safely bound to another key $k2$. Alternatively, k_1 can be used only by the entity that owns and can securely use key k_2.	Axiom 3, 6
StorageRoT(r, t)	RoT r is part of TCB t of a host and securely stores information that can not be tampered with.	Axiom 2
MeasurementRoT(r, t)	RoT r is part of TCB t of a host and securely measures the state of the TCB.	Axiom 2
ReportRoT(r, t)	RoT r is part of TCB t of a host and securely reports the collected measurements to the requested party.	Axiom 2

Table 6.4: Predicates for Valid Trusted Computing Base.

Axiom: Trusted Computing Base

Axiom 2 $t, h, r_S, r_M, r_R \; : \; \mathbf{Valid_{TCB}}(t,h) \;\Leftrightarrow\; \mathbf{Valid}_{RoT}(r_S,h) \wedge \mathbf{StorageRoT}(r_S,t) \wedge \mathbf{Valid}_{RoT}(r_M,h) \wedge \mathbf{MeasurementRoT}(r_M,t) \wedge \mathbf{Valid}_{RoT}(r_R,h) \wedge \mathbf{ReportRoT}(r_R,t)$

A host, h, has a TCB, t, that is valid if and only if it consists of a valid *RoT* for measurement, r_M, that securely measures the TCB software components' state, a valid *RoT* for storage, r_S, that securely stores the results and a valid *RoT* for reporting, r_R, that securely reports the measurements upon arrival of a challenge.

Table 6.5: Axiom for Static Attestation.

6.3 Static and Run-time Attestation Capabilities

One of the core predicates defined for the Axioms that capture the properties to be verified for the correctness of the *TCB* (Table 6.5), is related to the secure reporting of measurements that have been securely measured and stored within a valid *RoT* element. In the context of *CONNECT* there are various ways of attesting to static characteristics of a host or even run-time characteristics of an application running within a host. In fact, the number and type of attestation measurements that need to be collected - and the complexity of the attestation schemes to be enforced - are intrinsically linked with the required trust level associated with a device or software component: *the higher the required trust level we want to achieve for an entity, which translates to the trust appraisal based on a multitude of trust properties beyond integrity (including resilience, robustness, safety, etc. from the possible set of trust properties as defined in D3.1 [11]), the more rich attestation evidence we want to collect*. Attestation measurements can be static (i.e., such evidence can be the hash of the binary that is running as part of the *TCB*, the software version of the application) or they can refer to logs related to the run-time of an application (e.g., collecting the traces dynamically during the execution of an application). The security requirements for each of the two types of attestation evidence is analyzed below.

6.3.1 Zero-Touch Configuration Integrity Verification

The process of static attestation refers to the collection of measurements that is related to configuration related information. On start-up of a host, these measurements are performed by the boot code. The measurements are securely stored in the RoT and are not altered during runtime. When challenged by a Verifier during run-time, the attested host needs to provide the static attestation evidence in a fresh and verifiable manner. To address the first prerequisite, the Verifier (i.e., the challenger) requests the attestation evidence by securely creating and providing a nonce value. The Prover needs to use an Attestation Key that is securely created, stored and used to provide a verifiable evidence over the attestation measurements. The results are sent to the Verifier which are validated against the expected values that characterize the expected values that the Prover should report. In the complex and diverse ecosystem of *CONNECT*, the Axiom should capture hosts with different attestation capabilities as well as different supported cryptographic operations. For instance, a host might have a Trusted Execution Environment that supports asymmetric operations but at the same time there can be a low-end host supporting a Hardware Secure Module (HSM) with only symmetric cryptographic capabilities.

The formal definition of the Axioms in Table 6.7 requires the set of predicates defined in Table 6.6. The first axiom refers to the secure creation and secure storage of the necessary Attestation Keys to the host that needs to provide attestation evidence. The issuer of the Attestation Keys (Attestation Server instantiated as part of the Vehicle Manage) needs to ensure that the Prover host contains a valid TCB (Table 6.5) and, thus, provide a digital certificate confirming the secure construction of the HW-basd Key binded to the correct key restriction usage policies. In addition, to ensure the association of Attestation Keys with a valid *RoT* (Table 6.3), the keys need to be securely bound (been a child) to the endorsement key of the *RoT*. This implies that only the holder of the endorsement key (i.e., only that particular *RoT*) is able to use the Attestation Key to perform attestation procedures. Given the correct and secure creation and storage of Attestation Keys within the RoT of a host, a host is able to securely provide evidence attesting to its correct configuration and integrity if and only if the host is able to securely use - through its *RoT* - the Attestation Keys to provide a verifiable evidence of an audit value that a Verifier can compare with an expected (golden) value that signals the correct and expected state of the host. It is worth mentioning here that depending on the capabilities of a *RoT* element, the verifiable evidence over the audit value can be a digital signature using an asymmetric Attestation Key or a hash-based message authentication code (HMAC) using a symmetric Attestation Key.

Predicate	Description	Used in Axioms
SecurelyStored(x, t)	Blob of data x is securely stored in RoT of TCB t.	Axiom 3
Certified$_I(x)$	Entity (i.e., data, application, device) x is certified by an issuer entity I. In this case, I provides a verifiable credential of certifiability to the host acting as a certification of the validity of x. In addition, I is a trusted entity that has the capability to validate the correctness of x.	Axiom 3, 8, 9, 10, 18
AuditDigest(h, d)	A host h can securely compute measurements over its state and calculate an audit digest d representing its state.	Axiom 4
VerifiableEvidenceUsingKey(h, x, k)	A host h can securely provide a verifiable evidence for a blob of data x using key k.	Axiom 4, 7

Table 6.6: Predicates for Static Attestation.

Axioms: Reporting of static attestation measurements

Axiom 3 $t, h, I, k_{AK}, k_{EK} : \mathbf{SecureCreationOfAttestationKey}(k_{AK}, h) \Leftrightarrow \mathbf{Valid}_{TCB}(t, h) \wedge \mathbf{SecureDataExchange}(h, I) \wedge \mathbf{SecurelyStored}(k_{AK}, t) \wedge \mathbf{SecurelyStored}(k_{EK}, t) \wedge \mathbf{SafeBound}(k_{AK}, k_{EK}) \wedge \mathbf{Certified}_I(k_{AK})$

A host h can securely obtain and store Attestation Keys k_{AK} if and only if h has a valid TCB t, there is confidential and secure communication with a trusted issuer I (i.e., IAM), k_{AK} are safely bound to the endorsement key k_{EK} of the RoT and I has generated and sent verifiable evidence to h certifying k_{AK} to be used in attestation procedures.

Axiom 4 $k_{AK}, h, d : \mathbf{SecureStaticAttestationEvidence}(k_{AK}, h) \Leftrightarrow \mathbf{SecureCreationOfAttestationKey}(k_{AK}, h) \wedge \mathbf{AuditDigest}(h, d) \wedge \mathbf{VerifiableEvidenceUsingKey}(h, d, k_{AK})$

A host h can securely produce static attestation evidence (e.g., Configuration Integrity Verification (CIV) evidence) using a key k_{AK} if and only if h can securely create Attestation Keys and provide a verifiable evidence using its Attestation Key over an audit value d with respect to its configuration and integrity of the host.

Table 6.7: Axioms for Zero-Touch Configuration Integrity Verification.

6.3.2 Run-time Attestation

Static attestation measurements enable a host to securely attest to its correct configuration and integrity of instantiated/loaded software. However, in many cases we want to ensure that the attested host is in a correct state during the execution of its application. To achieve that, it is necessary to design run-time attestation mechanisms which enable the continuous monitoring of system critical applications and devices running within the *CONNECT* ecosystem. It should be highlighted that the type of run-time attestation to be adopted for each type of *TCB* is highly dependent on its capabilities to dynamically measure the state of the host in a secure fashion. The development of run-time attestation procedures require the existence of a dynamic tracer software application which is part of a host's *TCB*. According to Axiom 2 (Table 6.5), since the tracer is part of a valid *TCB*, it is required that the tracer software is statically attested along with the other *TCB* software components during the boot up process and is equipped with a pre-established key so as to guarantee that only authentic traces can be processed by the Attestation Agent running in an isolated environment (protect against attackers trying to impersonate a valid tracing module which will violate the correctness of the entire attestation process).

In the axioms presented in Table 6.9, there is a list of security requirements that describe the prerequisites for a valid tracer application to securely use Attestation Keys in order to participate in a run-time attestation procedure (Table 6.8). First of all, for a tracer application to securely collect and report attestation evidence, it needs to be part of a valid *TCB*. This enables the secure creation of (static) attestation evidence for the tracer application itself. Secondly, a tracer key needs to be securely produced and stored in the underlying RoT to allow only the tracer application to provide verifiable evidence for the collected attestation data. Assuming that such a tracer application resides in a valid *TCB*, the second axiom analyses the requirement for the tracer application to securely compute attestation evidence using a securely produced Attestation Key (Axiom 3 in Table 6.7). This introduces the requirement that the Attestation Key used for the run-time attestation operations need to be associated (i.e., safely bound) with the tracer's key. According to the final Axiom 7 of Table 6.9 if the host is able to securely prove to a Verifier that these key restriction requirements are enforced using a key restriction usage policy then the attestation evidence consists of a digital signature over a nonce value that is sent by the Verifier. This allows a Verifier to trust that the local attestation is securely performed by the Prover by validating the signature. If a host does not support such a local attestation scheme, then the challenger should provide - along with a nonce - the type of attestation evidence (i.e., run-time traces) that the Prover should provide in a verifiable manner. The Prover - through the tracer - securely computes the attestation measurements, constructs the verifiable evidence and provides the result back to the Verifier. Finally, the Verifier validates the results using the corresponding golden values describing the expected responses that the Prover should provide. Throughout the challenge-response protocol data confidentiality and integrity is assumed (e.g., the communication between a Verifier host and a Prover host is securely established using the necessary cryptographic credentials).

Finally, even though we focus on the security requirements of a valid tracer software application in the context of *CONNECT* there is one critical functional requirement that should also be met. Given the fact that the ECU devices constitute far edge devices within the *CONNECT* ecosystem, it is imperative to take into consideration the overhead that such a tracer application poses against the intended ECU application. In other words, it is essential to safeguard the operation profile of the ECU devices that perform safety critical processing of kinematic data under strict real-time constraints. Concerning the run-time tracing capabilities there are two main approaches that with different characteristics, namely the intrusive and non-intrusive tracing. In contrast to an intrusive tracing strategy, a non-intrusive tracer does not change the intrinsic timing behavior of the host under attestation. External trace probes or on-chip tracing capabilities enable the monitoring of a device during run-time without consuming resources from the device or affecting the execution time of the application under inspection. The aforementioned predicate of a non-intrusive tracer is also recorded in Table 6.8.

Predicate	**Description**	**Used in Axioms**
KeySecurelyProduced(k, t)	Key k has been securely generated and stored in RoT of a TCB t.	Axiom 5, 8
ApplicationInHost(a, h)	Application a is running on host h.	Axiom 9, 10, 18
ApplicationInTCB(a, t, h)	Application a is part of the TCB t of a host h.	Axiom 5, 8, 9, 18
ChallengeByHost(h)	A Challenge (e.g., including a nonce value) has been securely created and transmitted by host h.	Axiom 7
NonIntrusiveTracer(a, k_{TRACER})	A tracer application a is collecting traces in a non-intrusive manner using a dedicated key k_{TRACER}.	Axiom 5

Table 6.8: Predicates for Run-time Attestation.

Axioms: Run-time Attestation

Axiom 5 $tracer, t, h, k_{AK}, k_{TRACER}$: $\mathbf{AttestableTracerInHost}(tracer, h) \Leftrightarrow \mathbf{Valid}_{TCB}(t, h) \wedge \mathbf{ApplicationInTCB}(tracer, t, h) \wedge \mathbf{NonIntrusiveTracer}(tracer, k_{TRACER}) \wedge \mathbf{SecureCreationOfAttestationKey}(k_{AK}, h) \wedge \mathbf{KeySecurelyProduced}(k_{TRACER}, t)$

A tracer application $tracer$ running in a host is attestable if and only if $tracer$ is part of a valid TCB t running in the host h, h can produce static attestation for t, and a tracer key k_{TRACER} is securely generated and stored in the RoT.

Axiom 6 $tracer, h, k_{AK}, k_{TRACER}$: $\mathbf{TrustworthyTraces}(tracer, h) \Leftrightarrow \mathbf{AttestableTracerInHost}(tracer, h) \wedge \mathbf{SafeBound}(k_{AK}, k_{TRACER})$

A tracer application $tracer$ running in a host h can securely produce run-time attestation evidence if and only if the tracer application is attestable and the Attestation Key k_{AK} used in scope of run-time attestation operations is securely bound to the tracer key k_{TRACER}.

Axiom 7 $tracer, k_{AK}, h_1, h_2, d_{audit}$: $\mathbf{SecureRun-timeAttestationEvidence}(k_{AK}, h_1) \Leftrightarrow \mathbf{TrustworthyTraces}(tracer, h_1) \wedge \mathbf{SecureDataExchange}(h_1, h_2) \wedge (\mathbf{AuditDigest}(h_1, d_{audit}) \vee \mathbf{VerifiableEvidenceUsingKey}(h_1, \mathbf{ChallengeByHost}(h_2), k_{AK}))$

A host h_1 can securely produce run-time attestation evidence to a Verifier host h_2 using a key k_{AK} if and only if h_1 has a tracer $tracer$ that can securely produce run-time attestation evidence on top of a secure channel established between the two entities. Depending on the capabilities of h_1, the run-time attestation evidence can be:

- a signature of a challenge/nonce using the Attestation Key k_{AK} provided that the host has local attestation capabilities, or
- an audit digest d_{audit} that can be used by a Verifier to validate the correctness of the values by comparing them to a set of expected values.

Table 6.9: Axioms for Run-time Attestation.

6.4 Instantiating the *CONNECT* TCB

As aforementioned, the architecture of the *CONNECT* framework positions the various building blocks across the all layers of the CCAM continuum: from the in-vehicle computer and devices, to the *MEC* and cloud infrastructure. In all cases, there are security requirements that are pivotal for the *CONNECT* *TEE-GSE* to be instantiated and to securely perform its functions. The requirements are built on top of the fact that each component of the *CONNECT* TEE Guard is running on hosts that have their own valid *TCB* with attestation capabilities (forming a notion of a Distributed RoT). Even in the case of the in-vehicle architecture (Chapter 3) where all the *TEE-GSE* applications and other services are running within a single in-vehicle computer, each of the safety critical components is running in a confidential manner with its own *TCB*.

For the secure enrollment of any of the *TEE-GSE* components or other *CONNECT*-related services (i.e., *TAF*, *MBD*) as well as the secure on-boarding of *ECU* devices (to be examined in Section 6.8) it is essential that their state is validated by a trusted external entity. This entity is going to verify the correct bootup of the application or device that is to be enrolled. In addition, for the TEE-enabled components it shall be able to verify that the necessary enclaves have been launched correctly (e.g., with the expected manifest files) by leveraging the Grammine attestation extension capabilities. This role is addressed by the Attestation Server *Attestation Server (AS)* component. **This entity can be launched within an in-vehicle computer or to the backend, as part of an OEM's vehicle management capabilities.** In some cases, the *AS* component may need to acquire a set of reference values to validate that an enrolling entity is at an expected state. Depending on the scenario, these reference values can be fetched from an *IAM* application that is securely enrolled (see Section 6.4.1) within a topology or directly from the OEM services residing in the backend. For instance, such reference values can be the expected software version of an application or the contents of the manifest file for an *Enclave* responsible for running one of the *TEE-GSE* components. Upon successful validation of an entity's state, the AS component issues a *VC* which certifies that the entity (i.e., software component or device) has been successfully enrolled. This is equivalent to verifying the correct boot up of the enrolling entity and the fact that it is running a certified application. As shown in Axioms 9 (Table 6.12) and 18 (Table 6.21) this *VC* is necessary for an entity to acquire all the necessary application-related and *CONNECT*-related keys by the *IAM*.

6.4.1 Secure Enrollment of CCAM Services

Chapter 3 presents the core architecture of the *CONNECT* Trusted Computing Base. The first component that needs to be initialized is the *IAM* component which will be responsible (among other things) for the secure enrollment and on-boarding of all in-vehicle sensors as well as the CCAM services instantiated within each of these devices. Towards this direction, the *IAM* needs to securely establish a connection with the backend OEM server allowing the continuous interaction and secure exchange of information related to the types of application and services comprising the software stack of each in-vehicle sensor. Essentially, the *IAM* receives *application-centric VCs* certifying the deployed applications and including information such as type of OS an software version instantiated per in-vehicle sensors, and the reference value of the sensor's nominal configuration and behavioral state. In addition to that, the *IAM* is also responsible for the ratification of the secure enrollment of all *TEE-GSE* components.

The predicates required for the definition of the secure enrollment axioms are listed in Table 6.10. The Axiom in Table 6.11 describes the requirement for the secure launch of an *IAM* component within a host (i.e., In-vehicle computer or *MEC*). Specifically, for an *IAM* software component to be securely enrolled within a host, it needs to be running in an isolated environment . Within this isolated environment, the *IAM* needs to be part of a valid *TCB*. This ensures that during bootup of the host, the *IAM* application is securely measured by the underlying *RoT*, has securely attested to its correct configuration through the secure bootup process. This information is shared with the *AS* component which compares the attributes against the respective expected values provided by the *OEM* for an *IAM* application. Upon validation, the *AS* component issues a *VC* which certifies that the application has successfully enrolled. This *VC* allows the *IAM* application to connect with the backend application and fetch all the necessary reference values that are expected to be reported by any other service that wants to securely enroll or device that wants to securely onboard. These reference values are transmitted by the *OEM* in a form of a *VC* as well. Finally, a master key (e.g., in the case of the in-vehicle computer the master key is referring to the vehicle master key) needs to be securely generated and stored within the RoT of the *IAM*'s host and the *IAM* has securely initialised the *CONNECT* key management system. The *CONNECT* key management system within the *IAM* will be responsible for storing all of the required keys which are derived through key derivation functions from the master key. The *IAM* will manage these key's distribution to other components and devices as the are on-boarded.

Predicate	Description	Used in Axioms
KeySecurelyLoaded$_{IAM}(k, t, h)$	Key k has been securely issued by IAM and is securely loaded into the valid TCB t of host h.	Axiom 9, 14, 15, 16, 17, 18
ApplicationHasAccessToKey$_{IAM}(a, k)$	Application a has access to key k issued by the IAM component.	Axiom 10
SecureContainer(s, h)	Secure Container s is running in host h. This provides an isolated environment for an application to run with its own properties and is capable to provide guarantees on its design and run-time correct state.	Axiom 9

Table 6.10: Predicates for Secure Enrollment.

Axiom: IAM Secure Enrollment

Axiom 8 $iam, t, k_{VMK}, s, h : \mathbf{SecureEnrollment}_{IAM}(iam) \Leftrightarrow \mathbf{SecureContainer}(s, h) \land \mathbf{ApplicationInTCB}(iam, t, s) \land \mathbf{Certified}_{AS}(iam) \land \mathbf{Certified}_{OEM}(iam) \land \mathbf{KeySecurelyProduced}(k_{VMK}, t)$

IAM application iam is securely enrolled if and only if host h is able to spawn a secure container s, on which iam is running, equipped with a valid TCB t; iam has been attested to its correct configuration and it is certified by the AS component; has been certified by the OEM by obtaining a *application-centric VCs* with the expected (reference) values for for all services and devices as part of the target vehicle's E/E topology; has securely stored the Vehicle Master key k_{VMK} in its RoT; and has correctly instantiated the overall *CONNECT* key management system within its RoT (placeholder for the verification of the correct creation of Attestation Keys per TCB instantiated in each one of the other *TEE-GSE* components and in-vehicle sensors).

Table 6.11: Axiom for the secure enrollment of IAM.

For the remaining of the software components comprising the *CONNECT TEE-GSE*, they need to enrol in a secure manner including the connection with the *IAM* component in order to acquire the necessary cryptographic material. This implies that there needs to be an *IAM* component that is already securely enrolled (Axiom 6.11) prior to the launch of all other *CONNECT*-related security components that need to run within a secure container. Once the boot up of entire TCB has been achieved, the Trusted Third Party Attestation Server component needs to ensure that the software component is at a correct state. All the required reference values can be retrieved through the *IAM* component. With the issuance of the AS *VC* the software component can demonstrate to the *IAM* that is securely enrolled. This allows the IAM to securely transmit all application-related and *CONNECT*related keys to the enrolled software component. These keys shall be securely stored in the *RoT* of the *TCB* that the software component is running. Concerning the *CONNECT*related keys, there needs to be a policy dictating the conditions based on which they keys can be used.

Specifically, the *IAM* may use the reference values in order to enforce a key restriction usage policy to the enrolled component that allows us to dictate the conditions (e.g., configuration state of the component) based on which the respective *RoT* element can use a specific key.

On the contrary, within the *CONNECT* ecosystem there are also software components running in a host that are not part of its trusted software stack. For these software components, the security requirements are limited to ensuring the data confidentiality and integrity during data exchange. Such cryptographic material is received during the secure enrollment phase.

Both axioms for the secure enrollment of software components - either residing in and outside of the TCB of a host- is presented in Axiom 6.12.

Axioms: Secure Enrollment of Software Components

Axiom 9 $a, iam, s, t, h, p, k_{APP}, k_{AK} \subseteq k_{\text{CONNECT}}$: $\textbf{SecureEnrollment}_{TCB-SW}(a) \Leftrightarrow \textbf{SecureEnrollment}(iam) \wedge \textbf{SecureContainer}(s, h) \wedge \textbf{Certified}_{AS}(a) \wedge \textbf{ApplicationInTCB}(a, t, s) \wedge \textbf{SecureDataExchange}(s, iam) \wedge \textbf{KeySecurelyLoaded}_{IAM}(\{k_{APP}, k_{\text{CONNECT}}\}, t, h) \wedge \textbf{KeyRestrictionUsagePolicy}(p, t, k_{\text{CONNECT}})$

An application a is successfully enrolled to the IAM component iam if and only if iam is securely enrolled; a is certified by the AS component concerning its correct configuration (during bootup); is part of a Trusted Excution Environment within a secure container s meaning that it can be securely measured and reported at any point in time; has established secure communication with iam and has securely established the required application-related keys k_{APP} and *CONNECT*-related keys k_{CONNECT}. These keys need to be securely stored in the RoT of t. Regarding the latter set of keys, their usage is controlled by a key restriction usage policy which allows the use of the specific keys under the condition that a is at a correct state (extracted from the application-centric VCs the IAM acquired from the OEM). This $KeyRestrictionUsagePolicy$ is not expressed as a predicate since it is the foundation of *CONNECT*'s novel local attestation capabilities and will be formally verified. The related software components that need to meet this axiom are the following: AIV, TCH, TAF, MD.

Table 6.12: Axioms describing the security requirements for the secure enrollment of software components.

6.5 Modelling the TEEguard Extensions

This section focuses on the security requirements concerning the creation of all the verifiable evidence produced by the *TEE-GSE* applications. These requirements are essential for the dynamic trust assessment of the monitored infrastructure. In addition, they enable the inclusion of the trustworthiness information within the V2X communication. For the information transmitted outside of the vehicle, we need to ensure that the identity of the vehicle - or of its services - is not compromised. To this end, it is essential that the verifiable evidence included in the T-CAM/T-CPM messages cannot be linked back to a specific entity. This is achieved by assuming that each *TEE-GSE* component is able to acquire an anonymized credential, associated with its keys used for producing its verifiable credentials or presentations. This anonymized credential is issued by a trusted privacy certification authority (PrivacyCA). This allows each component to produce verifiable evidence without enabling this verifiable evidence to be associated with the identity of the vehicle (table 6.13). In fact, the predicate presented in the table below is also essential for the creation of any other verifiable credential that is going to be used to produce the IAM verifiable presentation to be included in a T-CAM/T-CPM message. Thus, the *AnonymizedCredential* predicate is also used for the secure acquisition of the credentials for the keys used by the TAF, the MD and the TCH when creating their *VCs* (Subsection 6.7 and 6.8 respectively).

Predicate	**Description**	**Used in Axioms**
AnonymizedCredential$_{PrivacyCA}(k)$	Key k has an anonymized credential associated with it. This credential is issued and transmitted securely by a PrivacyCA and it is securely stored along with k. This allows the signing entity to verifiable credentials and presentations in a privacy enhancing manner.	Axiom 14, 16, 17

Table 6.13: Formal Predicates for *CONNECT* functionalities.

6.5.1 Attestation and Integrity Verification

A set of security requirements are defined in order to enable all the functionalities supported by the *AIV* component. The AIV component - which is part of the *TEE-GSE* - is responsible for collecting attestation evidence from the securely on-boarded devices, computing the attestation results and forwarding them to the *TAF* and *TCH*. If an attestation process fails, then the collected evidence should be reported to a DLT in order to be available for analysis by the OEM and other regulatory authorities.

First and foremost, as stated in the first Axiom of 6.15, a software component can attest to its correct state if and only if it has been securely enrolled (6.12) and has successfully created its Attestation Key. Thus, it shows that the key restriction usage policy has been successfully enforced . This is equivalent to the component being able to provide run-time evidence about it being at a correct state. This Axiom is valid for only those components that have the necessary resources to instantiate a valid TCB. As was also aforementioned, not the entire software stack is subject to attestation but only those safety-critical components pertaining to the verification of the data collection and management software (especially for kinematic data).

Based on that, the AIV component can securely construct its attestation report for a set of devices if and only if the AIV component is itself in a correct state, the devices are securely on-board and data confidentiality and integrity is achieved throughout their communication. Finally, in case of a failed attestation, the AIV needs to be able to securely publish the failed attestation evidence to the *CONNECT* DLT. The failed attestation evidence acts as a diagnostic log to allow the device manufacturers to investigate and identify the reason for the error. According to the third axiom of 6.15 the AIV component needs to be securely enrolled with the IAM component, needs to prove that it is at a correct state (i.e., through the use of its local attestation scheme) and needs to provide the failed attestation evidence to the DLT by specifying the conditions based on which the data can be accessed (table 6.14). For example, this can be achieved through the issuance of an Attribute-Based Encryption Key - by the IAM component - that can be used by the AIV in order to enforce restrictions on who has access to the failed attestation evidence (i.e., only the consumers with the correct attributes are able to decrypt the content published on the *CONNECT DLT*). Data confidentiality and integrity is assumed throughout the communication between the AIV and the *CONNECT DLT*.

Predicate	Description	Used in Axioms
EnforceAccessControlPolicy(a, t, k)	Application a securely uses - through its TCB t - its key k to provide an access control policy over a piece of information.	Axiom 13

Table 6.14: Formal Predicate for the secure enforcement of access control policies dictating the conditions to access a resource.

Axiom: AIV core functionalities

Axiom 10 s, h, k_{AK} : $\mathbf{CorrectState}(s) \Leftrightarrow \mathbf{SecureEnrollment}_{TCB-SW}(s) \wedge \mathbf{SecureRun-timeAttestationEvidence}(k_{AK}, h)$

A software component s that is part of the *CONNECT* TCB is at a correct state if and only if s is securely enrolled, and can securely produce run-time attestation evidence through the use of its local attestation scheme.

Axiom 11 $a, \forall d \in DO, h$: $\mathbf{SecurelyProduceAttestationReport}(a, DO) \Leftrightarrow \mathbf{CorrectState}(a) \wedge \mathbf{SecureOnBoarding}(\mathbf{d}) \wedge \mathbf{SecureDataExchange}(h, d)$

The AIV component a can securely produce the attestation results for a set of devices comprising a data object DO (i.e., the set of devices is decided by the TAF through the request for evidence process. In fact, TAF might request evidence for a subset of DO) if and only if a is an application that is at a correct state, each device d in RFE is securely on-board and data integrity and confidentiality is achieved throughout the attestation procedure.

Axiom 12 $a, h, dlt, t, k_{\text{CONNECT}}$: $\mathbf{SecurelyPublishEvidenceToDLT}_{AIV}(a) \Leftrightarrow \mathbf{CorrectState}(a) \wedge \mathbf{SecureDataExchange}(h, dlt) \wedge$
$\mathbf{EnforceAccessControlPolicy}(a, t, k_{\text{CONNECT}})$

The AIV component a can securely publish attestation evidence of failed attestation procedures to the *CONNECT* DLT dlt if and only if a is in a correct state, has data integrity and confidentiality during the communication with the *CONNECT* DLT and can provide the failed attestation evidence by enabling access control policies for the consumption of the data using the necessary key, k_{CONNECT}.

Table 6.15: Axioms for the Attestation Integrity Verification component.

6.5.2 Trustworthiness Claims Handler

The reports produced by the *AIV*, the *MBD*, and the *TAF*, holding trust-related information and embodied into *Verifiable Credentials* are securely forwarded to the *TCH* component - which is part of the *TEE-GSE*. Assuming that the *TCH* component is at a correct state, it will be able to validate and aggregate all this information (for a given data object), into a Verifiable Presentation for sharing with the other CCAM actors (prior to the establishment of a trust relationship) but in a privacy-preserving manner. Essentially, the detailed attestation results will be adequately abstracted by having the *TCH* verifying the attestation results and issuing new (group-based) attribute assertions aggregating the attestation results (for the same type of trust property) for all devices comprising a service graph chain. This will allow, for instance, the creation of a *vehicle integrity* attribute which will be "*true*" if and only if the CIV results from all devices are "*true*". Which will be further protected through the appropriate cryptographic primitives; i.e., group-based signatures and link tokens. Given that the keys used to produce the input *AIV*, *MBD* and *TAF* credentials are constrained by key restriction usage policies, the TCH is assured that the local attestation schemes have been successfully applied and the respective components are in a correct state. Finally, these *VC*s are used by the TCH to construct the *VP* (selectively) disclosing this trust-related information needed for the neighboring vehicles to calculate their own local trust opinions whereas the MEC-instantiated TAF can maintain an up-to-date composite view of the trust state of the entire CCAM continuum. Prior to their broadcasting, the *VP*s are sent to the IAM to be wrapped around a signature leveraging PKI-issued short-term anonymous credentials (pseudonyms) for enhanced anonymity. The complete definition is presented in Axiom 6.16.

Axiom: Trustworthiness Claims Handler security requirement

Axiom 13 $tch, DO, md, aiv, taf, t, h, k_{AC}$: $\mathbf{SecurelyProduceVP}_{TCH}(tch, DO) \Leftrightarrow \mathbf{CorrectState}(tch) \wedge \mathbf{SecurelyProduceAttestationReport}(aiv, DO) \wedge \mathbf{SecurelyProduceVC}_{MD}(md, DO)$
$\wedge \mathbf{SecurelyProduceVC}_{TAF}(taf, DO) \wedge \mathbf{SecureDataExchange}(tch, aiv) \wedge \mathbf{SecureDataExchange}(tch, md) \wedge \mathbf{SecureDataExchange}(tch, taf) \wedge \mathbf{KeySecurelyLoaded}(k_{AC}, t, h) \wedge \mathbf{AnonymizedCredential}(k_{AC})$

The TCH component tch securely produces its TCH Verifiable Presentation for a data object DO if and only if tch is at a correct state; AIV component aiv securely produces its attestation report for devices that are part of DO; and MD md and TAF taf component produce their Verifiable Credentials (including the Misbehavior Report and Trust Opinion, respectively) for DO. All this information shall be sent to tch securely based on which a Verifiable Presentation will be constructed selectively disclosing the trust-related information needed for the vehicle-wide trust quantification.

Table 6.16: Secure creation of TCH Verifiable Presentation.

6.5.3 Identity Authentication Management

The final requirement has to do with the way that the TCH *VP* is converted into privacy preserving trustworthiness information that accompanies the corresponding CAM/CPM messages. As initial step, the creation of the IAM verifiable presentation requires that the TCH has securely produced (axiom 6.16) and transmitted (secure exchange of data) its TCH *VP* to the IAM component which is also part of the *TEE-GSE*. According to the ETSI TS 102 941 specification [23], the IAM shall be able to securely obtain, store and use pseudonym credentials (i.e., pseudonyms) issued by a trusted Pseudonym Certification Authority (*Pseudonym Certification Authority (PCA)*) as specified in the specification. By adopting the Trust and Privacy Management specification by

ETSI, the transmission of V2X messages is performed in a way that preserves the anonymity of the involved vehicles while enhancing observability to improve the safety offered by the offered CCAM services. This trust assumption is captured in the predicate presented in table 6.17. Equivalently, this mechanism enables pseudonymity that protects the privacy of the vehicle while allowing accountability for the V2X information transmitted. Finally, unlinkability ensures that the transmission of similar V2X messages do not allow an adversary to link or correlate the messages with a specific vehicle. Axiom 6.18 illustrates the correct creation of the IAM Verifiable Presentation.

Predicate	Description	Used in Axioms
PseudonymCredential$_{PCA}(k, t, h)$	Host h has securely obtained a pseudonym certificate by a *PCA* for its key k residing in the RoT of the host's TCB t. The entire process for obtaining such a certificate is aligned with ETSI TS 102 941 specification [23].	Axiom 15

Table 6.17: Formal Predicate for the secure acquisition of authorization tickets as specified in the ETSI TS 102 941 specification.

Axiom: Identity Authentication Management security requirement

Axiom 14 $iam, DO, tch, t, h, k_{AC}$: $\mathbf{SecurelyProduceVP}_{IAM}(iam, DO) \Leftrightarrow \mathbf{CorrectState}(iam) \wedge \mathbf{SecurelyProduceVP}_{TCH}(tch, DO) \wedge \mathbf{SecureDataExchange}(iam, tch) \wedge \mathbf{KeySecurelyLoaded}(k_{PSEUDONYM}, t, h) \wedge \mathbf{PseudonymCredentia}_{PCA}(k_{PSEUDONYM}, t, h)$

The IAM component iam securely produces its Verifiable Presentation for the trustworthiness data characterising a data object DO if and only if iam is at a correct state, TCH is securely producing and transmitting its verifiable presentation for DO to iam and iam has securely obtained, stored and used a pseudonym credential to construct its verifiable presentation.

Table 6.18: Secure creation of IAM Verifiable Presentation.

6.6 Modelling the Misbehavior Detection Component

The secure creation of the Misbehavior Report (again encoded as part of a *VC*)) outputted by the Misbehavior Detection component involves the consumption of raw data sent by the devices through the Facility Layer. By default, the Facility Layer - whose main purpose is to forward in-vehicle sensor (kinematic) data to the applications - doesn't provide an attack surface for an adversary since all relayed data are signed by the origin ECUs or Zonal Controllers (integrity preservation). However, as this module constitutes the *data gateway* between the in-vehicle sensors and the Vehicle Manager, it is periodically attested by each one of the *TEE-GSE* components when receiving such kinematic data. *By default the Facility Layer is not part of* CONNECT*'s TCB so as to avoid the additional overhead that this might pose in the relaying of kinematic data that* *needs to be rather efficient in such safety-critical applications*. As shown in Axiom 6.19, there needs to be a secure communication between the Facility Layer and the MD component to ensure the data integrity and confidentiality. This requires that both the MD component and the Facility Layer need to be securely enrolled with the IAM. Finally, the MD component needs to be at a correct state in order to be able to use the necessary anonymized credential to generate the Misbehavior report verifiable credential. The anonymized credential is necessary to ensure that the Misbehavior report can be shared with other V2X services (i.e., outside of the vehicle) without compromising the identity of the MD component - and the vehicle in general.

Axiom: Misbehavior Detection security requirement

Axiom 15 $md, fl, t, h, k_{AC}, DO : \mathbf{SecurelyProduceVC}_{MD}(md, DO) \Leftrightarrow \mathbf{CorrectState}(md) \wedge \mathbf{SecureEnrollment}(fl) \wedge \mathbf{KeySecurelyLoaded}(k_{AC}, t, h) \wedge \mathbf{AnonymizedCredential}(k_{AC})$

The MD component md securely produces its misbehavior report verifiable credential - i.e., assessing a data object DO - if and only if md is at a correct state it securely receives DO-related data from the Facility Layer component fl which is securely enrolled and md component has securely obtained, stored and used an anonymized credential for k_{AC} to generate the misbehavior report verifiable credential.

Table 6.19: Secure creation of Misbehavior Report Verifiable Credential.

6.7 Modelling the Trust Assessment Framework

The *TAF* requires input from multiple trust sources in order to compute its verifiable credential. In axiom 6.20 we specify the requirements assuming that trustworthiness evidence are coming explicitly from the *AIV* and the *MBD* components but it can be easily extended to support additional Trust Sources (e.g., trustworthiness evidence coming from an Intrusion Detection system). That being said, the TAF component securely produces its Trust Opinion Verifiable Credential if and only if the *AIV* is at a correct state and it securely transmits its attestation report. In parallel, the *MBD* shall be also at a correct state and shall securely send its own misbehavior report. Last but not least, the *TAF* component needs to securely obtain, store and use an anonymized credential which preserves the component's privacy when transmitting its information outside of the vehicle.

Axiom: Trust Assessment Framework security requirement
Axiom 16 $taf, DO, md, aiv, t, h, k_{AC}$: $\mathbf{SecurelyProduceVC}_{TAF}(taf, DO) \Leftrightarrow \mathbf{CorrectState}(taf) \wedge \mathbf{SecurelyProduceAttestationReport}_{AIV}(aiv, DO) \wedge \mathbf{CorrectState}(md) \wedge \mathbf{SecureDataExchange}(taf, aiv) \wedge \mathbf{SecureDataExchange}(taf, md) \wedge \mathbf{KeySecurelyLoaded}(k_{AC}, t, h) \wedge \mathbf{AnonymizedCredential}(k_{AC})$
The TAF component taf securely produces its Trust Opinion Verifiable Credential for a data object DO if and only if taf is at a correct state, aiv and md are at a correct state, they are both able to transmit their input to taf and taf has securely acquired, stored and used an anonymised credential for k_{AC} to construct its verifiable credential.

Table 6.20: Secure creation of Misbehavior Report Verifiable Credential.

6.8 Modelling In-Vehicle Devices

Concerning the in-vehicle *CONNECT* ecosystem, one key aspect that we need to capture in these security requirements is the *ECU* device life-cycle. In general, an important parameter that needs to be taken into account is the fact that *ECU* devices vary with respect to resources and can have different cryptographic capabilities. Therefore, the security requirements need to cover all types of devices that are provisioned to be securely on-boarded. It is worth mentioning that *N-ECU* devices that are not equipped with any RoT and any built-in cryptographic capabilities are considered out of scope of this security modelling since, by default, they can provide trustworthiness evidence with the appropriate signatures but not in a verifiable manner. Thus the *CONNECT* TAF will either not accept such trust sources or will process them with very low confidence resulting in a Trust Opinion with a high degree of uncertainty.

Concerning the *ECU* device life-cycle, it is essential that the axioms define the prerequisites for achieving the secure on-boarding of the devices. To that extent, we use a trusted Attestation Server *AS* - running either in the backend or within the Vehicle Manager - which verifies that the devices have been securely launched and that the applications running on top of the devices are certified by the *OEM*. In addition, we explore the requirements pertaining to the application-related functionalities of such devices. The aforementioned functionalities focus on the secure device-to-device and device to in-vehicle computer communication, as well as the secure transmission of provenance information from the devices.

6.8.1 Secure On-boarding of a new device

Before starting the secure on-boarding of a device there needs to be an *IAM* component that is already securely enrolled within the in-vehicle computer. In addition, the device needs to have a valid *TCB* and prove to the Attestation Server *AS* that it has been correctly configured and boot up. For TEE-enabled devices this can be achieved through the correct instantiation of the *Enclave* while for devices that have a less capable HSM this can be achieved by the secure transmission of a quote (e.g., TPM Quote) associated with a set of specific values stored in the HSM. The successful validation by the *AS* leads to the issuance of an *AS*-issued *VC* which certifies that the device is launched with the correct application. This credential can be then used by the device to communicate with the *IAM* application. Upon successful validation of that *VC* by the *IAM*, the secure onboarding concludes with the establishment of secure communication between *IAM* and the device (i.e., through the establishment of a pre-shared key). Thus, the necessary application-related and *CONNECT*related keys are securely exchanged and stored in the device's *RoT*. The axiom 6.21 on the secure on-boarding process is presented below:

Axiom: Secure On-boarding of a new device

Axiom 17 $d, iam, a, t, k_{APP}, k_{\text{CONNECT}}$: $\textbf{SecureOnboarding}(d) \Leftrightarrow \textbf{SecureEnrollment}(iam) \wedge \textbf{ApplicationInHost}(a, d) \wedge \textbf{Certified}_{AS}(d) \wedge \textbf{SecureDataExchange}(d, iam) \wedge \textbf{KeySecurelyLoaded}_{IAM}(\{k_{APP}, k_{\text{CONNECT}}\}, t, d)$

Device d can be securely on-board in the in-vehicle computer if and only if there is an IAM iam that is securely enrolled, the device has been certified by the attestation server AS that the d is launched correctly with a certified application a and the device has securely obtained and stored the necessary application-related k_{APP} and *CONNECT*-related keys k_{CONNECT}.

Table 6.21: Axiom for Secure On-boarding of a new device.

6.8.2 Secure In-vehicle communication

Upon the secure on-boarding, a device has securely obtained and stored all the necessary keys to perform its functionalities. Axioms 6.22 identify possible data flows starting from a device towards another device or a software component running within the in-vehicle computer. Firstly, when it comes to the transmission of kinematic data from an edge device (e.g., *Zonal controller (ZC)*) towards an in-vehicle software component (e.g., Facility Layer, *AIV*) the data confidentiality and integrity needs to be ensured. Given that the device is securely on-board and the recipient software component is securely enrolled, the secure communication of kinematic data can be established through the secure use of the respective cryptographic material issued by the *IAM* component. Similarly, in the case of exchanging kinematic data between two securely-onboarded devices (e.g., Lidar S-ECU and a *Zonal controller (ZC)*), the secure communication is achieved through the secure use of the necessary cryptographic material. Finally, the third axiom refers to the secure annotation of provenance data produced by a device. A device is able to securely prove the authenticity of the provenance data if and only if the device is securely on-board and it can securely use the respective cryptographic keys to compute the verifiable evidence (e.g., digital signature) over the provenance data.

Axiom: Application-related functionalities of a device
Axiom 18 d, s : $\mathbf{DeviceSendDataToApplication}(d, s) \Leftrightarrow \mathbf{SecureOnboarding}(d) \wedge (\mathbf{SecureEnrollment}(s) \vee \mathbf{SecureEnrollment}_{TCB-SW}(s)) \wedge \mathbf{SecureDataExchange}(d, s))$
Device d can securely transmit kinematic data to a software component s running in the in-vehicle computer if and only if d is securely on-board, s is securely enrolled and data integrity and confidentiality is achieved during their communication.
Axiom 19 d_1, d_2 : $\mathbf{DeviceSendDataToDevice}(d_1, d_2) \Leftrightarrow \mathbf{SecureOnboarding}(d_1) \wedge \mathbf{SecureOnboarding}(d_2) \wedge \mathbf{SecureDataExchange}(d_1, d_2))$
Device (e.g., S-ECU) d_1 can securely transmit kinematic data to another device (e.g., Zonal Controller) d_2 if and only if both devices are securely on-board and data integrity and confidentiality is achieved during their communication.
Axiom 20 d, k_{APP}, x_{PROV} : $\mathbf{AuthenticProvenanceData}(d) \Leftrightarrow \mathbf{SecureOnboarding}(d) \wedge \mathbf{VerifiableEvidence}(d, x_{PROV}, k_{APP})$
Device d can securely produce authentic provenance statement x_{PROV} related to kinematic data if and only if d is securely on-board and it securely computes a verifiable evidence over x_{PROV}, by securely using the respective keys k_{APP}.

Table 6.22: Axiom for device application-related functionalities.

Chapter 7

Conclusion and Outlook

7.1 Contributions of *CONNECT* Deliverable D4.1

In D4.1, we documented a refined architecture that documents key components and their interactions. This documents a bottom-up perspective: What components and services do we intend to provide to ensure security of automotive workloads in general and the *CONNECT* usage scenarios in particular.

In addition, we detailed user requirements. The user requirements that were described as ”user stories” specify specific usages by users, groups, and roles that will be provided by the *CONNECT* security architecture. This documents a top-down perspective where users express requirements that can then be used to validate the architecture.

We documented initial high-level designs for our Gramine and Intel SGX Trusted Execution Environments as well as required extensions. This will provide the security foundation of our vehicular software stack.

Finally, we documented formalized security requirements on the architecture that will provide a precise and unambiguous approach for validating the resulting concepts.

7.2 Open Questions and Next Step for Workpackage 4

After outlining the security architecture for *CONNECT*, this high-level architecture will be refined in Deliverable D4.2. We aim at resolving the following open questions in D4.2:

Refining Architecture and User Stories Since we continue to explore the technology and align with our user’s requirements, we expect to continue refining architecture and user stories.

Validating the TEE and Architecture: We plan to refine and test-drive the user stories to validate the architecture and TEE. The goal is to provide details for each user story to ensure that the architecture and TEE provides the required services.

Refined TEE Architecture We plan to provide more details on our TEE designs as well as the new features migration and upgrade.

Mobile Edge Cloud (MEC) Our current focus was on the security architecture of the vehicle. In D4.2, we plan to expand our scope to also cover the Mobile Edge Cloud (MEC).

High-Level Designs We plan to provide more details on the envisioned design of the architecture that has been documented in this deliverable.

Overall, we are within the envisioned time line for *CONNECT* and established a strong team that is collaborating towards the *CONNECT* objectives.

Appendix A

Appendix

A.1 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.

AMD SEV AMD Secure Encrypted Virtualization (AMD SEV) is a confidential computing technology offered by ARM CPU that protects virtual machines [1] .

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

AS Attestation Server.

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 [11]. 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 [9].

DAA Direct Anonymous Attestation.

DICE Device Identifier Composition Engine (DICE) is type of Trusted Execution Environment capable of providing runtime integrity garantees for each one of the processes comprising a software stack. This is achieved through the construction of conformity certificates. DICE also provide advanced secure bootup capabilities for the host device [1].

[1] https://trustedcomputinggroup.org/work-groups/dice-architectures/

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.

FL The Facility Layer (FL) manages the (kinematic) data stemming from the in-vehicle sensors by relaying them to all components of the Vehicle Manager that have subscribed to receive them. These are essentially the: (i) CAM/CPM Encoder/Decoder component that will start the construction of the respective V2X messages (e.g., CAM, CPM) to be broadcasted either following the currently ETSI specified standards or including also the Verifiable Presentations (VPs) comprising the trust-related information outputed by the TCH (i.e., T-CAM/T-CPM messages), (ii) CCAM Application module for constructing the local view of the vehicle's vicinity towards supporting the decisions making process of the service (e.g., breaking, changing lanes, etc.), (iii) the Misbehavior Detection service for checking the vercity of the measures kinematic data through a series of plausibility checks, and (iv) Trust Assessment Framework (TAF) for associating a Trust Opinion to each data object. It is important to highlight that the FL doesn't perform any processing or checks to the received data. Any verification controls, especially for asserting to the integrity of the data and its safety in the context of been signed and processed by only "*certified*" applications is performed by the IAM.

Group Signature A group signature allows a group member to digitally sign a message while staying anonymous within that group [10].

HSM Hardware Security Module.

IAM Identity and Authentication Management.

IMA Intersection Movement Assistance.

Intel TDX Intel Trusted Domain Extensions (Intel TDX) is a confidential computing technology offered by Intel CPU that protects virtual machines [27] .

IoT Internet of Things.

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.

N-ECU An *N-ECU* is a *ECU* with no cryptographic capabilities so, no trusted computing base (TCB), no secure storage for keys or other data.

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.

OS Operating System.

PCA The Pseudonym Certification Authority (PCA) is a trusted entity providing short-term anonymous credentials to on-boarded Vehicles. This is part of the ETSI standardized public Key Infrastructure (PKI) [13] for enabling the secure and privacy-preserving V2X communication.

PKI Public Key Infrastructure.

PUF Physically Unclonable Function.

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.

SDK Software Development Kit.

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.

TAR The Trust Assessment Request (TAR) is the triggering point, initiated by a CCAM application, for requesting from the CONNECT Trust Assessment Framework a trust opinion on the data exchanged within this specific service.

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 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.

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.

TPM Trusted Platform Module (TPM, also known as ISO/IEC 11889) is an international standard for a secure crypto processor, a dedicated micro-controller designed to secure hardware through integrated cryptographic keys. The term can also refer to a chip conforming to the TPM Standard of the Trusted Computing Group[26].

VC Vehicle Communication (V2X) This provides communication facilities for the vehicle. Connectivity – automotive ethernet, 5G, V2X.

VMM Virtual Machine Monitor.

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 [12].

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] AMD Secure Encrypted Virtualization (SEV) — AMD. https://www.amd.com/en/developer/sev.html.

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP*, 2013.

[3] Anna Angelogianni, Ioannis Krontiris, and Thanassis Giannetsos. Comparative evaluation of pki and daa-based architectures for v2x communication security. In *2023 IEEE Vehicular Networking Conference (VNC)*, pages 199–206, 2023.

[4] ARM®. TEE Reference Documentation. https://www.arm.com/technologies/trustzone-for-cortex-a/tee-reference-documentation.

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: secure Linux containers with Intel® SGX. In *OSDI*, 2016.

[6] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, 2014.

[7] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, page 132–145, New York, NY, USA, 2004. Association for Computing Machinery.

[8] Jan Camenisch, Manu Drijvers, and Anja Lehmann. Universally composable direct anonymous attestation. Cryptology ePrint Archive, Paper 2015/1246, 2015. https://eprint.iacr.org/2015/1246.

[9] Cooperative, connected and automated mobility (CCAM). https://transport.ec.europa.eu/transport-themes/intelligent-transport-systems/cooperative-connected-and-automated-mobility-ccam_en.

[10] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, volume 547 of *Lecture Notes in Computer Science*, pages 257–265. Springer, 1991.

[11] 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.

[12] 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.

[13] The CONNECT Consortium. Operational landscape, requirements and reference architecture - initial version. Deliverable D2.1, Project 101069688 within HORIZON-CL5-2021-D6-01, Nov. 2023.

[14] 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.

[15] 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.

[16] Intel Corporation. Intel® Software Guard Extensions Programming Reference. https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf, 2014.

[17] Intel Corporation. Intel SGX for Linux. https://github.com/intel/linux-sgx, 2023. [Online; accessed Aug-2023].

[18] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.

[19] 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.

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

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

[23] ETSI. ETSI TS 102 941 V2.2.1: Intelligent Transport Systems (ITS); Security; Trust and Privacy Management; Release 2, 2022.

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

[25] IETF Remote Attestation Working Group. Tee device interface security protocol (tdisp). RFC 18268, August 2022.

[26] Trusted Computing Group. Trusted Platform Module. https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/.

[27] Intel® Trust Domain Extensions (Intel® TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html.

[28] Benjamin Larsen, Thanassis Giannetsos, Ioannis Krontiris, and Kenneth Goldman. Direct anonymous attestation on the road: Efficient and privacy-preserving revocation in c-its. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, page 48–59, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking "security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2937–2950, New York, NY, USA, November 2021. Association for Computing Machinery.

[30] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2057–2073. USENIX Association, August 2020.

[31] Arttu Paju, Muhammad Owais Javed, Juha Nurmi, Juha Savimäki, Brian McGillion, and Billy Bob Brumley. SoK: A Systematic Review of TEE Usage for Developing Trusted Applications. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, pages 1–15, New York, NY, USA, August 2023. Association for Computing Machinery.

[32] Dimitrios Papamartzivanos, Sofia Anna Menesidou, Panagiotis Gouvas, and Thanassis Giannetsos. Towards efficient control-flow attestation with software-assisted multi-level execution tracing. In *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, pages 512–518, 2021.

[33] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51(6):130:1–130:36, January 2019.

[34] Dominik Arne Rebro, Stanislav Chren, and Bruno Rossi. Source code metrics for software defects prediction. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, page 1469–1472, New York, NY, USA, June 2023. Association for Computing Machinery.

[35] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *ASPLOS*, 2020.

[36] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB Linux Applications with SGX Enclaves. In *NDSS*, 2017.

[37] 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.

[38] Chia-Che Tsai, Mona Vij, and Donald Porter. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC*, 2017.

[39] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX'17, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Wikipedia. Trusted Computing Base. `https://en.wikipedia.org/w/index.php?title=Trusted_computing_base`. Accessed 10-Dec-2023.

[41] Wikipedia. Zero Trust Security. `https://en.wikipedia.org/w/index.php?title=Zero_trust_security_model`. Accessed 10-Dec-2023.