D4.3
REWIRE Runtime Assurance framework - Final Version

Project number:	101070627
Project acronym:	**REWIRE**
Project title:	REWiring the ComposItional Security VeRification and AssurancE of Systems of Systems Lifecycle
Project Start Date:	1st October, 2022
Duration:	36 months
Programme:	HORIZON-CL3-2021-CS-01
Deliverable Type:	Report
Reference Number:	HORIZON-CL3-2021-CS-01-101070627/ D4.3 / v1.0
Workpackage:	WP4
Actual Submission Date:	30th September, 2025
Responsible Organisation:	UBITECH
Editor:	Thanassis Giannetsos
Dissemination Level:	Public
Revision:	v1.0
Abstract:	Deliverable D4.3 documents the final version of the customizable REWIRE Trusted Execution Environment featuring a novel set of trust extensions for supporting secure SW Update; live migration; as well as the process state verification of a trusted app. We detail all provided trusted services towards securing the entire lifecycle of connected devices - including zero-touch onboarding, patching, security audits and establishment of secure and authenticated communication channels. As part of the final version of the REWIRE Trusted Computing Base, the new set of runtime Attestation Enablers, SW Update Processes and Secure Migration capabilities are also presented accompanied with detailed cryptographic protocols that safeguard a device's trust state. All these new sets of mechanisms have also been instantiated for RISC-V hardware architectures.
Keywords:	Attestation, Migration, Key Management, Process State verification

The The project REWIRE has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No 101070627.

Editor

Thanassis Giannetsos(UBITECH)

Contributors (ordered according to beneficiary numbers)

Spiros Kousouris, Sotiris Kousouris (S5)
Giannis Siachos, Stefanos Vasileiadis, Vasilis Kalos, Nikos Varvitsiotis, Giorgos Pekridis, Thanassis Giannetsos (UBITECH)
Sjors Van den Elzen (SECURA)
Samira Briongos, Javier de Vicente Gutierez (NEC)
Annika Wilde, Ghassan Karame (RUB)
Christiana Kyperounta (8BELLS)
Kaitai Liang, Zeshun Shi (TUD)
Yalan Wang, Liqun Chen (SURREY)

Disclaimer

Executive Summary

This deliverable, D4.3, presents the final technical specification and performance evaluation of the **REWIRE Runtime Assurance Framework**. It represents the culmination of the work package's efforts, building upon the initial framework presented in D4.2 and aligning with the final conceptual architecture from D2.2. This document provides the definitive cryptographic protocols, detailed action workflows, in-depth security analyses, and crucial performance benchmarks for the core runtime security enablers. It solidifies the transition from high-level design to a fully specified and evaluated set of tools for providing continuous, verifiable trust in devices throughout their operational life. All functionalities are built upon the **REWIRE Trusted Computing Base (TCB)**, which leverages the Keystone TEE framework on RISC-V to provide a secure foundation for all runtime operations.

A key contribution of this deliverable is the formalization of the complete trust lifecycle, organized into distinct phases. The first phase, **preparing a device to establish trust**, is accomplished through the **Zero-Touch Onboarding (ZTO)** protocol. We present the final design of two distinct ZTO flows, one based on selective disclosure via Verifiable Presentations and a variant using Zero-Knowledge Proofs for full attribute privacy, which enable a "Below Zero Trust" security posture. The former is powered by a novel **Attribute-Based Signcryption (ABSC)** scheme, whose design and performance are also detailed.

The second phase, **enabling a device to establish trust**, focuses on runtime attestation. The central mechanism is the **REWIRE Implicit Configuration Integrity Verification (CIV)** scheme, a novel approach that uses policy-restricted keys to provide privacy-preserving proof of a device's state. This is supported by the final design of the **REWIRE Tracer**, a non-intrusive, daemon-based component enhanced with EDR-like memory introspection capabilities. Furthermore, we outline the design for **Process State Runtime Verification**, a forward-looking capability for generating verifiable, DICE-like certificates for the entire enclave lifecycle.

Finally, the framework provides robust mechanisms for **re-establishing trust**. We present the finalized, secure protocols for both **Software Update** and **Live Migration**, including their security analyses. These processes ensure the atomicity and confidentiality of state transfer between enclaves, facilitated by specialized TCB-mediated mechanisms such as **Mirrored Keys** and an enclave-based **LRBC Key Manager**. The deliverable also details the **Harmonised Key Management** system, which provides a unified interface for all cryptographic operations within the Security Monitor. A significant contribution is the extensive performance evaluation of these runtime components. We provide one of the first detailed, comparative benchmarking studies of a CIV scheme across multiple TEE architectures—**Keystone (RISC-V)**, **Intel SGX**, and **OP-TEE (ARM TrustZone)**, offering critical insights into the performance trade-offs of different Roots of Trust. This empirical data, combined with the final protocol specifications, provides the definitive technical foundation for the runtime security and assurance capabilities of the REWIRE project.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Translation
ABE	Attribute-Based Encryption
ABS	Attribute-Based Access
ACC	Assisted Cruise Control
AE	Authenticated Encryption
AIC	Attestation Integrity Verification
AIK	Attestation Identity Key
API	Application Programming Interface
ATL	Actual Trust Level
BBL	Berkeley Boot Loader
BIOS	Basic Input/Output System
CDI	Compound Device Identifier
CIV	Configuration Integrity Verification
CPU	Central Processing Unit
CRL	Certificate Revocation List
CRTM	Core Root of Trust for Measurement
DDA	Direct Anonymous Attestation
DICE	Device Identifier Composition Engine
DID	Decentralised Identity Documents
DMA	Direct Memory Access
EAP	Enhanced Authentication Protocol
ECU	Electronic Control Units
EK	Endorsement Key
EPC	Enclave Page Cache
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
HSM	Hardware Security Module
HW	Hardware
IDM	Identity Management
IOMMU	Input/Output Memory Management Unit
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
KMS	Key management system
MUD	Manufacturer Usage Description
OEM	Original Equipment Manufacturer
PCR	Platform Configuration Register
PMP	Physical Memory Protection
PPI	Physical Presence Interface
PSK	Pre-Shared Key

PUF	Physically Unclonable Function
RA	Rich Application
RATS	Remote ATtestation procedureS
REE	Rich Execution Environment
RoT	Root of Trust
RT	Runtime
SBI	Supervisor Binary Interface
SC	Side-Channel
SDLC	Software Development Lifecycle
SGX	Software Guard Extensions
SM	Security Monitor
SoC	System-on-Chip
SP	Service Provider
SRK	Storage Root Key
SUIT	Software Updates for Internet of Things
SSA	Security-Sensitive Application
SSI	Self-Sovereign Identity
SW	Software
TA	Trusted Application
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TDISP	TEE Device Interface Protocol
TEE	Trusted Execution Environment
TMFS	TEE Management Framework Specification
TOCTOU	Time of Check Time of Use
TPM	Trusted Platform Module
TTP	Trusted Third Parties
UDS	Unique Device Secret
VC	Verifiable Credentials
VM	Virtual Machine
VP	Verifiable Presentations
vTPM	Virtual Trusted Platform Module
ZTO	Zero-touch onboarding

Versioning and contribution history

Version	Date	Summary of changes	List of Contributors
v0.1	13.01.2025	Table of Contents & Allocation of tasks to the partners	Spiros Kousouris (S5), Thanassis Giannetsos (UBITECH)
v0.2	27.03.2025	Description of the final version of the REWIRE CIV (Chapter 4) and SW Update process (Chapter 3)	Samira Briongos, Javier de Vicente Gutierrez (NEC), Annika Wilde, Ghassan Karame (RUB), Stefanos Vasileiadis, Vasilis Kalos, Thanassis Giannetsos (UBITECH), Kaitai Liang, Zeshun Shi (TUD)
v0.3	12.04.2025	Definition of the requirements for the construction of the Verifiable Presentations holding device trust attributes. This set the scene for the later finalization and evaluation of the REWIRE ZTO scheme (Chapter 6) and Attribute-based SignCryption (Chapter 7)	Yalan Wang, Liqun Chen (SURREY), Stefanos Vasileiadis, Vasilis Kalos, Thanassis Giannetsos (UBITECH)
v0.4	24.04.2025	Description of the novel Key Management System (KMS) (Chapter 5) based on a set of harmonized interfaces so as to allow th management of the underlying crypto primitives directly through the Security Monitor (SM)	Samira Briongos, Javier de Vicente Gutierrez (NEC), Annika Wilde, Ghassan Karame (RUB), Stefanos Vasileiadis, Vasilis Kalos, Thanassis Giannetsos (UBITECH), Spiros kousouris (S5), Christiana Kyperounta (8BELLS)
v0.5	01.05.2025	Description of the final version of the REWIRE attestation enablers and live migration process. This was also associated with the final design of the REWIRE TEE capabilities for the secure enclave management (Chapters 4 & 3, respectively)	Samira Briongos, Javier de Vicente Gutierrez (NEC), Annika Wilde, Ghassan Karame (RUB), Stefanos Vasileiadis, Vasilis Kalos, Thanassis Giannetsos (UBITECH), Kaitai Liang, Zeshun Shi (TUD)
v0.6	18.05.2025	Finalization of the formal security analysis of the REWIRE SW Update process and migration scheme (Chapter 3)	Yalan Wang, Liqun Chen (SURREY), Stefanos Vasileiadis, Vasilis Kalos, Thanassis Giannetsos (UBITECH)
v0.7	12.06.2025	Quantitative analysis of the REWIRE TCB featuring all internal attestation enablers and crypto primitives - especially, as it pertains to the security properties offered by the newly designed attestation enablers and the process state verification (Chapter 4)	Annika Wilde, Ghassan Karame (RUB), Stefanos Vasileiadis, Vasilis Kalos, Nikos Varvitsiotis (UBITECH)
v0.8	09.07.2025	Documentation of the REWIRE monitor hooks capable of tracing the runtime system measurements of the target device (Chapter 8)	Sjors Van den Elzen (SECURA)
v0.85	15.07.2025	Final version of the overarching REWIRE crypto agility layer (Chapter 7)	Yalan Wang, Liqun Chen (SURREY), Stefanos Vasileiadis, Vasilis Kalos, Dimitris Papamartzivanos, Thanassis Giannetsos (UBITECH)
v0.92	21.07.2025	Internal review for quality assurance	Liqun Chen, Nada El Kseem (SURREY), Claudio Soriente (NEC)

v1.0	29.09.2025	All results documented throughout the deliverable were ratified also against the end-to-end experiments performed in the context of the envisioned use cases (D6.2). This was the reason behind the consortium opting to submit the final version of all deliverables on M36 where all results were available	Giannis Siachos, Thanasis Giannetsos (UBITECH)
v1.0	30/09/2025	Submission of deliverable	Thanassis Charemis (UBITECH)

Chapter 1

Introduction

1.1 Scope and Purpose

The primary purpose of this deliverable is to present the final and complete technical specification of the **REWIRE Runtime Assurance Framework**. This document moves beyond the initial concepts outlined in D4.2 and the high-level architecture of D2.2 to provide the definitive protocols, cryptographic designs, security analyses, and component-level performance evaluations for the core technologies that enable continuous security verification for devices in the field. It serves as the main technical guide for the runtime security components developed within WP4.

The scope of this document is comprehensive, covering the final design and analysis of all major runtime functionalities of the REWIRE TCB. Specifically, this deliverable details:

- **Secure Lifecycle Management Protocols:** The finalized protocols for **Secure Software Update** and **Secure Software Migration**, including their detailed action workflows and full security analyses against the defined threat model. This represents the final version of the mechanisms sketched in D4.2.
- **REWIRE Trust Extensions:** The full design space of the trust extensions that allow a device to establish and re-establish its trust level. This includes the final specification and extensive performance evaluation of the **REWIRE Implicit Configuration Integrity Verification (CIV)** scheme across multiple TEEs. It also details the mechanisms for authenticated usage of hardware-based keys, such as **Mirrored Keys** and the **Enclave-based LRBC Key Manager**, and provides the conceptual design for **Process State Runtime Verification**.
- **Harmonised Key Management:** The final design and interface specification for the **REWIRE Harmonised Key Management (KMS)** system. This includes its integration as a Security Monitor extension, the key lifecycle it manages, and the specific Supervisor Binary Interface (SBI) calls exposed to enclaves.
- **Zero-Touch Onboarding (ZTO):** The complete cryptographic specification and evaluation of the ZTO process. This includes the formalization of two distinct flows—one based on **Selective Disclosure** and an advanced variant using **Proof-of-Ownership**—and the detailed design, security analysis, and benchmarking of the underlying **Attribute-Based Signcryption (ABSC)** scheme.
- **REWIRE Tracer:** The final design, implementation details, and operational workflow of the **REWIRE Tracer**. This includes its enhanced capabilities for non-intrusive memory introspection and its role in providing the verifiable evidence required by the CIV scheme.

Ultimately, this deliverable provides both the complete technical blueprint for the REWIRE runtime security mechanisms and the empirical performance data that validates their practical feasibility on representative hardware platforms.

1.2 Relation to other WPs and Deliverables

This deliverable, D4.3, provides the final version of the **REWIRE Runtime Assurance Framework**. At its core resides the **REWIRE Trusted Computing Base (TCB)**, which exposes the trusted services necessary for securing the entire lifecycle of connected devices. This deliverable solidifies the initial designs from D4.2 and provides the complete technical specifications and performance evaluations for all runtime security enablers. These include mechanisms for establishing, maintaining, and re-establishing trust, such as **Zero-Touch Onboarding (ZTO)**, runtime integrity monitoring through the final **Configuration Integrity Verification (CIV)** scheme, and lifecycle management actions like **Secure Software Update** and **Secure Live Migration**.

As the final output of the core technical work in **WP4 (Runtime Trust and Security Assurance)**, this deliverable serves as a critical link between the project's design phases and its final integration and evaluation. Its relationship with other project activities is as follows:

- **Input from WP2 and WP3:** D4.3 is the direct implementation of the runtime architecture and requirements defined in **D2.2 (REWIRE Operational Landscape, Requirements and Reference Architecture)**. It takes as input the security policies generated by the **Compositional Verification and Validation** pipeline from **WP3 (Design-Time Trust and Security Assurance)** and provides the TCB-based enablers to enforce them. The formal verification activities of WP3 have also informed the security design of the protocols detailed herein.
- **Evolution from D4.2:** This document is the final and complete version of the framework introduced in **D4.2 (REWIRE Runtime assurance framework - Initial Version)**. It replaces the initial protocol sketches and high-level designs with finalized specifications, comprehensive security analyses, and crucial performance benchmarks for all runtime components.
- **Foundation for WP5:** The runtime cryptographic primitives and attestation mechanisms finalized in this deliverable, such as the Attribute-Based Signcryption (ABSC) scheme and the Verifiable Credentials produced by the CIV process, provide the technical foundation for the trust-aware authentication and access control mechanisms of the **Blockchain Infrastructure (WP5)**.
- **Input for WP6:** D4.3 provides the definitive set of technical artefacts for the runtime assurance framework that will be integrated and evaluated in the project's final use cases as part of **WP6 (Framework Integration, Use Cases and Impact Maximisation)**. The component-level benchmarks presented here will inform the end-to-end evaluation that will be documented in **D6.2**.

In summary, D4.3 consolidates all the runtime security innovations of the REWIRE project, providing the stable and validated technical foundation required for the final integration and impact assessment activities.

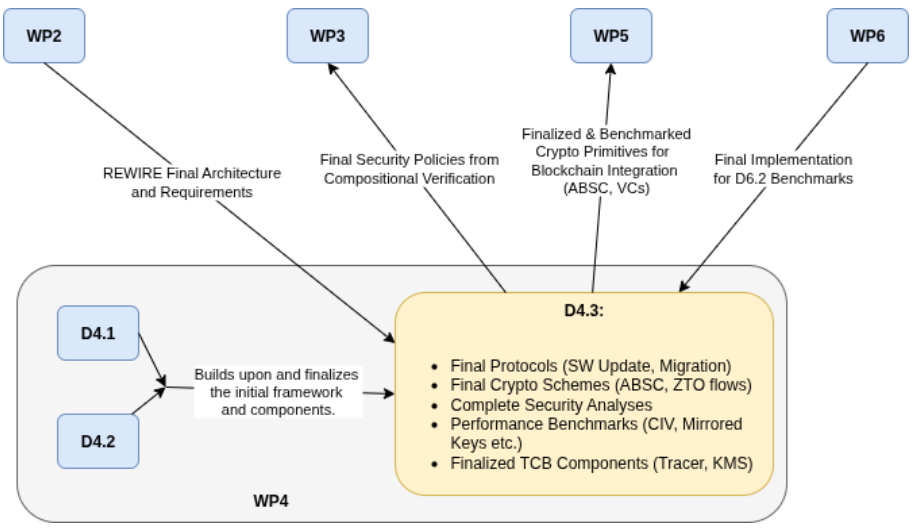

Figure 1.1: Relation to other WPs and Deliverables

1.3 Deliverable Structure

This deliverable is organised into the following chapters, providing a logical progression from the foundational concepts to the detailed technical specifications and evaluations of the REWIRE Runtime Assurance Framework:

- **Chapter 1** provides an introduction to the deliverable, outlining its scope, purpose, and relationship with other project activities.
- **Chapter 2** presents a conceptual overview of the **REWIRE Trusted Computing Base (TCB)**, establishing the architectural foundation upon which all the runtime assurance mechanisms are built.
- **Chapter 3** details the finalised functionalities for secure lifecycle management, covering the protocols and comprehensive security analyses for both **Secure Software Update** and **Secure Software Migration**.
- **Chapter 4** describes the final design and evaluation of the core **REWIRE Trust Extensions**. This includes the mechanisms for preparing and establishing a device's trust level, with a focus on the **Implicit Configuration Integrity Verification (CIV)** scheme and the mechanisms for authenticated usage of hardware-based keys, such as **Mirrored Keys**.
- **Chapter 5** presents the design of the **REWIRE Harmonised Key Management** system. It details its integration as a Security Monitor extension, the key hierarchy, lifecycle management, and the specific SBI calls exposed to enclaves.
- **Chapter 6** provides a comprehensive description and security analysis of the **REWIRE Zero-Touch Onboarding (ZTO)** process. It details the two distinct onboarding flows and the underlying **Attribute-Based Signcryption (ABSC)** scheme.
- **Chapter 7** details the REWIRE instantiation of the ABSC scheme, including its use of attributes, access trees, and performance benchmarks.
- **Chapter 8** presents the final design, implementation path, and operational workflow of the **REWIRE Tracer**, the component responsible for non-intrusive evidence collection.
- **Chapter 9** concludes the deliverable, summarising the key contributions and the final status of the REWIRE Runtime Assurance Framework.

Chapter 2

Summary of the TCB

2.1 Conceptual Overview of REWIRE's TCB

The REWIRE framework is designed to provide verifiable security and trust guarantees for the entire lifecycle of devices operating in complex, heterogeneous Systems-of-Systems (SoS). Central to this goal is the **REWIRE Trusted Computing Base (TCB)**, which comprises newly designed software components incorporating trust extensions and cryptographic mechanisms, alongside existing hardware capabilities provided by a secure element. Together, these components enable REWIRE to ensure a high level of assurance for the device. The TCB works in tandem with the underlying hardware Root of Trust and leverages its isolation capabilities to protect all sensitive computations and data, even in the event that the host is compromised. This chapter provides a conceptual overview of the final architecture of the REWIRE TCB and its runtime functionalities, as they pertain to secure lifecycle management, i.e., software update, live migration, and configuration integrity verification based on verifiable evidence, monitoring, and tracing. It significantly expands upon the high-level descriptions presented in D2.2 [29]. The TCB exposes a rich set of security services that are orchestrated to enhance a device's capabilities, not only enabling it to establish its trust level, but also to continuously monitor it and react to any changes, ensuring ongoing compliance with the required level of assurance. The REWIRE TCB comprises trust extensions and attestation capabilities that are agnostic to the underlying Root of Trust, as long as the secure element provides a specific set of capabilities (detailed in Deliverable D2.1 [22]) centred around secure measurement, storage, and reporting. Nevertheless, particular focus in the instantiation of the TCB is placed on the **Keystone Trusted Execution Environment (TEE)** [30] framework, which leverages standard features of the RISC-V Instruction Set Architecture (ISA) [7], such as Physical Memory Protection (PMP), to create secure enclaves. The core of the TCB is the **Security Monitor (SM)**, a highly privileged software layer running in machine mode (M-mode) that is responsible for enclave lifecycle management, memory isolation, and mediating all interactions between the untrusted world and the trusted enclaves through a secure Supervisor Binary Interface (SBI) [4]. All core trust extensions and cryptographic primitives that can be integrated into higher-layer security services are provided by REWIRE's TCB, either through harmonized interfaces exposed by the security monitor or as separate trusted applications deployed within isolated enclaves. The final architecture of this TCB is depicted in Figure 2.1, and it's core components and capabilities are detailed in Table 2.1 (alongside with their new features since D4.2).

The security policies produced during the Design Phase (see Deliverable D2.2 [29]), and especially through the **Compositional Verification and Validation** pipeline, address the arduous task of narrowing down the set of properties that must be attested at runtime to achieve the required trust level. REWIRE is **one of the first** frameworks to converge the benefits of **Formal Verification and Runtime Attestation** in order to reduce the design space of properties that need to be verified. The **Policy Orchestrator** serves as the critical bridge for enacting upon the security policies to be enforced. As already mentioned, originating from the **Compositional Verification and Validation** component, security and operational

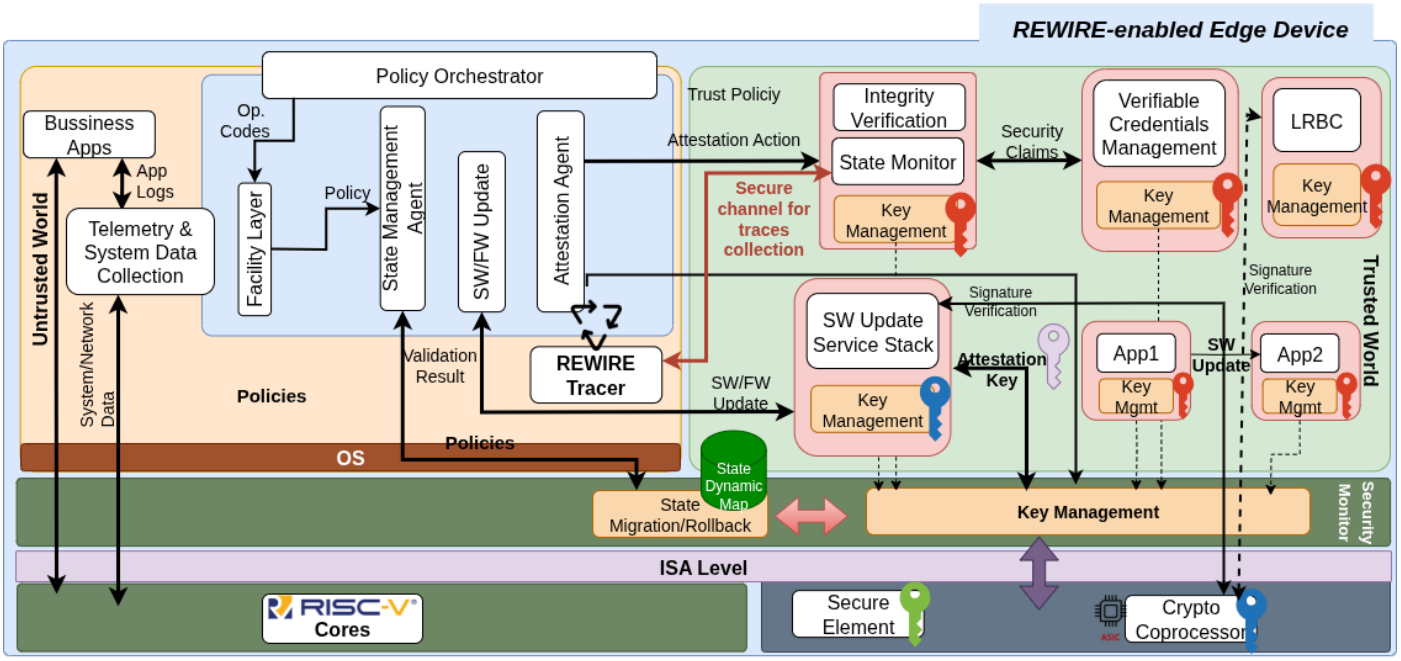

Figure 2.1: Final Architecture of the REWIRE Trusted Computing Base

policies, expressed in the Medium Security Policy Language (MSPL), are transmitted to the Policy Orchestrator. The Orchestrator interprets these high-level policies and translates them into concrete execution logic for the device's **Facility Layer**. This layer, in turn, invokes the appropriate TCB capabilities, whether it is triggering an attestation task, initiating a software update, or commencing a live migration.

Before a device can perform any runtime trust operations, it must first be securely provisioned with the cryptographic material required to secure it's identity management. This crucial preparatory phase is accomplished through the **Zero-Touch Onboarding (ZTO)** protocol, a new scheme that leverages the TCB for allowing the secure enrolment of a device if and only if it can exhibit the required set of attributes. REWIRE opted to provide an extension of the well-established Extensible Authentication Protocol (EAP) [33] tailored for embedded systems and topologies that require device enrolment based not only on authentication, but also on the verification of required attributes. The ZTO process enables a device to securely enrol in a new domain without manual intervention. Leveraging its pre-established Root ID Key, the device authenticates itself to a trusted and a **Privacy CA** with the support of a Manufacturer. Upon successful authentication, the Privacy CA, after accessing the device's extended Manufacturer Usage Description profile [15], issues a **Verifiable Credential (VC)**. This VC contains the device's core attributes and is cryptographically bound to its public **Attestation Key**. All keys and credentials generated during this process (including the private part of the Attestation Key) are securely managed by a set of enclaved components, such as the *Attestation Agent* and the *Verifiable Credential Manager*, which are equipped to produce, at runtime, zero-knowledge reports or verifiable presentations. These two functionalities constitute another key innovation of REWIRE in the context of implicit attestation, where the trust assumptions placed on the verifier are weakened by enabling evidence of a device's correct state without disclosing raw traces, and by supporting Verifiable Presentations with Selective Disclosure. As discussed in Chapter 6, REWIRE provides two cryptographic schemes that enable either selective disclosure or the generation of a zero-knowledge proof for all attributes comprising a Verifiable Credential, both leveraging advanced cryptographic structures such as BBS signatures [31]. Beyond that, and in an effort to address the existing fragmentation of secure elements used as Roots of Trust across heterogeneous devices, REWIRE also introduces a **Harmonised Key Management System (KMS)**. This system enables centralized key management through the Security Monitor. Specifically, it extends the Security Monitor with a cryptographic key management subsystem, allowing all keys to be migrated from enclaved trusted applications to be managed directly by it. These keys can then be accessed in an authenticated and predicate-controlled manner.

With the device now securely equipped with all the necessary primitives, the TCB can unlock the runtime monitoring and the establishment of trust. The primary mechanism for this is the **Configuration Integrity Verification (CIV)** scheme, a novel approach to runtime attestation that is both efficient and privacy-preserving. The core innovation of REWIRE's CIV is that it enables **implicit attestation** based on **key restriction usage policies**. Instead of exporting measurements for remote verification, the device proves its integrity through its ability to use it's restricted key to perform a cryptographic operation. This process is managed by two key components: the **REWIRE Tracer**, a daemon-based component that non-intrusively collects runtime evidence like configuration hashes, and the **Attestation Agent**, a secure enclave that compares this evidence against the policies enforced by the Security Monitor. The agent can then successfully sign a challenge from a Verifier, and this signature serves as conclusive, zero-knowledge proof that the device is in a correct state. This mechanism, detailed further in D4.2 [26], alleviates the need for a powerful remote Verifier for every check and allows for frequent, low-overhead integrity assessments. In addition to this, REWIRE extends its implicit attestation scheme with capabilities for ensuring the correctness of the enforced Key Restriction Usage Policy. This enhancement elevates the CIV protocol to *CIV with Verifiable Policy Enforcement*, by enriching the TCB with an additional component of the same name. This component acts as a *silent observer*, continuously verifying that only authenticated and fresh Key Restriction Usage Policies are enforced, i.e., ensuring that no obsolete policy bound to a previous software version is applied.

Building upon this capability, REWIRE also proceeds to introduce a more granular **Process State Verification** enabler. This mechanism moves beyond static configuration checks to provide a verifiable audit trail upon request for a specific process. As will be documented in Section 4.2.2, these required novel memory introspection capabilities allowing the decomposition and reading of the ELF representation of the trusted app binary in the enclaved page cash. It provides the TCB with the ability to translate the output of attestation processes into secure, auditable certificates that can be inspected by any authorized remote Verifier. This is a significant innovation, as it allows a device to self-issue **DICE-like certificates** that provide runtime guarantees about an enclave's behaviour from its initial launch through its entire execution. To achieve this, the TCB leverages new Supervisor Binary Interface (SBI) calls that allow the Security Monitor to securely introspect the state of enclaves, including their memory allocation and execution status, by querying the hardware's PMP registers. This provides a fine-grained, verifiable record of an enclave's runtime integrity, a feature essential for high-assurance systems and compliance scenarios.

The final, and perhaps most critical, role of the TCB is to manage the secure evolution of the device, thereby re-establishing trust whenever its configuration or trust level changes. This aligns with the concepts of *Resilient Computing* The REWIRE framework provides two primary mechanisms for this secure lifecycle management. The first is a robust **Secure Software Update** capability, designed to apply patches or feature updates with verifiable evidence on their correct enforcement. The process, orchestrated by the TCB, begins with a CIV attestation to ensure the device is in a valid state to receive an update. The TCB then securely initializes the new enclave and allows it to run if and only if it can verify the correct provisioning and loading of the initial application state. This process leverages the underlying sealing keys provided and mediated by the TCB in order to extract the state of the source application and load it to the target one thus ensuring application and service continuity with high integrity. In addition to this, REWIRE has also provided an AES variant called LRBC-2 which is a symmetric key cryptographic mechanism with strong guarantees against key leakage. Accessing this LRBC key for managing the authenticated encryption of the update is protected using dedicated, single-use transport keys known as **Mirrored Keys**. These keys are managed by the Security Monitor to ensure that the state remains confidential and that the old enclave cannot access the new state, guaranteeing backward secrecy. Crucially, the TCB enforces atomicity by ensuring the old enclave is terminated before the new one begins execution, preventing two versions from running in parallel. A final CIV attestation is performed post-update to verify the correctness of the new installation. The full protocol, which has been refined for Keystone to handle race conditions, is detailed in Chapter 3.

In modern cybersecurity engineering for complex Systems-of-Systems (SoS), components and services

do not exist in isolation but are embedded in a complex ecosystem with many interdependencies. Security maintenance must be a continuous process throughout the operational phase of these systems. However, traditional maintenance processes, which often rely on manual triage and deployment of updates, can take considerable time, leaving systems vulnerable in the interim. The REWIRE ecosystem requires more dynamic and automated response strategies to handle security-critical attack scenarios as they unfold in the field. One such strategy is to migrate services or their dependencies to a more secure environment when a threat is detected. The trigger for such a migration is directly linked to the interdependencies between components. If an onboard analysis mechanism, reveals that a component's security requirements can no longer be guaranteed, the system faces a critical decision. Rather than simply shutting down a service, which could impact availability, a more resilient approach is to migrate the function or its dependency to another, more trustworthy component with similar features. This need is amplified by the broader industry trend towards Software-Defined Systems, where a decoupling of hardware and software is essential. As applications evolve into platform-independent components, they must possess their own identities, which are decoupled from the hardware they run on. Consequently, when an application is relocated, its identity (represented by its cryptographic keys) must migrate alongside it. Since these keys are critical security assets, securely migrating them from one trusted platform to another is a non-trivial challenge that requires robust, verifiable protocols. To address this challenge, REWIRE offers a **Secure Live Migration** capability. This process is a significant innovation that focuses on verifiable state management rather than just a simple state transfer. It is structured into three phases, all managed by the TCBs of the participating devices. Firstly, the devices are provisioned with the necessary credentials and migration policies. Secondly, the source TCB can extract the volatile and persisted application state which is then securely sealed and encrypted. As part of this process, REWIRE can also leverage the memory introspection capabilities of the Process State Verification to able to extract the volatile state of the application. Finally, the target TCB receives this state, decrypts it, and reinstates it within a new enclave. It then performs a local attestation to generate verifiable proof that the application has been correctly restored and is executing as expected. This mechanism ensures that critical services can be moved to healthy platforms without compromising their integrity or confidentiality. The detailed protocol for this process is presented in Chapter 3.

In summary, the REWIRE TCB is a comprehensive security solution that provides a holistic toolkit for runtime assurance. It facilitates the initial provisioning of a device, enables continuous and privacy-preserving attestation of its runtime state, and orchestrates secure lifecycle management actions to ensure the device can evolve safely over time. The preceding overview has established *what* the TCB provides; the subsequent chapters of this deliverable will detail precisely *how* these services are implemented through finalized protocols, cryptographic schemes, and security analyses.

Table 2.1: Full Design Space of REWIRE Trust Extensions

Component / Capability	Description
Hardware Root-of-Trust (RoT)	The immutable hardware foundation of the TCB, typically a RISC-V CPU featuring the Physical Memory Protection (PMP) extension. The RoT provides the ultimate source of trust for the system, anchors the device's unique identity via a hardware-protected Root ID Key, and provides the essential primitives for enforcing memory isolation. As stated, we have to highlight that all trust extensions are agnostic to the underlying Root of Trust.
Harmonised Key Management System (KMS)	An integrated subsystem running as part of the Security Monitor, providing a unified interface for all cryptographic key operations. The KMS is responsible for the secure generation, storage, usage, and destruction of all keys required by REWIRE services, including Attestation Keys, Tracer Keys, and Mirrored Keys for state migration. This is a new feature that is envisioned to be part of the final REWIRE TCB.
REWIRE Tracer	A daemon-based component operating within the trusted world, designed for non-intrusive evidence collection. It performs real-time introspection of target binaries to gather configuration data and execution traces, which it then securely provides to the Attestation Agent for verification.
Implicit Configuration Integrity Verification (CIV)	A primary attestation service enabled by the TCB. CIV leverages the Attestation Agent and Tracer to provide implicit, local proof of a device's configuration integrity. By making key usage conditional on policy compliance, it allows for verification via a simple challenge-response protocol, eliminating the need to expose sensitive measurements. In this document, an extended benchmarking has been performed, allowing a concrete evaluation of the CIV implementation in Keystone against other well-established RoTs.
Process State Verification	An advanced runtime verification service that provides an auditable, verifiable record of an enclave's lifecycle. It introduces new SM extensions that can allow advanced introspection capabilities and issue DICE-like certificates, offering strong, auditable guarantees about the correct runtime state of enclave processes to any authorized remote party.
Secure Software Update	A critical lifecycle management service orchestrated by the TCB. This service ensures the authenticated and confidential update of enclaved software, guaranteeing the atomicity of the operation and the secure transfer of application state between versions using TCB-managed transport keys. Here, the final version of the SW Update, that was implemented and benchmarked in D6.1 [27], is formally verified and proved to be secured/
Secure Live Migration	The provision of a resilient service that enables the verifiable migration of a full application state from a potentially compromised device to a new, trusted host. The TCB orchestrates the secure export, encrypted transfer, and verifiable import of the enclave state, ensuring service continuity and integrity.
Mirrored Keys	A specialized cryptographic mechanism managed by the KMS within the TCB to facilitate secure state hand-off during lifecycle events like updates or migration. These are single-use, symmetric transport keys derived by the SM to allow a source enclave to encrypt its state such that only a specific, authorized target enclave can decrypt it, ensuring confidentiality and forward/backward secrecy during the transition.

Chapter 3

Functionalities regarding SW Update and Migration

In Deliverable D4.1 [23], we elaborated on the importance of **SW Update** and **Migration** functionalities for supporting a device's secure lifecycle, especially as it pertains to efficiently reacting to possible indications of risks (e.g., patch management, secure migration of tasks to other devices (or the Edge - task offloading)) for maintaining the required trust level of the device. These capabilities are not merely maintenance functions; they are core enablers for the broader vision of **resilient computing** [11] within the next-generation "Systems-of-Systems" (SoS) that REWIRE targets. In these modern ecosystems, which are distributed over the continuum from cyber-physical end devices to edge servers and cloud facilities, the ability to dynamically manage workloads is paramount for achieving fault tolerance and load balancing. Resilient computing is facilitated by the ability to replicate or move services in response to changing conditions; for instance, if an infrastructure element's workload significantly increases or if it fails, its function can be automatically migrated to another element. However, realizing this vision in ecosystems secured by Trusted Execution Environments (TEEs) introduces significant security challenges that traditional migration schemes do not address. The core problem is how to ensure verifiable application state migration for TEE-based applications. Hardware-enforced security guarantees that isolate enclave data, meaning that an enclave's state is often sealed with keys that never leave the processor, making a simple state transfer impossible. Thus, new introspection capabilities are required that can balance state monitoring with the strong isolation guarantees of a TEE. This is precisely the challenge that the REWIRE runtime extensions address, providing mechanisms for the secure and verifiable management of enclave applications and their state.

The **SW Update** protocol, on one hand, allows the secure update of the application binary which has been launched in an enclave while preserving the enclave state, e.g., in the event of a detected security vulnerability that is fixed by a released patch. On the other hand, to **increase service availability and to cope with situations where a device needs to undergo maintenance or reaches the end of its life, it is important that services can be migrated.** As aforementioned, this can happen between elements across the entire far-edge/edge/cloud continuum - between devices that have already established a trust relationship or from a device to the backend Edge or Cloud infrastructure, where more resource-rich functional assets can be deployed to support the operation of safety-critical applications. In D4.2 [26], we specified the details of the secure SW Update process to allow the exchange of application updates through *authenticated* and *confidential* communication channels: As documented in D3.2 [25], this is achieved through the use of an **enhanced AES crypto mode of operation** (**REWIRE LRBC**) that also provides strong security guarantees against physical attacks; i.e., resilience against SW Update Key leakage attacks especially in the "*one-to-one*" scenario where a dedicated application update is pushed by the SW Distribution Service to a specific (target) device. Additionally, we provided an initial sketch of REWIRE's RISC-V trust extensions for enabling the secure migration. In what follows, we provide

a detailed description of the SW Migration process. Furthermore, we analyze the security of both protocols based on a set of security properties. For both schemes, the associated interfaces have been implemented to enable interactions with other REWIRE components. Concrete documentation of the implementation details of the update protocol was provided in D6.1 [27]. In contrast, the implementation details of the migration protocol will be documented in D6.2, along with a detailed benchmarking and evaluation in the context of the envisioned use cases.

3.1 Threat Model

Our system model comprises two RISC-V devices, D_S and D_T, running instance SM_S and SM_T, respectively, of the same Security Monitor (SM) software. Throughout the secure SW Update and Migration processes, we migrate an enclave state S from the source enclave E_S to the target enclave E_T. In the secure SW Update process, E_S and E_T have distinct software versions v and v', respectively, and are both hosted by SM_S. In contrast, E_S and E_T have the same version v during the secure SW Migration process, but are operated by SM_S and SM_T, respectively. A Software Distribution Service (SDS) initiates and authorizes the SW update and migration. Furthermore, V_X denotes the view of party X on the state of a SW update/migration process.

We adopt a conventional threat model for TEEs, in which the hardware platform is part of the TCB. All privileged software components, including the OS, hypervisor, and system management software, are considered untrusted and potentially adversarial. Within this model, the adversary is assumed to have full control over system resources external to the enclave, including main memory, persistent storage, and network interfaces. Moreover, the adversary controls the host application and can instantiate arbitrary enclaves, including enclaves with malicious behavior.

In the context of enclave update and migration, we consider three primary threat scenarios:

1. **Update to an Unauthorized SW version.** The adversary attempts to update an enclave to a self-crafted, malicious enclave version. Such an enclave designed by the adversary may leak the enclave's state, compromising it's confidentiality.
2. **Unauthorized Migration to a Compromised Platform.** The adversary attempts to migrate an enclave to a device under their control, thereby compromising its security guarantees. For instance, if the adversary gains access to the SM's secret key, they may forge attestation reports or derive the sealing key, enabling unauthorized access to sealed data. This compromises both the confidentiality and integrity of the enclave's state and execution.
3. **State Forking and Consistency Violations.** The adversary aims to induce a forking attack by creating multiple concurrent instances of the same enclave, violating the uniqueness and consistency of its internal state. This can be achieved by launching the same enclave twice on a single device, or migrating the enclave to a new device while allowing the original instance to continue executing on the source device. As a result, the adversary obtains two logically identical enclave instances, denoted E_1 and E_2, both initialized in the same state S at the moment of forking. The adversary can then supply divergent inputs I_1 and I_2 to E_1 and E_2, respectively—driving them into inconsistent successor states S' and S''. Although the integrity of each enclave instance remains intact, the consistency of their shared logical state is violated, potentially leading to semantic discrepancies or application-level vulnerabilities.

A high-level sketch of how an adversary might orchestrate such attacks by exploiting the lifecycle management protocols is as follows, using a structure inspired by well-known TEE attack methodologies:

1. **Start-stop-restart (State Capture):** The adversary begins by establishing a baseline. A legitimate enclave, E_S, is started on a source device, D_S. The adversary allows it to perform an initial operation and then signals for termination, causing the enclave to persist its initial state, S_1 (corresponding to software version v_1), to untrusted storage using the TCB's sealing mechanism. The

adversary captures this sealed state blob. They may then restart the enclave on the source device to confirm the state is valid before proceeding.

2. **Initiate Lifecycle Event & Intercept:** The adversary waits for or triggers a legitimate lifecycle management operation, such as a **Software Update** to version v_2 or a **Live Migration** to a destination device, D_T. During this critical transition phase, the adversary leverages their control over the untrusted network and storage to execute their primary attack:
   - To achieve a **State Forking** attack, the adversary allows the legitimate update or migration protocol to complete successfully, resulting in a new, valid enclave instance, E_T, on the target device. The attack proceeds to the next step.
   - To perform an **Unauthorized Update** or **Migration to a Compromised Platform**, the adversary intercepts the protocol flow. They might inject a self-crafted, malicious software package ($v_{malicious}$) in place of the legitimate one, or they could redirect the entire state transfer to a device under their complete control, D_{adv}.
3. **Terminate-restart (Fork Execution):** This step actualizes the state forking attack. After the legitimate process has created the new enclave instance E_T on the destination device, the adversary uses the captured state blob (S_1) from Step 1 to restart the original enclave, E_S, back on the source device, D_S. If the protocol lacks a robust mechanism to permanently invalidate the source enclave after a lifecycle event, this action succeeds, creating two concurrent instances of the enclave with inconsistent state.
4. **Achieve Malicious Objective:** If the protocol's security guarantees are insufficient, the adversary's attack succeeds. In the case of a fork, they can now supply divergent inputs to the two enclave instances to violate application-specific logic. In the case of an unauthorized update or migration, the enclave's confidential state is exposed on the compromised platform, or the device is now running malicious code under the guise of a legitimate update, violating both confidentiality and integrity.

The security protocols for SW Update and Migration detailed in this chapter are explicitly designed to thwart this attack sketch (see Sections 3.3.2 and 3.2).

3.2 Security Objectives

The SW Update and Migration protocols were designed to comply with the following security objectives, ensuring they can withstand the adversary described above. We analyze how the SW Update and Migration protocols achieve these properties in Sections 3.3.2 and 3.2, respectively.

- **Atomicity.** Unsuccessful updates or migrations cannot be installed. More specifically, either (i) the update (or migration) is performed successfully, and only the destination enclave (on the destination host) is running, or (ii) the update (or migration) fails, and only the source enclave (on the source host) is running.
- **Authentication.** An untrusted entity should not be able to issue an enclave update (or migration) on a certain device. Furthermore, the source should only update (or migrate) to a trusted destination (host), while the destination (host) should only accept enclaves from trusted sources.
- **Code integrity.** Any modification of the enclave code during the SW Migration process should be detectable.
- **State confidentiality and integrity.** An untrusted entity must not learn any meaningful information about the enclave state during the Update or Migration process. Furthermore, any modification of the enclave state during the Update or Migration process should be detectable.
- **Version enforcement.** Once version v of enclave software is installed, any older version $v' < v$ cannot be installed.

- **Cloning protection.** At each point in time, only one instance of each enclave (regardless of its version) must be running on the source or destination host.

3.3 Secure & Authenticated SW Update

REWIRE's secure and authenticated SW Update process enables the target RISC-V HW architecture to update *enclavized* software securely. On a high level, the update payload is assembled by the SW Distribution Service, which then also triggers the update in the enclave. The actual update is then performed by the Security Monitor on the respective device. It will only be installed if it is coded by a known and established developer and triggered by the SW Distribution Service. We briefly summarize the protocol flow in Section 3.3.1 and analyze its security in Section 3.3.2. We refer the interested reader to Deliverable D4.2 [26] for a detailed protocol treatment.

Note that refinements were made systematically throughout the implementation process. Specifically, we made three core changes to adapt to the provisions of Keystone [30]. First, Keystone does not support hyperthreading. Hence, two enclaves cannot be orchestrated by the same host, such that we had to work with a source and target host. Furthermore, the SM is limited in the means to identify where the enclave stores state data, such that we opted to let the source enclave seal its state instead of directly extracting it from within the SM. Similarly, an enclave can only be destroyed by the host. We isolate the source enclave from the SM by not forwarding any further ecalls after a successful update to ensure its decomissioning. The implementation details were documented in D6.1 [27].

3.3.1 High-Level Flow for Enclave Update

Below, we give an overview of the SW Update process as detailed in Deliverable D4.2 [26]. For the enclave update process, we assume a scenario where a developer provides an update version v' for an enclave software with the identifier ID. The developer then signs a hash of the ID, the version number, and the software. The signature binds the ID and version number to the software and attests that the software was issued by an authorized developer. The ID, version number, software, and signature are forwarded to the software distribution service.

The SW Distribution Service, dubbed SDS, then prepares the data blob to install the software on a device. First, a nonce N_{SDS} is chosen at random to link logs to the specific update request, i.e., be able to track which updates failed exactly. This nonce is added to the update data received from the developer. In the case of a one-to-one update, i.e., the update is issued for exactly one device, the update is passed through an authenticated and encrypted channel using the SW Update Key, leveraging the REWIRE LR-BC operation mode. Decrypting and verifying the encrypted data later allows us to prove the authenticity of the update request. If the update targets multiple devices, encryption is impractical. Instead, the SDS signs the update and nonce with its private key. The verification of this signature proves that the SDS intended the update. Further, the SDS sets a timeout for the messages subsequently exchanged with the enclave and SM. If the timeout expires before the SM confirms the update, it is considered failed.

The signed or encrypted update is then sent to the enclave that is to be updated. The enclave forwards the update with its local enclave ID, eid, to the SM, which will perform the update. If the update is a *one-to-one* update, the SM forwards the encrypted data to the SC-resistant AE enclave, requesting decryption and validation. The SC-resistant AE enclave interacts with the underlying ASIC or other secure element to securely leverage the SW Update Key for this process. Secure access to the SW Update Key is granted through the Key Management extensions of the SM. The SW Update Key never leaves the packaging of the ASIC to ensure the resilience property. The SC-resistant AE enclave returns a valid bit, and the decrypted nonce and developer's update payload. If the update is in the *one-to-many* mode, the enclave verifies the SDS's signature with the public keys that are known to the enclave. Only if the valid bit is set

does the SM accept the update and proceed with installation. The same holds for all the following steps: if any checks fail, the update is aborted.

The following steps are equal for *one-to-one* and *one-to-many* updates. After successfully verifying the SDS's intent to install the update, the SM verifies the authenticity and integrity of the provided software. That includes verifying the software hash, the developer's signature, and the version number. The version number v' of the provided software must be strictly higher than that of the currently installed software, v, to guarantee protection against SW rollback attacks (see Property "Version enforcement" in Section 3.2).

The SM now chooses an id eid' for the updated enclave. It appends (eid, eid') to a data structure, keeping track of all enclaves that are currently being updated. If the SM crashes during the update, it will detect, at restart, that the update (eid, eid') is not yet completed. If the enclave with the ID eid still exists, it will resume only this enclave and not the enclave with ID eid'. Otherwise, it will resume eid', assuming the update was installed successfully. This ensures (1) that the updated enclave is only launched if the update has been completed successfully and (2) that no two versions of the same enclave are launched in parallel. The SM then initializes the new enclave and stops the old enclave that is to be updated. If the enclave is marked as stateful, i.e., the state needs to be transferred to the updated enclave, the SM makes a snapshot of the enclave's state. It then copies the state to the memory of the updated enclave. Afterward, the new enclave is started.

The last step of the update process is creating an authenticated log message. Therefore, the SM first attests the updated enclave using the REWIRE enhanced CIV attestation scheme. The log message then includes the attestation report, the nonce, the software ID, and the version number to bind the message to a specific update request. The log message is forwarded to the SC-resistant AE enclave, which encrypts the message with the SW Update Key, proving the authenticity of the log message. The encrypted message is returned to the SM and forwarded to the SDS. The SM additionally sets a timeout. If it does not receive an OK message from the SDS before the timeout expires, the SM resumes the old enclave and destroys the updated enclave. The SDS verifies the message and appends it to a log file. It sends an OK message to the SM if the verification is successful. The SM destroys the old enclave and removes (eid, eid') from the struct of updates in progress to ensure that the enclave will be restarted after an SM crash. Finally, the SM sends another OK message to the SDS confirming the completion of the update.

3.3.2 Security Analysis

In this section, we analyze the security of the proposed SW Update process based on the security objectives defined in Section 3.2.

Property 1—Atomicity: The SM and SDS have the views V_{SM} and V_{SDS}, respectively, regarding the outcome of the update process. An update process has only two possible outcomes: either the update fails and the old enclave version is resumed (denoted by $V = 0$), or it succeeds and the updated enclave version is running (denoted by $V = 1$). This property guarantees a consistent view of the update process between all involved parties, primarily the SM (which performs the update) and the SDS (which initiates the update). After the process, both components must agree on one of two outcomes: either the update was completed successfully, or the update failed due to a crash, and the original enclave remains operational. Hence, the **adversary's goal** is to create inconsistent views of the update process among the involved parties, i.e., $V_{SM} \neq V_{SDS}$. Once an update is triggered by the SDS sending the update trigger to the SM, the adversary has the following **capabilities** during the update execution to achieve this goal:

- The adversary controls all network communication exchanged between the SM and the SDS, i.e., they can drop any message on the channel.
- The adversary controls the target device running the SM, i.e., they can delay any messages between the old/updated enclave and the SM.

- The adversary can cause a crash of the SDS, the SM, and/or the enclaves running on the target device.

The attack is considered successful if, at the end of the update process, $V_{SM} \neq V_{SDS}$, i.e., the SM has a different view of the update than the SDS. Synchronized timeouts and message acknowledgments ensure that the view of both SM and SDS remains consistent in the face of crashes of either party or loss of messages, caused by the adversary or system failures.

Note that all communication between the target enclave and the SDS is sent over a secure TLS channel. Hence, the adversary cannot modify or forge any messages transmitted between the SDS and the enclave to signal the SDS a successful update despite failure, and vice versa. Similarly, the communication between the enclave and the SM in Keystone is protected from the untrusted host—the adversary—by means of memory isolation, such that the adversary cannot tamper with these messages.

When the SDS sends an update request, it initiates a timeout that spans the full update duration. Only if the SDS receives an acknowledgement (OK) from the updated enclave before the timeout expires, it considers the update successful ($V_{SDS} = 1$). Similarly, the SM initiates a timeout when responding to the SM. Only if it receives an acknowledgement by the SDS does it destroy the old enclave and consider the update successful ($V_{SM} = 1$), acknowledging this to the SDS. If any message before the acknowledgement from the SDS to the SM is dropped, these timeouts expire, causing the SM to resume the old enclave, and both parties consider the update failed ($V_{SDS} = V_{SM} = 0$). Delaying messages between the enclave and the SM in the worst case also leads to timeouts expiring. Any party crashing or being crashed before this message is sent has the same outcome as if the respective following message were not sent. Since the updated enclave is only running on a lease before the acknowledgement from the SDS, this reflects the actual update state.

The first change in the SM's view of the update state occurs when the SM receives an acknowledgment of the update from the SDS. In this case, it destroys the old enclave and the timeout, and considers the update successful ($V_{SM} = 1$) such that the updated enclave returns another acknowledgement to the SDS. When receiving the acknowledgement from the SM, the SDS also advances to consider the update successful ($V_{SM} = V_{SDS} = 1$).

Note that the SDS cannot transition to $V_{SDS} = 1$ if $V_{SM} = 0$, since the SM would not send the acknowledgement in this case. However, if the target device crashes before sending the final acknowledgment to the SDS or this message is dropped, a conflict arises in the views of the update, specifically $(V_{SDS} = 0) \neq (V_{SM} = 1)$ (dubbed scenario A). We argue that there is no solution ensuring a consistent view without a crash fault-tolerant layer. Namely, if the SDS transitions to $V_{SDS} = 1$ before sending an acknowledgement to the SM, and this message is not received by the SM due to a crash or being dropped, the same issue remains, but in the opposite direction (scenario B). In scenario A, (i) the updated enclave is running (which is assumed to be more secure) and (ii) the SDS is expected to retry installing the failed update. We assume that the SDS checks the enclave version running on the target device before attempting to install an update. In this step, the SDS would detect that the update has been previously installed, allowing to resolve the discrepancy.

On the other hand, the SM has no means to detect that the SDS assumes it runs a different enclave version than it does in scenario B. Thus, the old enclave instance (which may contain a vulnerability) remains running until the next enclave version is installed, posing a security risk. Hence, we opt for scenario A, which we argue to be more secure.

Property 2—Authentication: An update is initiated in response to the detection of a security risk or when expanding the features of an enclavized software with the software identifier ID. The version of the current enclave software is denoted by v, and the corresponding enclave binary by $SW(v)$. The developer then develops a new version v' of the software and provides an updated binary $SW(v')$. ID, v', and $SW(v')$ are sent to the software distribution service (SDS), which forwards them to the

target enclave. This property guarantees that only authorized parties—i.e., the developer and the SDS—can provide and initiate software updates. Hence, the **adversary's goal** is to initiate an unauthorized update on the target device.

To enforce authorized updates, the REWIRE secure update protocol leverages ECDSA signatures from the developer and the SDS to authorize update payloads. The SM only installs updates that are signed by the SDS. Specifically, the developer signs the update payload with its private key sk_{Dev}, computing a signature σ_{Dev}. Additionally, the update payload is encrypted with AES and the symmetric session key (k_{DS}) of a TLS session. We denote the encrypted payload by Enc_{Dev}. The adversary is given **access** to the update process and may attempt to modify σ_{Dev} and Enc_{Dev}. The SDS then decrypts Enc_{Dev} and verifies the signature using the corresponding public key pk_{Dev}. Upon verification success, the SDS signs the payload with its own private key (sk_{SDS}), and forwards the signature (σ_{SDS}) to the target enclave, together with the payload, which is encrypted with the corresponding TLS session key.

The adversary performed a successful attack if the following conditions were met:

1. $Dec(Enc_{Dev}, k_{SD}) \neq N||ID||v'||SW(v')$
2. $ECDSA\_verify(\sigma_{Dev}, Dec(Enc_{Dev}, k_{SD})) = 1$

The same conditions apply to communication between the SDS and the target enclave. We assume that the ECDSA signature scheme and AES encryption are secure. Hence, the adversary can only modify Enc_{Dev} to decrypt to a meaningful plaintext knowing the correct encryption key. Similarly, the adversary can only forge a valid signature if they know Dev's private signing key. We further assume that the developers' (and SDS's) private keys are uncompromised. The use of a nonce further ensures that each update session is uniquely bound to a specific update instance, effectively mitigating replay attacks. Together, the use of encryption, ECDSA signatures, and nonces ensures that only updates authenticated by the developer and authorized by the device owner can be installed.

Property 3—State confidentiality & integrity: An enclave may optionally maintain an internal state S, which must be transferred to the updated enclave during the update process to ensure uninterrupted functionality across versions. Formally, the initial state S' of the updated enclave must equal the latest state S of the original enclave, i.e., $S' = S$. Additionally, this state is considered confidential and must remain protected against unauthorized disclosure or tampering.

This property ensures that an adversary cannot obtain information about the internal enclave state S, nor can it modify this state without detection during the update procedure. Thus, the **adversary's objective** is to violate either the integrity of the state, such that $S' \neq S$, or gain knowledge of S during its transition. We assume a powerful adversary model with **privileged access** to the device throughout the update process, including the ability to modify sealed state data at rest. The REWIRE secure update protocol leverages Keystone's sealing functionality and hardware-enforced runtime memory isolation via Physical Memory Protection (PMP) to mitigate such threats.

To protect both S and S' at rest, Keystone employs a sealing mechanism that encrypts the state using a symmetric key derivable only by the Security Monitor and the enclave itself. Provided that the underlying cryptographic primitives are secure, this mechanism guarantees confidentiality of the sealed state. Moreover, the sealed state includes a Message Authentication Code (MAC), enabling the SM to detect any unauthorized modifications upon decryption. Any tampering with the sealed data will be detected during MAC verification, thereby preserving state integrity.

During the update process, the sealed state S is temporarily decrypted into the SM's runtime memory before being re-encrypted and delivered to the updated enclave. During this period, PMP entries are configured to restrict access to this memory region exclusively to the SM, preventing all other software entities—including any adversarial code executing at the host level—from accessing or modifying the plaintext state.

Therefore, under the assumptions that (i) the sealing key remains inaccessible to untrusted components, (ii) MAC verification is correctly performed by the SM, and (iii) PMP is enforced correctly by the underlying hardware, the REWIRE update protocol ensures that the enclave's internal state remains both confidential and unmodified throughout the update lifecycle. Any attempt by an adversary to violate these guarantees will either be prevented (via access control) or detected (via MAC failure), ensuring that $S' = S$ holds with high assurance.

Property 4—Version enforcement: A common motivation for triggering an enclave update from software version v to a higher version v' is the remediation of security vulnerabilities present in the earlier version $SW(v)$. Allowing the original enclave software $SW(v)$ to continue execution after the update—particularly with access to the state now owned by the updated enclave $SW(v')$—may violate core security guarantees.

This property ensures the permanent decommissioning of the original enclave instance following a successful update, thereby preventing unauthorized access to state by obsolete enclave versions. Let E denote the original enclave instance executing software $SW(v)$, and let E' denote the updated enclave instance running $SW(v')$. The **adversary's goal** is to resume the execution of E after the update to E' has completed, thereby gaining access to the enclave state now associated with $SW(v')$. Hence, the adversary gets **privileged access** to the target device when the update starts. This privileged access includes the capability to initialize and run arbitrary enclaves.

To prevent restarts of outdated enclave binaries, the REWIRE secure update protocol ensures that the source enclave E is permanently destroyed by the SM immediately after the successful initialization of the target enclave E'. Once this destruction occurs, any attempt to resume or execute E is rejected by the SM, as the corresponding enclave no longer exists within the system's enclave registry. However, this guarantee applies only to the specific instance E involved in the update. A malicious host may attempt to circumvent this protection by launching a new instance of the old enclave binary corresponding to $SW(v)$. To address this, REWIRE assumes that the SM enforces strict version checks during enclave initialization and rejects any attempt to instantiate enclave software with a version identifier $v < v'$, i.e., any version known to be superseded. Throughout the update process, a similar check prevents the installation of outdated versions, also known as software rollback attacks.

Together, these mechanisms ensure that once an enclave has been updated to a newer software version, the original, potentially corrupted enclave software cannot be reused—either through resumption or reinstallation.

Property 5—Cloning protection: Depending on the use case of an enclave application, an adversary may benefit from running multiple concurrent instances of the same software. Specifically, the **adversary's goal** is to start three enclave instances E, E', and E'', where E runs $SW(v)$ and E' and E'' run $SW(v')$. In an instance of a cloning attack, the adversary can forward attestation requests to E' while routing service requests to E, allowing for the exploitation of vulnerabilities in $SW(v)$ despite the attestation proving $SW(v')$. Likewise, the adversary can split service requests to E' and E'', each unaware of the other's inputs, which results in conflicting enclave states. Similar to Property 4, the adversary is granted **privileged access** to the target device, allowing the installation of arbitrary enclave software.

This property guarantees that at most one enclave instance—whether running the original software version $SW(v)$ or the updated version $SW(v')$—is active under a specific SM at any point in time. During the update process, the SM enforces strict sequential execution semantics. Specifically, the source enclave is paused before the target enclave is launched. If the update completes successfully, the source enclave is permanently decommissioned (cf. Property 4). If the update fails, the SM securely destroys the target enclave instance and resumes the original enclave. At no time during this procedure are both the source and target enclave instances permitted to execute concurrently. Consequently, an adversary cannot launch E in parallel to E'.

Beyond the update and migration processes, REWIRE assumes that the SM enforces a system-wide singleton policy by rejecting any request to initialize a new enclave instance if another enclave with the same software identifier and version is already active. This invariant prevents multiple concurrent instantiations of the same enclave software version, thereby preventing parallel execution of E' and E''. Together, these mechanisms prevent concurrency-induced inconsistencies and ensure that enclave state, execution, and control flow remain tightly regulated throughout and beyond the update lifecycle.

3.4 Secure SW Migration

REWIRE's secure enclave migration protocol allows the REWIRE chip to migrate enclave software and state from one host to another. Specifically, an enclave state can be migrated from one enclave to another, and/or from one device to another. Migration from one enclave to another on the same device is equal to an update as described in Section 3.3. An enclave migration will only be executed if the source and destination hosts are mutually trusted and comply with the enclave's security policy. More specifically, the trust level of the target node needs to comply with the requirements of the trust level a device needs to have for hosting the specific safety critical application. A realistic use case can be found in the Automotive Domain. Here, safety-critical operations provided by an Advanced Driver-Assistance System (ADAS) may run as non-trusted applications outside of an enclave, yet they still require access to a consistent set of cryptographic keys for identity management and authenticated communication with other Electronic Control Units (ECUs). Should a sensor component be identified as compromised, an effective resilience strategy is the migration of its persistent state—specifically its cryptographic keys—to a neighboring ECU. This migration allows the neighboring unit to assume the compromised sensor's functions, ensuring service continuity, under the condition that it has access to the equivalent safety-critical application. This takeover remains in effect until the source ECU can be remediated and restored to its required trust level.

3.4.1 Protocol for Enclave Migration

The following protocol ensures secure enclave migration from one device to another while preserving the enclave state. It guarantees that only one of the source and target enclaves is running at the end of the migration protocol, both in the case of successful completion and migration abort. In this section, we start by describing our assumptions and the core building blocks of the enclave migration protocol.

3.4.1.1 Preliminaries and Building Blocks

This section outlines the fundamental components underlying the REWIRE secure enclave migration protocol. We assume that the TCB offers the required capabilities, and therefore abstract away these details in the protocol description in the following.

- **State abstraction.** We distinguish two types of enclave state: *volatile* and *persistent*. Persistent state refers to data that is explicitly preserved across sessions by the enclave, typically encrypted using its sealing key. This category includes any information intentionally sealed for long-term storage, and it can be readily extracted and verified using cryptographic primitives. In contrast, volatile state resides in the enclave's runtime memory and includes elements such as file descriptors, counters, data structures, and symbolic links—typically located on the stack or heap. Extracting this state requires specialized support. We assume the TCB provides two interfaces, `get_enclave_state` and `set_enclave_state`, which allow the state to be extracted and re-instantiated at the correct memory offsets, respectively. Note that the extraction of volatile and persistent memory can be realized in RISC-V by inspecting the memory layout. We refer to Section 4.2.2 for further details. Notably, this approach does not account for pointers. Correctly handling pointers would require

identifying them, determining their referents, and accurately remapping them during migration—a significantly more complex task.

- **Cloning protection.** A key security objective of REWIRE's enclave migration protocol is to guarantee that, at any point in time, only a single active instance of a given enclave exists—either on the source or the destination host (cf. Section 3.2). The migration protocol ensures that once migration succeeds, the source enclave is deleted; if migration fails, the destination enclave is deleted. In both cases, the protocol prevents simultaneous execution on both devices. However, the protocol does not prevent a malicious party from launching multiple instances of the same enclave on a single device. To mitigate this, we assume that the SM enforces runtime exclusivity by rejecting enclave initialization requests if an instance of the same binary is already active. This could be implemented, for example, via a registration check in the enclave's initialization routine.
- **Migration agent.** Communication during the migration process primarily occurs between the SMs of the source and destination hosts. This communication, along with any interaction with external entities, is facilitated by an untrusted user-space component referred to as the *Migration Agent*. The Migration Agent acts as an intermediary, relaying messages to and from the SMs. For clarity, this component is omitted from subsequent protocol descriptions.

3.4.1.2 Protocol Overview

For the enclave migration process, depicted in Figure 3.1, we assume a scenario in which the Software Distribution Service (SDS) receives a migration request for an enclave E_S currently operated by a Security Monitor SM_S, identified by its public key. In the first step, the SDS attests SM_S and the enclave by means of the public key and REWIRE's enhanced CIV attestation, respectively. The SDS then selects a target device for the migration, SM_T, and assembles the migration request, which encompasses the enclave hash and the public keys of the source and target SM together with a signature over this data. This signature confirms that the source and target devices belong to the same pool and migration between them is intended.

Next, the SDS sends the migration request to SM_S using a secure TLS connection. Furthermore, it sets a timeout for the subsequent messages. If the timeout expires before the SM confirms the migration, it is considered failed. We use timeouts for all messages to identify failed migrations.

Upon receiving the migration request, SM_S validates its authenticity by verifying the signature, including its public key and the enclave hash. The SM proceeds with the migration only if this verification succeeds. It then establishes a secure TLS connection to the target device, SM_T, and sends a `PREPARE` message including the request. SM_T equally verifies the request and signals its readiness to receive migration data with a `READY` message only if the verification succeeds.

SM_S pauses the execution of E_S, ensuring that it does not process any further inputs, potentially modifying the enclave state, during the migration process. Next, SM_S extracts the enclave's code and data from the device's runtime memory. Furthermore, it decrypts the enclave's sealed state with the corresponding sealing key. It then sends the code, data, and sealed state to SM_T, which initializes a new enclave with the received code and checks the enclave's measurement against the one from the migration request. Furthermore, SM_T re-encrypts the state with the new enclave's sealing key. It then sends a `DONE` message to SM_S, which attests the new enclave using REWIRE's enhanced CIV attestation scheme. Upon successful attestation, SM_S destroys its enclave and sends an `OK` to SM_T, signaling successful completion of the protocol, which in turn responds with an `OK`. If SM_T receives this message, it sends another `OK` to the SDS. Otherwise, it raises an alarm.

In summary, the migration protocol is comprised of the following steps:

1. The Software Distribution Service (SDS) receives a migration request for the enclave E_S operated by the Security Monitor SM_S, represented by its public key pk_{SM_S}, on the source device D_S. A migration request can result from various incidents, such as maintenance work, the identification

of a hardware bug on D_S, or an update of the platform platform. Each of the mentioned scenarios requires the migration of the enclave to a new device to ensure availability or restore security guarantees.

2. The SDS initiates a secure TLS connection with SM_S, yielding keys for the encryption of all data exchanged between the SDS and SM_S. All subsequent communication between the SDS and SM_S is transmitted and secured through this TLS connection.
3. The SDS retrieves the measurement H_{E_S} of E_S.
4. The SDS attests the source enclave that should be migrated, E_S, leveraging the REWIRE enhanced CIV attestation scheme.
5. The SDS selects a suitable target device SM_T, represented by its public key pk_{SM_T}.
6. The SDS assembles the migration request: $req = H_{E_S}||pk_{SM_S}||pk_{SM_T}$.
7. The SDS computes the ECDSA signature over the migration request: $\sigma_{SDS} = ECDSA\_sign_{sk_{SDS}}(H_{E_S}||pk_{SM_S}||pk_{SM_T})$.
8. The SDS sends the req and σ_{SDS} to SM_S and sets a timeout t_1. If SM_S does not reply with an `OK` before t_1 expires, the SDS considers the migration procedure failed.
9. SM_S validates the authenticity of the migration request by verifying σ_{SDS}: $ECDSA\_verify_{pk_{SDS}}(\sigma_{SDS}, H_{E_S}||pk_{SM_S}||pk_{SM_T}) \stackrel{?}{=} 1$.
10. SM_S initiates a secure TLS connection with SM_T, yielding keys for the encryption of all data exchanged between the SM_S and SM_T. All subsequent communication between the SM_S and SM_T is transmitted and secured through this TLS connection.
11. SM_S sends a `PREPARE` message, including req and σ_{SDS} to SM_T, indicating the migration request to SM_T. Furthermore, it sets a timeout t_2. If SM_T does not respond with a `READY` message before t_2 expires, the migration is considered failed by SM_S.
12. SM_T validates that its public key matches pk_{SM_T}, and that SM_S's public key matches pk_{SM_S}.
13. SM_T validates the authenticity of the migration request by verifying σ_{SDS}: $ECDSA\_verify_{pk_{SDS}}(\sigma_{SDS}, H_{E_S}||pk_{SM_S}||pk_{SM_T}) \stackrel{?}{=} 1$. Successful verification validates that (i) the SDS triggered the migration request and (ii) SM_S and SM_T are within the same device pool.
14. SM_T sends a `READY` message to SM_S, indicating that it is ready to receive the migration of the enclave with the hash H_{E_S}. Additionally, it sets a timeout t_3. If SM_S does not respond with a transmission of code and data before t_3 expires, the migration is considered failed by SM_T.
15. SM_S pauses the execution of E_S.
16. SM_S retrieves E_S's DRAM pages, i.e., its code C.
17. SM_S retrieves E_S's sealed state and decrypts it with the corresponding sealing key, yielding the plaintext state S.
18. SM_S sends the a `DATA` message, including C and S, to SM_T and sets a timeout t_4. If SM_T does not respond with a `DONE` message before t_4 expires, the migration is considered as failed by SM_S. SM_S resumes E_S in this scenario.
19. SM_T initializes a new enclave, E_T with the code C.
20. SM_T validates that E_T's measurement matches the expected value: $H(E_T) \stackrel{?}{=} H_{E_S}$.
21. SM_T re-encrypts S with E_T's sealing key.
22. SM_T starts the execution of E_T.

23. SM_T sends a `DONE` message to SM_S and sets a timeout t_5. If SM_S does not respond with an `OK` message before t_5 expires, the migration is considered failed by SM_T. SM_T suspends E_T and deletes its data in this scenario. Consequently, E_T runs on a lease until the migration is considered finalized.
24. SM_S attests E_T, confirming that it runs the same code as E_S on the correct host SM_T.
25. SM_S destroys E_S, deleting all associated data.
26. SM_S removes the lease and returns an `OK` message to SM_T.
27. SM_T responds to SM_S with an `OK` message.
28. SM_S sends an `OK` message to the SDS if it received the `OK` from SM_T and raises an alarm otherwise.

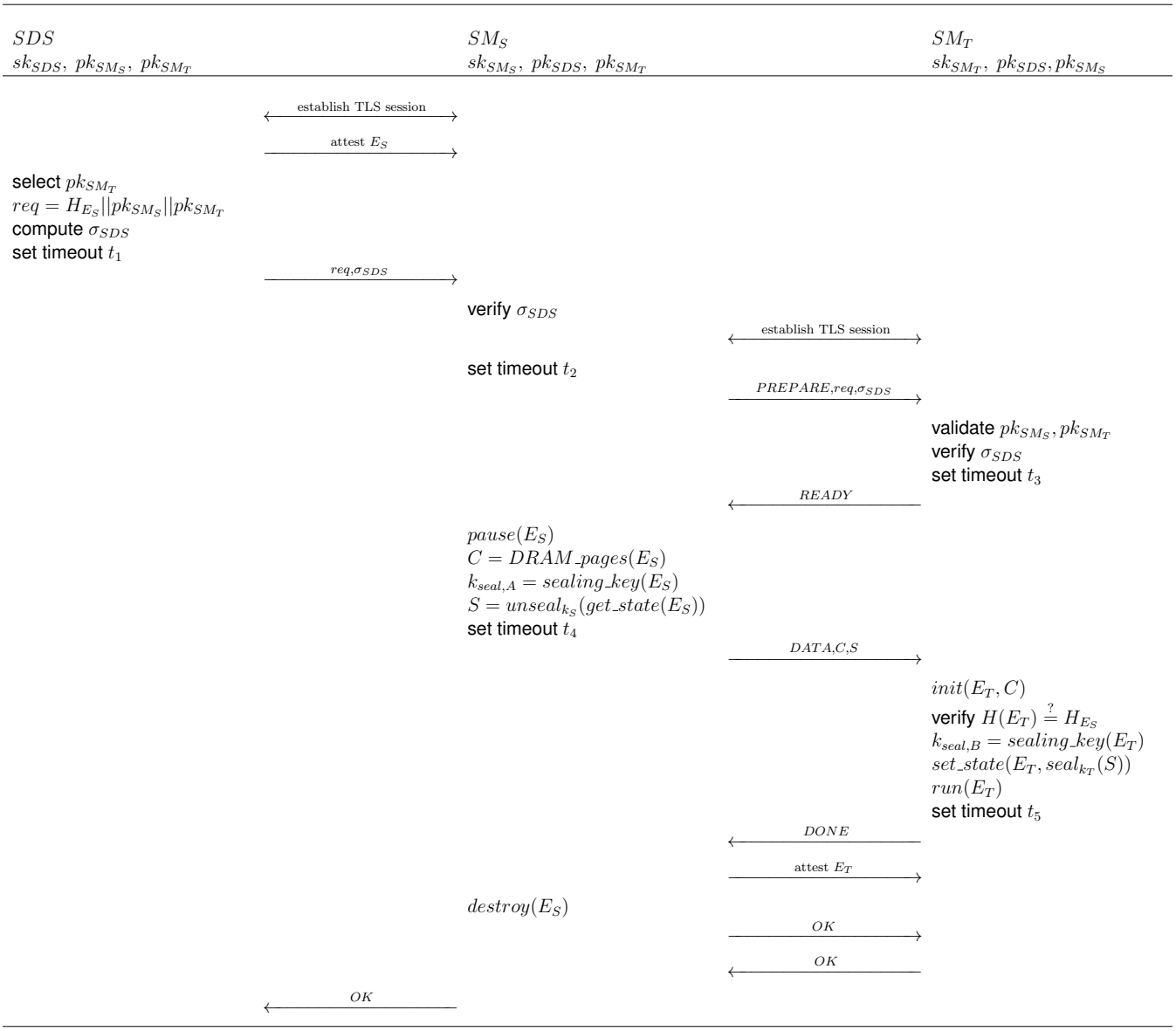

Figure 3.1: Migration protocol for enclaves.

3.4.2 Security Analysis

In this section, we analyze the security of the proposed SW Migration process based on the security objectives defined in Section 3.2.

Property 1—Atomicity: The source/destination SMs and the SDS have the views V_{SM_S}/V_{SM_T} and V_{SDS}, respectively, regarding the outcome of the migration process. A migration process has only two possible outcomes: either the migration fails and the source enclave is resumed on the source host (denoted by $V = 0$), or it succeeds and the destination enclave is running on the destination host (denoted by $V = 1$). This property guarantees a consistent view of the migration process between all involved parties, primarily the SMs (which perform the migration) and the SDS (which initiates the migration). After the process, all components must agree on one of two outcomes: either the migration was completed successfully, or the migration failed due to a crash, and the source enclave remains operational. Hence, the **adversary's goal** is to create inconsistent views of the migration process among the involved parties, i.e., $V_{SM_S} \neq V_{SDS}$ and/or $V_{SM_S} \neq V_{SM_T}$. Once a migration is triggered by the SDS sending the migration trigger to the SM, the adversary has the following **capabilities** during the migration execution to achieve this goal:

- The adversary controls all network communication exchanged between the SMs and the SDS, i.e., they can drop any message on the channel.
- The adversary controls the target devices running the SMs, i.e., they can delay any messages between the source/destination enclave and the SM.
- The adversary can cause a crash of the SDS, the SMs, and/or the enclaves running on the target devices.

The attack is considered successful if, at the end of the migration process, $V_{SM_S} \neq V_{SDS}$ or $V_{SM_S} \neq V_{SM_T}$, i.e., the source SM has a different view of the migration than the SDS or the destination SM. Synchronized timeouts and message acknowledgments ensure that the view of both SMs and the SDS remains consistent in the face of crashes of either party or loss of messages, caused by the adversary or system failures.

Note that all communication between the source SM, the destination SM, and the SDS is sent over a secure TLS channel. Hence, the adversary cannot modify or forge any messages transmitted between the SDS and/or the SMs to signal the SDS a successful migration despite failure, and vice versa. Similarly, the communication between the enclave and the SM in Keystone is protected from the untrusted host—the adversary—by means of memory isolation, such that the adversary cannot tamper with these messages.

When the SDS sends a migration request, it initiates a timeout t_1 that spans the full migration duration. Only if the SDS receives an acknowledgement (`OK`) from the source SM before the timeout expires, it considers the migration successful ($V_{SDS} = 1$). It only receives this message if the source SM considers the migration successful ($V_{SM_S} = 1$). We assume that the SDS investigates failed migrations to determine if the migration was completed successfully despite t_1 expiring.

The most important aspect is maintaining a consistent view of the migration process between the source and destination SM. This consistency is guaranteed through the timeouts t_4 and t_5 set by the source and destination SM, respectively, alongside a set of acknowledgement messages. After the destination SM signalled its readiness to receive the migrated enclave, the source SM pauses its enclave and sends the code and data to the destination SM. Additionally, it starts a timeout t_4. If it does not receive an acknowledgement from the destination SM before t_4 expires, it considers the migration failed ($V_{SM_S} = 0$), and resumes the source enclave. Otherwise, it attests the destination enclave and destroys the source enclave, considering the migration successful ($V_{SM_S} = 1$). It then signals the destination SM again for it to finalize the migration.

The destination SM only considers the migration successful if it received the message signaling successful destruction of the source enclave. Hence, it can only advance to $V_{SM_T} = 1$ if $V_{SM_S} = 1$. However, an inconsistency may arise when the last message from the source to the destination SM does not reach the destination SM, in which $(V_{SM_S} = 1) \neq (V_{SM_T} = 0)$. We argue that the source SM can raise an alarm to trigger an investigation of the issue if it does not receive an acknowledgement from the destination SM. In summary, we showed that $V_{SM_S} = V_{SDS}$ and $V_{SM_S} = V_{SM_T}$, ensuring that $V_{SM_S} = V_{SM_T} = V_{SDS}$ holds.

Property 2—Authentication: A migration is initiated by the SDS when preparing maintenance or permanent decommissioning of the host device of an enclavized software with the hash H_{E_S}. The SDS is identified by its public key pk_{SDS} while the host, or source device, and the target device are identified by pk_{SM_S} and pk_{SM_T}, respectively. This property guarantees that only an authorized party—i.e., the SDS—can initiate an enclave migration. Hence, the **adversary's goal** is to initiate an unauthorized migration on the target device.

To enforce authorized migrations, the REWIRE secure SW Migration protocol leverages ECDSA signatures from the SDS to authorize migration requests. The SM only executes migration requests that are signed by the SDS. Specifically, the developer signs the migration request with its private key sk_{SDS}, computing a signature σ_{SDS}. Additionally, the migration payload is encrypted with AES and the symmetric session key (k) of a TLS session. We denote the encrypted payload by Enc_{SDS}. The adversary is given **access** to the migration process and may attempt to modify σ_{SDS} and Enc_{SDS}. The source SM (SM_S) then decrypts Enc_{SDS} and verifies the signature using the corresponding public key pk_{SDS}. Upon verification success, SM_S establishes a TLS session to the destination SM (SM_T) specified by the public key included in the request. SM_T then repeats the verification.

The adversary performed a successful attack if the following conditions were met:

1. $Dec(Enc_{SDS}, k) \neq H_{E_S}||pk_{SM_S}||pk_{SM_T}$
2. $ECDSA\_verify(\sigma_{SDS}, Dec(Enc_{SDS}, k)) = 1$

The same conditions apply to communication between SM_S and SM_T. We assume that the ECDSA signature scheme and AES encryption are secure. Hence, the adversary can only modify Enc_{SDS} to decrypt to a meaningful plaintext knowing the correct encryption key k. Similarly, the adversary can only forge a valid signature if they know the SDS's private signing key. We assume that the SDS's (and SM_S's) private keys and the TLS session are uncompromised. Together, the use of encryption and ECDSA signatures ensures that only migrations authenticated by the SDS can be installed.

Property 3—Code integrity Part of the a migration is the transfer of the enclave code C from the trusted source device to the destination device. Formally, the enclave code C' on the destination device must equal the code on the source device, i.e., $C' = C$.

This property ensures that an untrusted entity cannot tamper with the enclave code undetected during the SW Migration process. Thus, the **adversary's objective** is to modify the code during transition such that $C' \neq C$—e.g., to insert a security vulnerability that can be exploited later—without any involved entity detecting the modification. We assume that the adversary has **privileged access** to both the source and destination devices and controls the network communication between them throughout the migration process. The REWIRE secure SW Migration protocol relies on TLS, ECDSA signatures and remote attestation to ensure the integrity of the enclave software during migration.

The TLS protocol provides confidentiality and integrity of transmitted data. Assuming that TLS is secure and the private keys of each party are stored securely, the enclave code cannot be modified during transmission (see details in Property 4). Furthermore, the signed migration request contains the hash H_{E_S} of the enclave code. As shown in Property 2, the request cannot be modified undetected. Before starting the destination enclave, the destination SM validates the received software by comparing its hash against H_{E_S}. This prevents the deployment of incorrect software sent by SM_S. Similarly, the source SM verifies that the destination enclave is deployed correctly (with the correct software) using remote attestation. The attestation prevents long-term deployment of incorrect software by SM_T. Together, the use of TLS, signatures, and remote attestation ensure the integrity of the enclave's software throughout the migration process.

Property 4—State confidentiality & integrity: An enclave may maintain an internal state S, which must be transferred to the destination enclave during the migration process to ensure uninterrupted functionality

across hosts. Formally, the initial state S' of the destination enclave must equal the latest state S of the source enclave, i.e., $S' = S$. Additionally, this state is considered confidential and must remain protected against unauthorized disclosure or tampering.

This property ensures that an adversary cannot obtain information about the internal enclave state S, nor can it modify this state without detection during the migration procedure. Thus, the **adversary's objective** is to violate either confidentiality or integrity of the state, such that $S' \neq S$, or gain knowledge of S during its transition. We assume a powerful adversary model with **privileged access** to the device throughout the migration process, including the ability to modify sealed state data at rest. The REWIRE secure SW Migration protocol leverages Keystone's sealing functionality, hardware-enforced runtime memory isolation via Physical Memory Protection (PMP) and a secure TLS channel to mitigate such threats.

To protect both S and S' at rest, Keystone employs a sealing mechanism that encrypts the state using a symmetric key derivable only by the Security Monitor and the enclave itself. Provided that the underlying cryptographic primitives are secure, this mechanism guarantees confidentiality of the sealed state. Moreover, the sealed state includes a Message Authentication Code (MAC), enabling the SM to detect any unauthorized modifications upon decryption. Any tampering with the sealed data will be detected during MAC verification, thereby preserving state integrity.

During the migration process, the sealed state S is temporarily decrypted into the SM_S's runtime memory before being re-encrypted and delivered to SM_T. During this period, PMP entries are configured to restrict access to this memory region exclusively to the SM. Furthermore, the state is transferred through a TLS channel, preventing all other software entities—including any adversarial code executing at the host level—from accessing or modifying the plaintext state.

Therefore, under the assumptions that (i) the sealing key remains inaccessible to untrusted components, (ii) MAC verification is correctly performed by the SM, (iii) PMP is enforced correctly by the underlying hardware, and (iv) TLS is secure, i.e., provides confidentiality and integrity of all transferred data, the REWIRE Migration protocol ensures that the enclave's internal state remains both confidential and unmodified throughout the migration. Any attempt by an adversary to violate these guarantees will either be prevented (via access control) or detected (via MAC failure), ensuring that $S' = S$ holds with high assurance.

Property 5—Cloning Protection. Depending on the use case of an enclave application, an adversary may benefit from running multiple concurrent instances of the same software. Specifically, the **adversary's goal** is to start three enclave instances E, E', and E'', where E runs on SM_S and E' and E'' run on SM_T. In an instance of a cloning attack, the adversary can split service requests to E and E' or E' and E'', each unaware of the other's inputs, which results in conflicting enclave states. The adversary is granted **privileged access** to the target device, allowing the installation of arbitrary enclave software.

This property guarantees that at most one enclave instance—whether running on the source or destination device—is active at any point in time. During the migration process, the protocol enforces strict sequential execution semantics. Specifically, the source enclave is paused before the destination enclave is launched. If the migration completes successfully, the source enclave is destroyed. Note that we assume that an SM does not allow reinstallation of a previously migrated enclave software unless it is part of another migration request. If the migration fails, the protocol ensures secure destruction of the destination enclave instance and resumption of the source enclave. At no time during this procedure are both the source and destination enclave instances permitted to execute concurrently. Consequently, an adversary cannot launch E in parallel to E'.

Beyond the migration process, REWIRE assumes that the SM enforces a system-wide singleton policy by rejecting any request to initialize a new enclave instance if another enclave with the same hash is already active. This invariant prevents multiple concurrent instantiations of the same enclave software, thereby preventing parallel execution of E' and E''. Together, these mechanisms prevent concurrency-

induced inconsistencies and ensure that enclave state, execution, and control flow remain tightly regulated throughout and beyond the migration lifecycle.

3.5 Discussion

We argue that the design is portable to other TEE architectures, provided that the Root of Trust (RoT) supports memory introspection or equivalent capabilities for measuring and verifying enclave state. For instance, SGX [5] provides mechanisms for enclave measurement and supports models such as forking parent-child enclaves, which allow secure derivation and attestation of enclave lineage. These capabilities align well with our design requirements and are documented in official SGX literature. In contrast, ARM-based TEEs, such as OP-TEE [1], pose greater challenges. While OP-TEE supports secure execution, it lacks a standardized RoT-based measurement mechanism for TEEs. This limitation complicates the implementation of secure update, migration, and introspection features, making direct portability more complex. Addressing this would require extending the OP-TEE framework or integrating external trusted components to provide equivalent measurement and attestation guarantees.

Chapter 4

REWIRE Trust Extensions

This chapter focuses on the final technical specifications and component-level performance evaluations of the new core extensions that constitute the runtime capabilities of the **REWIRE Trusted Computing Base (TCB)**. Building upon the foundational architecture described in previous deliverables, this chapter describes the mechanisms that allow a device to establish its trust level at runtime and to securely re-establish that trust in response to a security incident. The following sections will present the final designs, and initial performance metrics for these critical TCB extensions, while a more in-depth evaluation in the context of the project's use cases will be provided in D6.2 [28].

A primary focus of this chapter is the comprehensive performance evaluation of the **Configuration Integrity Verification (CIV)** scheme. Going beyond the initial documentation in D6.1 [27], we present a detailed benchmarking study of REWIRE's novel implicit attestation scheme implemented across varying types of Trusted Execution Environments (TEEs). This constitutes one of the first detailed evaluations of such advanced attestation capabilities, measuring their overhead when exposed to different Roots of Trust (RoTs) like RISC-V with Keystone [30], x86 with Intel SGX [5], and ARM with OP-TEE [1]. What is particularly noteworthy, as the results demonstrate, is that executing certain cryptographic operations within the trusted, isolated environment of a TEE can, depending on the underlying hardware capabilities, result in greater efficiency than their untrusted counterparts.

Furthermore, we extend this analysis to the microbenchmarking of the schemes responsible for predicating access to critical cryptographic primitives, which are essential for secure lifecycle management. This translates to a detailed evaluation of the **Mirrored Keys** protocol and the instantiation of the **REWIRE LRBC** scheme within a secure enclave. As detailed in D3.2 [25] and D3.3 [24], while LRBC is ideally suited for dedicated hardware, we have emulated its hardware-bound requirements by instantiating it within a second, isolated enclave to measure its performance in a realistic, TCB-mediated environment.

In addition to these benchmarked components, we also sketch the design of the **Process State Runtime Verification** capability. This section outlines the principles behind the DICE-like process that REWIRE follows to generate verifiable certificates for the entire lifecycle of an enclave, positioning it as a key innovation for future development. It is important to note that while this chapter provides detailed component-level benchmarks, the end-to-end evaluation and performance analysis of the verification and migration protocols within the context of the project's use cases will be fully documented in D6.2 [28].

4.1 Preparing a Device to Establish it's Trust Level

Before a device can dynamically attest to its runtime state or participate in secure lifecycle operations, it must first be securely provisioned with the foundational cryptographic material required for trusted interactions. This crucial preparatory phase is accomplished through the **REWIRE Zero-Touch Onboarding (ZTO)** protocol. The ZTO process enables a device to securely join a domain, prove its identity and

attributes to authoritative entities like the Privacy CA and Domain Manager, and in turn, receive the necessary credentials and keys for subsequent runtime operations.

A comprehensive description of the ZTO process is provided in Chapter 6 of this deliverable. The purpose of this section is to position this preparatory phase within the overall runtime framework.

4.2 Enabling a Device to Establish it's Trust Level

4.2.1 REWIRE Implicit Configuration Integrity Verification

The REWIRE framework employs the Configuration Integrity Verification (CIV) scheme as a primary mechanism for ensuring the correctness of a device's runtime configuration. As introduced in D4.2 [26], CIV operates on the principle of *implicit attestation* or *attestation by proof*. In this model, a Verifier can ascertain the integrity of a Prover simply by challenging it with a fresh nonce. The Prover's ability to successfully generate a valid digital signature over this nonce serves as conclusive evidence that its internal configuration adheres to a predefined, correct state. This approach is inherently privacy-preserving and can be considered a form of zero-knowledge attestation, as no sensitive configuration details or measurement traces are ever transmitted to the Verifier; the signature itself constitutes the proof.

The cryptographic mechanism enabling this is the use of a policy-restricted Attestation Key, which acts as the trust anchor for the process. During the Zero-Touch Onboarding phase, the Domain Manager dictates the *Key Restriction Usage Policies* for the device. These policies are then securely deployed within the device's Trusted Computing Base (TCB) and enforced by the Security Monitor. In order to achieve this, REWIRE elevates CIV to Verifiable CIV. Consequently, the Security Monitor will only grant the Prover's enclave access to the private Attestation Key if, and only if, the device's current configuration satisfies these enforced policies. This design elegantly solves the challenge of dynamic state changes, such as software updates, by ensuring that policies are bound to specific, valid configurations, thus preventing rollback attacks where a device might attempt to use an obsolete but previously valid policy.

To validate the practical feasibility of the REWIRE Configuration Integrity Verification (CIV) scheme, its performance was evaluated across a range of hardware platforms. As REWIRE is designed to operate on a heterogeneous ecosystem of edge devices, from resource-constrained IoT sensors to more powerful edge gateways, it is essential to understand the performance characteristics and computational overhead of the CIV's cryptographic operations on different Roots of Trust (RoT). This evaluation provides valuable insights into the trade-offs between security and performance, ensuring the CIV scheme is suitable for the diverse deployment environments envisioned by the project. REWIRE innovates by introducing benchmarks that were conducted on three representative TEE architectures: **Keystone** on a RISC-V platform, **Intel SGX** on an x86 platform, and **OP-TEE** [1] on an ARM TrustZone [18] platform. The evaluation focuses on measuring the execution time of the key cryptographic processes that enable the CIV workflow, which is composed of two primary components: the **Attestation Agent (AA)** and the **Verifiable Policy Enforcer (VPE)**.

Benchmarking Methodology The evaluation measures the time taken for the distinct operational steps within the CIV scheme's two main software components: the **Attestation Agent (AA)** and the **Verifiable Policy Enforcer (VPE)**. The action workflow, representing the intersection of the processes benchmarked across the different platforms, is as follows:

Attestation Agent (AA): This component manages the device's attestation key and responds to challenges. The core measured phases are:

- **Join:** The initial one-time setup where the device, having been securely onboarded, is provisioned with a local attestation key governed by a policy from the Domain Manager.

- **Activate Credential (Verify):** The process where the AA receives an authorized credential from the Domain Manager containing reference values binded to the public part of the Device's Attestation Key. The AA decrypts and verifies this credential to prepare for runtime operations.
- **Policy Reconstruction & Verification:** During a runtime attestation, the AA reconstructs the key usage policy by verifying evidence. This includes checking enclave measurements (**UUIDs**), verifying signatures from other trusted components like the VPE, and validating the runtime traces collected by the system Tracer (**Ticket Verification**).
- **Load Private Key & Sign:** Once all policy checks are satisfied, the AA is permitted to reconstruct or load its private attestation key and use it to sign the challenger's nonce, thereby proving its integrity.
- **Total Runtime:** The cumulative time for all runtime operations following the initial Join phase, representing a full attestation cycle.

Verifiable Policy Enforcer (VPE): This TCB component is responsible for validating that the correct key restriction usage policy is being enforced by the AA.

- **Activate Credential (Verify):** The VPE receives and verifies its own encrypted credentials from the Domain Manager, which include its private key and the reference values of the AA and Tracer.
- **Tracer Verification:** During runtime, the VPE validates the signature on the evidence provided by the system Tracer to ensure its authenticity before forwarding it to the AA to check it against the predefined policy.
- **Load Key & Sign:** After verifying the tracer's evidence, the VPE is permitted to load its own private key and sign an authorization ticket for the AA holding the hash of the key restriction usage policy that needs to be enforced by it. This ticket confirms the authenticity of the evidence collected by the tracer and the correct state of the AA wrapped with the appropriate key restriction usage policies.
- **Total Runtime:** The cumulative time for all runtime operations performed by the VPE during a single attestation cycle.

Unit Test Results The following tables present the benchmarking results for the CIV scheme's components across the different TEE platforms. The **Keystone on RISC-V** benchmarks were performed on a StarFive VisionFive 2 board, featuring a quad-core 1.5 GHz CPU and 4 GB of RAM. The **Intel SGX** evaluation utilized an industrial server with an Intel Xeon processor, running Ubuntu 22.04 and the Gramine framework to enable enclaves. Lastly, the **ARM TrustZone** tests were conducted using the OP-TEE [1] implementation on a representative embedded development board. It is important to note that operations involving communication between the untrusted host and the trusted enclave (via *ocalls* or shared buffers) incur a baseline overhead. As measured on our RISC-V testbed (see Table 4.1), this overhead is approximately 0.2-0.3 ms for a 2KB buffer and is included in the measurements where such communication occurs.

Table 4.1: Communication overhead between trusted and untrusted worlds on the Keystone (RISC-V) platform.

Attestation Agent Enclaved Version				
Measurement	**Min (ms)**	**Max (ms)**	**Avg (ms)**	**StD (ms)**
JOIN	61.5845	62.1818	61.8535	0.1392
VERIFY	561.9310	562.9610	562.5092	0.2625
SEND CHALLENGE	0.2290	0.4898	0.3139	0.0867
RUNTIME	432.6410	433.7455	433.2617	0.2554
Policy UUID	0.0140	0.0155	0.0148	0.0004
Policy Signed VPE	123.4560	123.9958	123.7527	0.1281
Policy OR	0.0148	0.0153	0.0150	0.0002
Policy Signed Tracer	123.3995	123.3995	123.6264	0.1153
Policy Authorize	123.4435	123.8185	123.6001	0.0884
Load Attestation Key	0.0285	0.0300	0.0292	0.0003
SIGN	61.7405	62.0893	61.9010	0.0856

Table 4.2: Benchmarking of the Verifiable Policy Enforcer (VPE) on the Keystone (RISC-V) platform.

VPE Enclaved version				
Measurement	**Min (ms)**	**Max (ms)**	**Avge (ms)**	**StD (ms)**
VERIFY	562.8385	563.5673	563.2141	0.1754
SEND CHALLENGE	0.2178	0.4753	0.2928	0.0794
RUNTIME	185.8598	187.1473	186.4158	0.3282
Tracer Signature Verification	123.6748	124.5555	124.0845	0.2199
Load Key	0.0233	0.0245	0.0238	0.0004
Sign	61.7923	62.3133	62.0773	0.1269

Table 4.3: Benchmarking of the Attestation Agent on the Intel SGX platform.

Attestation Agent Enclaved Version Intel SGX				
Measurement	**Mean(ms)**	**Min(ms)**	**Max(ms)**	**Std(ms)**
AA JOIN	5.77	2.23	27.50	1.21
AA VERIFY	6.73	2.71	9.87	0.05
AA UUID POLICY (MREnclave+MRSigner	0.18	0.116	0.36	0.018
AA POLICY SIGNED VPE	3.31	1.02	3.59	0.45
AA POLICY OR	0.02	0.008	0.10	0.008
AA TRACER VERIFICATION	4.74	2.09	5.63	0.52
AA POLICY AUTHORIZE/AA TICKET VERIFICATION	4.40	1.91	6.26	0.51
AA LOAD & SIGN WITH AK	1.72	0.53	3.18	0.25
AA total JOIN	47.38	37.8	97.73	3.2
AA total RUNTIME	46.72	29.15	61.19	3.17

Table 4.4: Benchmarking of the Attestation Agent on the OP-TEE [1] (ARM TrustZone) platform.

Attestation Agent Enclaved Version ARM				
Measurement	**Mean(ms)**	**Min(ms)**	**Max(ms)**	**StD (ms)**
AA JOIN	207.029	206	208	0.535
AA VERIFY	1636.35	1626	1648	4.802
AA UUID POLICY	19.802	< 1	1	0.139
AA POLICY SIGNED VPE	421.544	419	424	0.777
AA POLICY OR	29.703	< 1	1	0.169
AA TRACER VERIFICATION	421.316	419	423	0.743
AA POLICY AUTHORIZE/AA TICKET VERIFICATION	579.901	< 421	424	0.6175
AA LOAD & SIGN WITH AK	289.138	<210	213	0.5477
AA total JOIN	1867.238	1856.942	1885.178	5.475
AA total RUNTIME	1526.499	1522.609	1530.88	1.308

VPE Enclave version Intel SGX				
Measurement	**Mean(ms)**	**Min(ms)**	**Max(ms)**	**Std(ms)**
VPE VERIFY	7.14	7.01	9.21	0.01
VPE TRACER SIGNATURE VERIFICATION	5.26	5.18	6.45	0.65
VPE LOAD KEY & SIGN	1.76	2.04	1.18	0.003
VPE total RUNTIME	1.81	1.76	2.04	3391.1187684880307e-05

Table 4.5: Category-wise VPE as a trusted application benchmarks in Intel SGX.

VPE Enclave version ARM				
Measurement	**Mean(ms)**	**Min(ms)**	**Max(ms)**	**StD**
VPE VERIFY	1636	1625	1648	0.535
VPE TRACER SIGNATURE VERIFICATION	409.821	407	411	0.236
VPE LOAD KEY & SIGN	255.653	< 206	208	0.789
VPE total RUNTIME	637.507	635.323	639.430	0.820

Table 4.6: Category-wise VPE as a trusted application benchmarks.

The performance evaluation across the three distinct TEE architectures provides a clear and insightful view into the practical application of the REWIRE CIV scheme. As anticipated, the end-to-end measurements show that **Intel SGX** delivers the highest performance by a significant margin. This advantage is largely attributable to two factors: the use of dedicated hardware cryptographic accelerators within the Intel CPU, which drastically reduce the latency of operations like signing and verification, and the maturity of the **Gramine** framework, which is highly optimized to enclave applications with minimal overhead. The performance gap is especially pronounced in the individual cryptographic operations, underscoring the efficiency of Intel's Memory Encryption Engine (MEE). While the MEE provides robust confidentiality guarantees, its proprietary and less modular nature stands in contrast to REWIRE's architectural goals. Furthermore, future Intel technologies like Trust Domain Extensions (TDX), which allow for the isolation of entire virtual infrastructures, promise even greater performance, highlighting the trajectory of hardware-accelerated confidential computing.

On the other hand, the **Keystone on RISC-V** platform, while exhibiting higher latencies, represents the core strategic vision of REWIRE. The current performance is a reflection of its software-based cryptography running on general-purpose hardware (StarFive VisionFive 2) and the relative immaturity of the RISC-V TEE technology stack compared to its commercial counterparts. However, this is not a limitation but a reflection of its potential. Keystone's primary advantage is its fully **open-source architecture**,

CIV Enclaved Version Intel SGX				
Measurement	**Mean(ms)**	**Min(ms)**	**Max(ms)**	**Std(ms)**
CIV total	46.72	29.14	61.11	3.17

Table 4.7: Total CIV with AA and VPE as trusted apps benchmark in SGX.

CIV Enclaved version ARM				
Measurement	**Mean(ms)**	**Min(ms)**	**Max(ms)**	**StD**
CIV total	5997.037	5978.723	6028.792	7.386

Table 4.8: Total CIV with AA and VPE as trusted apps benchmark in ARM.

which provides unparalleled flexibility, customization, and adaptability. Unlike the monolithic design of SGX, Keystone is built on a foundation of modular extensions managed by the Security Monitor. This architecture makes it significantly easier to integrate new hardware accelerators or software optimizations without compromising the core security model. Therefore, while Keystone's current performance on commodity hardware is behind SGX, its open and modular design gives it the potential to match, or even exceed, the performance of proprietary TEEs as the RISC-V ecosystem matures and dedicated cryptographic co-processors become more prevalent.

What is particularly compelling is Keystone's performance in the novel policy-based authorization mechanisms central to the CIV scheme. For instance, when comparing the **policy signing and authorization** steps, Keystone exhibits performance that is highly competitive, especially when measured against a well-established TEE like ARM TrustZone. The overall time for these advanced trustworthiness claims is nearly equivalent, demonstrating that Keystone's architecture is already capable of efficiently handling the complex logic required by REWIRE. This proves that REWIRE's vision of flexible, policy-driven security on an open hardware platform is not only feasible but also performant where it matters most. With access to more powerful RISC-V hardware, these results would undoubtedly improve further, solidifying Keystone as the ideal foundation for the future of auditable, resilient, and open secure computing.

4.2.2 Process State Runtime Verification

In modern production devices, especially edge (and far-edge), arises the need to authenticate that the device operates in a robust and secure way; that is, the device operates as expected without being affected by an exploit that can lead to an unwanted or malevolent behaviour. In the Rewire context, this falls into the threat model described in D2.2 [29]. But most modern devices are not single purpose, in fact they may have multiple processes running simultaneously. So for devices to prove that they are trustworthy, cannot simply rely on their identity-based authentication: simply asking a device for its name, or whether it is in a good state, without requesting proofs-of-correctness of its internal building blocks is not good enough. Hence, there has been a concurrent evolution of how to obscure identity- and attestation-based mechanisms for reporting on the health of a device.

In this context, attestation is a process by which a computing device can produce difficult-to-forge cryptographic evidence showing the state of its operational software. This evidence can then be used by an external entity to verify that the device's software is known and expected on this device, comes from a trusted source, and has not been tampered with on the device. As aforementioned, and is the case also for REWIRE TCB, attestation always depends on a ”chain of trust” model, in which an implicitly-trusted root can be used to validate the lowest-level software, which in turn can be trusted to validate the next, and so on until the device is fully operational.

To effectively ensure device liveliness and correct operation, attestation must also work at a process level, in order to transfer this to statements on the security properties of hierarchical compositions of systems ('Systems-of-Systems'). Furthermore, device identity is an often-overlooked part of the trust

model for attestation of the integrity of software on the device. Networked devices are notoriously hard to identify reliably, so if a compromised device can simply take on the identity and evidence-of-integrity of an untampered unit, the proof is of little value.

It is also plausible, in such multi-layered systems that a software layer of the device is compromised, but another layer, operating in a higher security context stays unaffected, rendering meaningless a sole characterization of the device as a whole about its integrity.

In other words authentication should be based on more granular unique identifiers, which are tied to both the device and to another more granular property of the device, a “function item”, or more generally a state of the device (a process, a key, an item). This approach was first detailed in the DICE Architectures Work Group, as Process State Verification.

DICE operates by bootstrapping layers of firmware, each providing its own attestation facility and its own asymmetric cryptographic attestation identity, implemented in shielded locations. This deviates from having one single RoTs (Root of Trust) and RTR (Root of Trust for Reporting). But this is what allows for this layered attestation at a process level and this is what we are trying to emulate at a software level. More specifically, in the Rewire context we are using Keystone, a TEE that leverages the hardware security features of the RISC-V architecture, such as 3 different privilege levels controlled by PMP (Physical Memory Protection). So, having this in mind PSV will be an elevation of single layer to multi-layer attestation, which aims to provide runtime evidence about the integrity of a binary running in protected memory.

For Keystone, Trusted Hardware is: “Trusted Hardware is a CPU package built by a trustworthy vendor, and must contain Keystone-compatible standard RISC-V cores and root of trust. The hardware may also contain optional features such as cache partitioning, memory encryption, cryptographically-secure source of randomness, etc. The Security Monitor requires platform specific plug-ins for optional feature support.”

In PSV, we consider part of our TCB the trusted Hardware, and the Security Monitor. According to Keystone: “Security Monitor (SM) is M-mode software with small TCB. The SM provides an interface for managing the lifecycle of an enclave as well as for utilizing platform-specific features. The SM enforces most of Keystone’s security guarantees since it manages the isolation boundary between the enclaves and the untrusted OS.”

The Security Monitor, is essentially programmable microcode, which is provided by the developer, has a very specific footprint, has no dynamic allocation of memory (everything is stored in the stack) and is loaded and verified at the very first steps of a system’s boot, akin to secure boot. So, in alignment with DICE, this would be considered part of layer 0 (a figure will be presented later to visualise said security levels).

PSV leverages and expands upon Keystone security mechanisms, to ensure that an enclave is still in the expected state, after the secure launch, by providing attestation evidence to a verifier. This additional guarantee is critical, to reduce the size of our TCB; that is, lessen what we consider trusted, and examine some cases where an attacker, without managing to gain access to the SM (which resides in the highest privilege plane of RISC-V) would be able to affect the behaviour of enclaves that live in protected memory managed by the SM, by leveraging bugs in the higher layers that are exposed for communication, or certain Keystone transitional time phases where PMP permissions change. In short, only trust the measurement extractor, the SM, not whatever resides inside protected memory as a whole.

This proposed protocol differs significantly from the already proposed and implemented CIV (Configuration Integrity Verification), since that investigates and ensures the correctness of the normal/untrusted world of the device, and not of protected enclaves. It also produces information about the device, not about specific processes. In few words, PSV differs from CIV in what it measures (the tracing part), and how it attempts to ensure the device’s integrity.

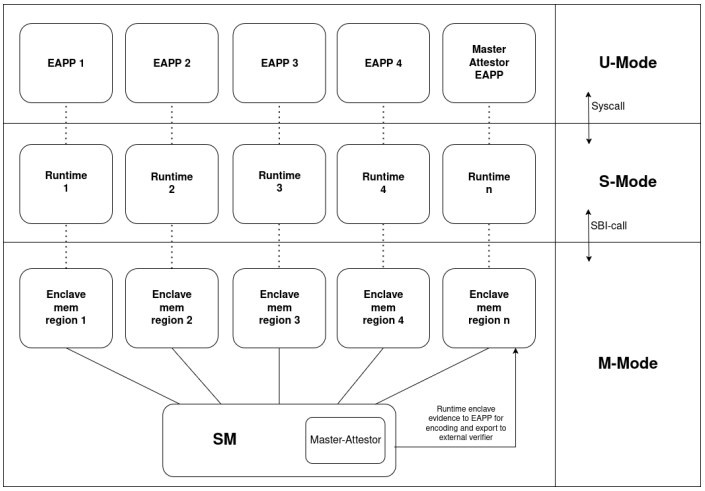

Figure 4.1: PSV System Overview

Let us further analyze Rewire's proposition to address this issue, which is split in 2 major components:

- The runtime measurement extraction
- Encoding the measurement into a suitable cryptographic construct/presentation, suitable for verification by an external actor

Starting with the measurement extraction:

The first step in this process is the creation of a specialized enclave, hereafter referred to as the Master Attestor. Master Attestor has two parts; a Security Monitor function, and an enclave that calls this function, and processes the data created within the SM. The function's primary responsibility is to act as a centralized collector, gathering integrity evidence from all other concurrently active enclaves on the device, by utilizing the existing physical memory mapping of protected regions (enclaves) and extracting measurements from that memory. This privileged function will iterate through all running enclaves, which have already been loaded and measured, and collect critical data. This includes evidence such as the launch-time memory hash for each enclave, a value already maintained by Keystone's core security mechanisms. The collected data is then securely passed from the SM, operating in the highest privilege level (M-Mode), to the Master Attestor enclave running in user space (U-Mode). This cross-privilege-level communication is facilitated by a sequence of Secure Binary Interface (SBI) calls from the SM to the kernel, followed by syscalls from the kernel to the specific user-space enclave application.

The above are also evident on the provided diagram, where the Master-Attestor (a function residing in the SM) traces the allocated active enclave memory regions when requested, and exports that data to the Master-Attestor enclave application (EAPP), the host side of that enclave.

After that, we expand the scope of evidence from static launch-time measurements to dynamic runtime state by collecting appropriate memory evidence:

- Memory Integrity Snapshots: This involves capturing snapshots of memory regions within each target enclave, that under normal operation should be static. Comparing these runtime snapshots against their expected values provides a strong guarantee against memory-tampering attacks that modify an enclave's logic after it has been securely launched.

Following up with the Cryptographic Presentation and Verification:

The final step involves morphing the collected evidence into a secure and verifiable format. All gathered measurements and proofs are consolidated within the Master Attestor enclave. Here, they are assembled into a credential, that is created via the use of a Sealing key, which is the product of kdf procedures, starting from the hash of the SM and a unique hardware device key (UDS). This will finally be packaged in a x.509 certificate, and this process will be shown in detail in the following diagram.

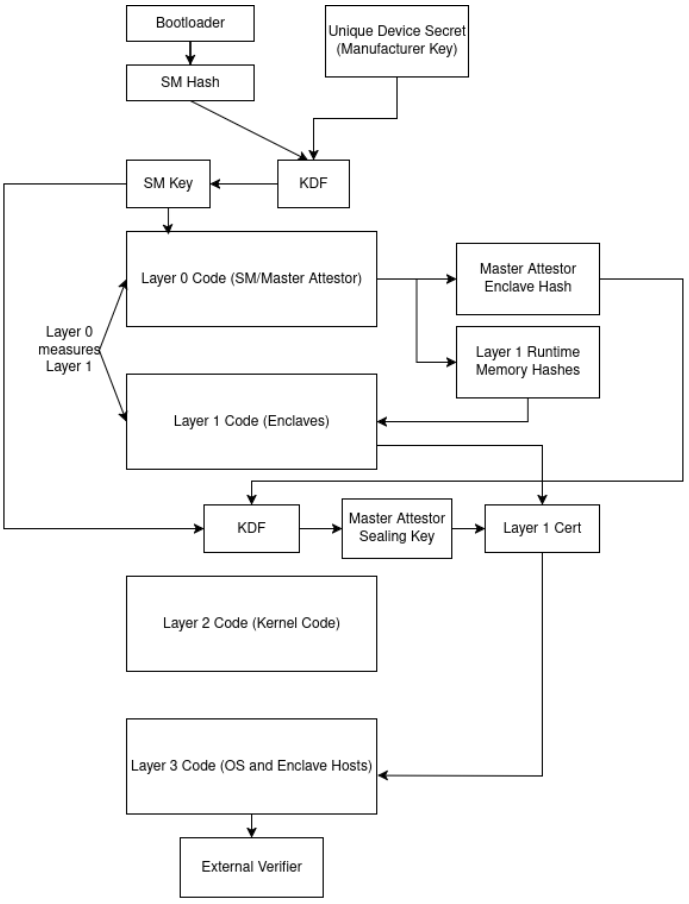

Figure 4.2: PSV Flow

Crucially, this sealing key is not static; it is derived dynamically according to the DICE (Device Identifier Composition Engine) specification. This ensures the key is uniquely tied to both the device's hardware root of trust and the specific software measurement (i.e., the identity) of the Master Attestor enclave itself. This process creates a strong, verifiable chain of trust from the hardware up to the attestation report. The final, signed presentation is then exported from the enclave to the external verifier.

4.3 REWIRE Mechanisms for Authenticated Usage of HW-based Keys

In secure embedded and edge computing environments, it is essential to enforce strict and authenticated usage of hardware-protected cryptographic keys, particularly when those keys are used to safeguard software integrity and update mechanisms. One such key is the Leakage-Resistant Block Cipher (LRBC) Key, which is used to encrypt and authenticate software updates before they are installed on edge devices.

Given its critical role, access to the LRBC Key must be tightly managed to prevent misuse, leakage, or unauthorized use. In the REWIRE project, this access is mediated by a trusted Attestation and Enforcement (AE) Enclave, which acts as a gatekeeper for sensitive cryptographic operations involving the LRBC Key.

To securely bridge the trust boundary between the AE Enclave and other components in the update pipeline—specifically the Software Update Service Stack Enclave and the secure element holding the LRBC Key—we introduce the concept of Mirror Keys. This scheme allows the Security Monitor to create a derived key within its internal key hierarchy that functions as a cryptographic mirror of the LRBC Key. This mirrored key does not replace or expose the original LRBC Key but serves as a controlled intermediary.

By using the Mirror Keys scheme, the system ensures that software updates are encrypted and authenticated through a mechanism that is both enclave-mediated and hardware-anchored, without directly exposing the LRBC Key. This design enforces end-to-end integrity, supports key usage auditing, and upholds the security principles defined in REWIRE's architecture for trusted software lifecycle management on edge devices.

4.3.1 REWIRE Mirror Keys - High Level Overview & Evaluation

As, aforementioned in the deliverable D4.2 [26], the **Mirror Keys** scheme is a lightweight key mediation mechanism designed within the REWIRE project to securely enable software updates on edge devices using the Leakage-Resistant Block Cipher (LRBC) Key. The protocol ensures that access to the LRBC Key—stored in a Secure Element—is only granted to authorized enclaves, without exposing the key material or violating policy constraints.

The protocol is instantiated in two phases:

- A *Set Up Phase*, executed once during onboarding, where the Software Update Service Stack (SUSS) creates and registers its key material.
- A *Mirror Keys Instantiation Phase*, executed per software update, which derives a temporary shared key (Mirror Key) between the Secure Element and the enclave, ensuring policy compliance and authenticated access.

4.3.1.1 Mirror Keys- Evaluation

The Mirror Keys protocol was implemented in `C`, leveraging the **mbedTLS** cryptographic library for elliptic curve operations, key derivation, and symmetric primitives. For trusted execution and enclave isolation, the implementation was deployed atop the **Keystone** framework, enabling a lightweight and modular Trusted Execution Environment (TEE) suitable for embedded and edge platforms.

All experiments were conducted on a **StarFive VisionFive 2** single-board computer, equipped with a **1.5GHz Quad-Core RISC-V CPU** and **4GB of RAM**. This hardware platform represents a typical resource-constrained edge device within the REWIRE architecture.

To ensure statistical consistency and mitigate outliers, each phase of the protocol was executed **1,000 times**, and latency statistics (min, max, average, and standard deviation) were collected. Each row in Table 4.9 corresponds to a distinct interactive step in the Mirror Keys protocol execution flow.

The measurements include the latency of both enclave-bound computations (e.g., policy checks, key derivations) and interactions with the secure element (e.g., nonce exchange, authentication digest validation). Total runtime includes all phases required to instantiate and validate a mirror key during a single software update session.

The majority of execution time is spent in key agreement and validation phases, involving elliptic curve operations, key derivation functions, and secure element interactions. Although the full protocol takes approximately 1.83 seconds on average to complete, this cost is acceptable in the context of infrequent software updates, where strong key isolation and policy enforcement are prioritized over speed. These results confirm that the Mirror Keys scheme is a viable and secure approach for enforcing authenticated access to protected keys during critical software update workflows in REWIRE-enabled edge devices.

4.3.2 Enclave-based LRBC Key Manager & Runtime SW Update Integrity Check

In addition to the Mirror Keys protocol, we evaluated the performance of the **Enclave-based Leakage-Resilient Block Cipher (LRBC) Key Manager** and the **Runtime Software Update Integrity Check**.

Phase	Min (ms)	Max (ms)	Average (ms)	Std Dev (ms)
Initiate mirror key agreement	510.679	513.046	511.995	0.725
Response to initiation	1.286	1.714	1.543	0.2
Finalize mirror key agreement	260.386	266.655	261.214	1.922
Response to finalization	0.798	3.939	1.573	0.988
Respond to initial challenge	1036.423	1042.496	1038.164	1.781
Follow-up response	0.864	1.309	1.207	0.125
Verify mirror key challenge	0.370	0.587	0.534	0.86
Send verification response	0.564	1.179	966.30	0.234
Decrypt ciphertext (LRBC)	7.360	7.887	7.473	0.160
Total Mirror Keys Runtime	**1822.806**	**1845.508**	**1831.099**	**6.660**

Table 4.9: Mirror Keys Protocol Benchmark Results

The LRBC component is a custom flavor of the AES cryptosystem, designed to provide enhanced resilience against key leakage. It integrates tightly with the enclave-based infrastructure provided by Keystone, ensuring that cryptographic key material remains protected during encryption and integrity verification operations.

The runtime integrity check mechanism verifies that the software update process complies with predefined configuration and policy constraints before granting access to protected operations. Both components operate inside the enclave and contribute to enforcing strong guarantees over software authenticity, freshness, and configuration state.

The experimental setup is identical to that used in the Mirror Keys evaluation. All components were implemented in `C` using **mbedTLS**, deployed on the **Keystone** trusted execution environment, and evaluated on a **StarFive VisionFive 2** board (1.5GHz quad-core RISC-V, 4GB RAM). Each operation was executed **1,000 times**, and the latency results were averaged.

As shown in the final row of Table 4.9, the combined execution time for decrypting the received ciphertext using the LRBC algorithm and performing runtime integrity validation averages approximately **7.47 milliseconds**. This includes both the cryptographic decryption step and the policy-enforced verification logic inside the enclave.

This low-latency footprint confirms that the LRBC-based decryption and runtime integrity checks are suitable for update-time validation without introducing noticeable overhead in the edge device's operation.

Chapter 5

REWIRE Harmonised Key Management

Trusted Execution Environments (TEEs) are foundational hardware technologies for enhancing platform security and enabling confidential computing. The integrity and security of any cryptographic system fundamentally hinges on the proper management of its cryptographic keys. If the keys are leaked or become known to an unauthorized entity, even the most formidable cryptographic algorithms offer no protection against data compromise. Therefore, a well-designed Key Management System (KMS) is paramount to the utility of TEEs and fundamental to the operation of the REWIRE TCB. Such a KMS must protect cryptographic keys, regardless of their type, against unauthorized modification and disclosure, and it must manage the secure generation, storage, distribution, operational use, and ultimate destruction of the keys.

However, the proliferation of different TEE technologies has led to a significant fragmentation of capabilities and application programming interfaces (APIs). This forces developers to create platform-specific solutions for key management, hindering portability and increasing complexity. To address this challenge, REWIRE strives to create a harmonized set of interfaces for accessing and managing the cryptographic primitives that are securely stored within the TEE. The goal is to provide a unified abstraction layer that is agnostic to the specific TEE instantiation, whether it be Keystone [30], Intel SGX [5], or ARM TrustZone [18].

In the following, we explore the state of the art in key management for these technologies, argue for the necessity of common cryptographic interfaces for TEEs, and propose the conceptual design for the REWIRE system, which employs such interfaces.

5.1 Security Monitor - Global Interface Definition for Harmonized Key Management

As mentioned above, key management refers to the secure generation, storage, usage, and protection of cryptographic keys. The National Institute of Standards and Technology (NIST) provides comprehensive guidance through its Special Publications, primarily SP 800-57 [20], Recommendation for Key Management, and SP 800-130 [19], a Framework for Designing Cryptographic Key Management Systems (CKMS). The goal of the framework is to guide the CKMS designer in creating a complete and uniform specification of the CKMS that can be used to build, procure, and evaluate the desired CKMS. Both documents synthesize the core principles to outline a robust framework for designing a KMS.

KMSs should be built upon a solid set of principles and guidelines (rules for key generation, protection, distribution, and use). Such guidelines should consider the system requirements or organization's goals for key management, assign roles and responsibilities, and establish the rules of operation. It should be technology-neutral (it should work without introducing significant changes in the current infrastructure) and focus on what needs to be protected and why.

As mentioned earlier, a KMS must keep encryption and signing keys secure throughout their entire lifecycle — from creation to eventual destruction. This lifecycle includes key generation, storage, distribution, usage, rotation, revocation, or destruction (more details are given in Section 5.1.1). In REWIRE, the **KMS is part of the SM, which gives it full control over the keys, its usage, and protection from any other entity in that platform.** This approach increases the TCB size, but ensures that the most critical assets are managed by the most trusted software component.

Keys are associated with metadata that specifies characteristics, constraints, acceptable uses, and parameters applicable to them. For example, a key may be associated with metadata that specifies it's type, how and when it was generated, its owner's identifier, the algorithm for which it is intended, and its cryptoperiod. Like keys, metadata needs to be protected from unauthorized modification and need to be protected from disclosure; the metadata also needs to have its source adequately authenticated. There are many possible metadata elements for a given key, and they depend on the criteria of the designer. The KMS establishes a trusted association between a key and its associated metadata to perform key management functions. For example, a public key is associated with the owner's enclave identifier, so third parties can get these guarantees when it is exported.

When designing a KMS, there are different design patterns that one might consider [21]. The choice of architectural pattern is a foundational decision in designing an efficient, resilient, and secure Cryptographic Key Management System. Different architectural patterns present distinct advantages and challenges, particularly concerning security and scalability. Centralized Systems [21] concentrate processing and data storage on a single central server or a closely connected cluster of servers. It offers simplicity and ease of initial deployment. Decentralized Systems [21] represent an evolution from centralized models, distributing control and processing power among multiple nodes, often spread across different geographical locations, without relying on a single central authority. Each node operates independently, but collaborates to achieve common objectives. However, decentralized systems introduce an increased complexity in terms of coordination and communication among nodes. Distributed Systems [21]represent a further advancement, comprising multiple independent nodes or computers that work collaboratively across a network, appearing to the end-user as a single, coherent system. These systems are specifically designed to maximize performance, reliability, scalability, and resource sharing by leveraging the collective processing power of interconnected devices.

REWIRE uses a centralized pattern, i.e. every device is shipped with its own KMS that manages its own keys. Although one of the disadvantages of centralized systems is that the compromise of the manager may lead to full exposure of the constructed keys, we assume the SM cannot be compromised, as this is the general assumption for TEEs. Nevertheless, the SMs of the different devices can authenticate each other assuming they expose their public key identifier to an external server (equivalent to the SGX model).

A KMS should be compatible with existing approaches or systems. We note that while TEEs and TPMs offer powerful security capabilities, the diversity in their underlying technologies and APIs presents challenges [9]. Since different TEE technologies (e.g., ARM TrustZone-based TEEs like OP-TEE [1], Intel SGX, AMD SEV) and TPM versions (1.2 vs. 2.0) have distinct low-level APIs and interaction models, developers are forced to acquire specialized knowledge for each platform, increasing the learning curve and development effort. **A common set of cryptographic interfaces would offer significant advantages and REWIRE makes the effort to present this unified system.**

Most cryptographic services in REWIRE are built leveraging a cryptographic library (mbedTLS) integrated into the SM - although the designs are agnostic, ther instantiation revolves around mbedTLS which is a well-established crypto library commodity[1]. Thus, operations such as key generation are executed in the most privileged mode, where interrupts are not allowed and all the data is protected by the PMPs that the SM manages. It's important to note that widely used cryptographic libraries have undergone extensive testing and analysis, making them more secure than ad-hoc implementations. Additionally, they follow known good practices for the different operations.

[1] https://mbed-tls.readthedocs.io/en/latest/

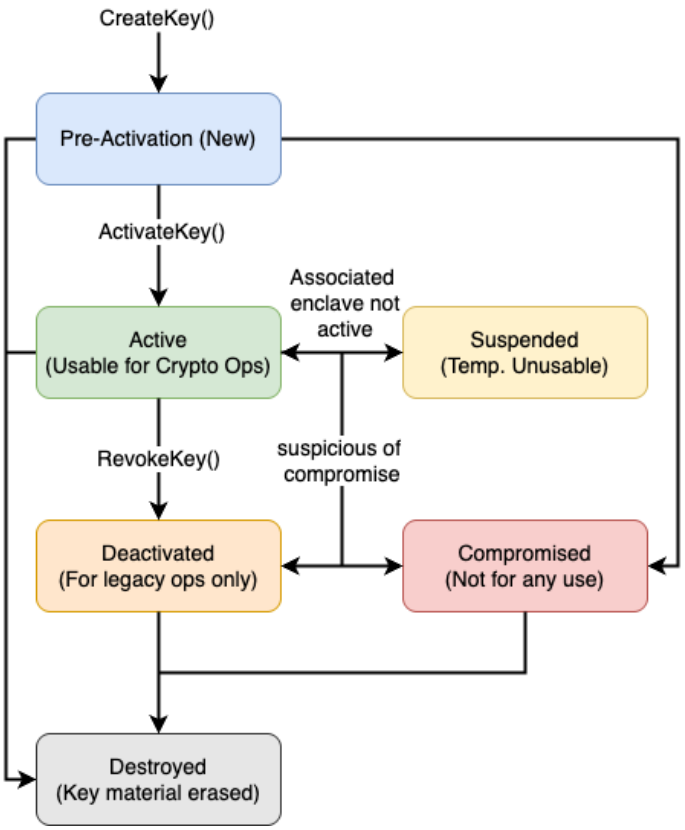

Figure 5.1: Diagram of lifecycle of the Key management system

REWIRE aims to unify all interfaces for key generation and management, as well as for obtaining the results of basic cryptographic operations. Requests will be made with various arguments; some of them will be mandatory, while others will be dependent on the specific operation. The SM performs validation, authorization, and consistency checks before forwarding requests to the cryptographic library. Supported key types include elliptic curve keys for attestation, symmetric keys for sealing, Tracer Keys, update keys (LRBC keys), and migration keys. Further details can be found in Section 5.1.2.

To design and implement the Harmonized Key Management System (HKMS), REWIRE **adopts and extends** the principles of the GlobalPlatform TEE Internal Core API specification [2]. This well-established standard, which defines the API exposed to Trusted Applications, serves as the foundational model for our system. Implementations compliant with this standard, such as the open-source OP-TEE [1] for ARM TrustZone, have provided a valuable reference for our work. However, while we adopt the architectural concepts and functional scope of the GlobalPlatform API [2], our implementation is not a strict one-to-one mapping; we have adapted the nomenclature and interface to align with the Supervisor Binary Interface (SBI) call structure of the REWIRE TCB and have introduced extensions specific to REWIRE's unique security requirements.

5.1.1 Lifecycle

A key passes through several states between its generation and its destruction. The REWIRE CKMS manages its entire lifecycle, conforming to the states defined in NIST SP 800-57 [20]. Figure 5.2 depicts the key states and the transitions among them.

The primary key states include:

- **Pre-Activation / Initialization:** In this initial state, the key material has been generated but is not yet available for normal cryptographic operations. Keys may be newly created or in a preparatory phase, undergoing initial checks, or awaiting policy-driven activation. For example, when an enclave

is created, an identifying keypair is created, but is not available until everything has been properly verified and checked.

- **Active:** The key material is fully operational and available for its intended cryptographic functions, such as encryption, decryption, or signing. This is the primary state during which the key is actively used to protect or process data. Usage should be monitored for any unusual or unauthorized activities that might lead to its deactivation or revocation.
- **Deactivated:** In this state, the key is no longer allowed to aply new cryptographic protection (e.g., new encryption or signing operations). However, it may still be used to process previously protected data (e.g., decryption or signature verification). This state is typically entered when a key has reached the end of its defined cryptoperiod or has been replaced by a new one. This is important after applying updates or patches to enclaves, as the previous keys in a deactivated state could be used for decryption or verification.
- **Suspended:** A temporary state where the use of a key is halted. This is comparable to a reversible revocation and may be necessary if a key's security status is undetermined or if temporary suspension is desired (e.g., for an enclave migration to another host). Depending on the reason for the suspension, a suspended key or key pair may be restored to an active, deactivated, or destroyed state or may transition to the compromised state.
- **Compromised:** This critical state indicates that the key is believed to have been exposed to unauthorized entities. Once a key is deemed compromised, it is considered insecure for any further use in protecting data.
- **Destroyed** / **Erasure**: This is the final state in a key's lifecycle, where the key has been securely zeroized, rendering it permanently unusable and irrecoverable. The destruction process must be verified to ensure no copies remain. It is, however, possible to retain certain information for audit purposes or to be able to detect compromises of the key after its destruction, for compliance requirements, or the ability to decrypt historical data. Access to this information should be highly restricted.

As we describe later, REWIRE uses the metadata associated with each of the keys to manage their lifecycle (the current state is part of such metadata) and to decide whether certain operations are possible or not. For example, if the sealing key is compromised, the SM cannot use it to seal the latest state of the enclave to disk.

5.1.2 Types of Keys in REWIRE Ecosystem

All keys generated for a particular enclave are securely stored by the SM in a protected area, which also contains additional enclave-related metadata. When a request is made to retrieve a previously generated key, it is fetched from this secure storage. *Note that any keys not hardcoded or derived from hardware components, may be lost if the system loses power before they are in a persisted state.* These keys are isolated from other enclaves and the operating system and are never exposed to the untrusted world.

Several keys are used in REWIRE for different purposes and come with their associated metadata. According to NIST [20], the CKMS design must specify a high-level overview of the CKMS system that includes:

1. The use of each key type,
2. Where and how the keys are generated,
3. The metadata elements that are used in a trusted association with each key type,
4. How keys and/or metadata are protected in storage at each entity where they reside,
5. How keys and/or metadata are protected during distribution, and

6. The types of entities to which keys and/or metadata can be delivered (e.g., user, user device, network device).

In the following, we try to clarify the previous points. Items 1 and 2 are covered in Table 5.1. For listing 4, all the keys reside in the SM, so they are protected there and not distributed 5. However, the SMs of two different machines might choose to exchange keys for migration after successful authentication of the respective parties. The same requirements apply for the metadata 6, except that some of them might be manipulated by the enclave owner who might set, for example, a new time for the expiration of the key.

Key	Description and Need
Root ID Key	An hardware-based asymmetric key in the underlying Root of Trust (RoT), embedded at manufacturing time. The Root ID Key (RK) is unique for every RoT, acting as the device identifier. Thus, it cannot be changed, removed, or used outside of the device, and is considered its identifier. These keys might be used to derive others that need binding with the platform.
Attestation Key	Attestation Keys (AKs) are asymmetric identity keys bound to the Root ID Key of the device, and must be certified by the Domain Manager (DM) before being used in the context of an attestation enabler. In case privacy-preserving capabilities need to be added to the attestation process, the AK can be elevated to a Direct Anonymous Attestation (DAA) Key.
Attribute-based Signing and Encryption Keys	These keys are used as part of the Attribute-Based SignCryption (ABSC) protocol. Specifically, for each attribute defined in an attribute-based policy that a device must adhere to in order to be onboarded or interact with the Blockchain infrastructure, each device should be able to ask the Domain Manager (or the appropriate Certification Authority) for the appropriate keys, which enable it to exhibit proof of ownership of the required attributes.
LRBC Key	A hardware-based key for the establishment of an encryption communication channel, over which software updates are securely transmitted.
Sealing Keys of the TEE	In general, in Keystone, sealing keys are derived for enabling storage of data in untrusted, non-volatile memory outside an enclave. Sealing keys are bound to the identity of the processor, the Security Monitor (SM), and the enclave, thus allowing the enclave to derive the same key after an enclave restart so that it can retrieve the data from the untrusted storage and decrypt it using the derived key. In the context of REWIRE, sealing keys are used to ensure the integrity of enclavized applications during update and migration processes.
DH-based Symmetric Keys	When performing a live migration process from one device to a different one, symmetric Diffie-Hellman (DH) keys are generated in order to establish a secure communication channel between the devices.
BBS+ Keys	BBS+ based keys in the context of REWIRE are used to establish authentic and anonymous (if needed) channels between devices, belonging either to the same domain, or to different ones.
Tracer Key	The Tracer Key (TK) is a unique, hardware-rooted cryptographic key managed by the Security Monitor and used exclusively by the REWIRE Tracer to sign the collected attestation evidence, thereby ensuring its integrity and authenticity.

Table 5.1: Keys used in REWIRE Key Management Layer

Crucially, every cryptographic key must be associated with comprehensive Key Metadata. The metadata specify the key's characteristics, operational constraints, and permitted uses. The metadata is considered as critical as the key material itself and demand robust protection against unauthorized modification and disclosure. Furthermore, the source of this metadata must be authenticated to prevent malicious injection or alteration. REWIRE keeps a key table with many entries (`struct sm_key_entry[]`) in the security monitor memory. Each entry includes the following metadata:

- **Label and Identifier:** A human-readable name and a unique identifier for the key.
- **Owner:** The entity (individual or system) responsible for the key.
- **Lifecycle State:** The current operational state of the key (e.g., active, deactivated).
- **Format and Creation Product:** Details about the key's encoding and the system that generated it.

- **Algorithm and Parameters:** The cryptographic algorithm (e.g., AES, RSA, ECC) and any associated schemes or modes (e.g., GCM, CBC) and parameters.
- **Key Length and Security Strength:** The size of the key (e.g., 256-bit AES, 2048-bit RSA) and its assessed security strength.
- **Applications:** The specific applications or systems authorized to use the key.
- **Security Policy Identifier:** A reference to the security policy governing the key's use.
- **Access Control List (ACL):** Defines which users or roles have permission to access or operate on the key or metadata.
- **Usage Count:** The number of times the key has been used for cryptographic operations.
- **Parent Key:** For derived keys, identifying the key from which it was generated.
- **Sensitivity and Protections:** The classification of the data protected by the key and the specific security measures applied to the key itself.
- **Date-Times:** Creation, activation, expiration, and rotation dates.
- **Revocation Reason:** If applicable, the reason for a key's revocation.

5.2 Key Management Design

This section outlines the design and architecture of the Key Management System (KMS). The KMS provides a secure and robust framework for generating, storing, managing, and using cryptographic keys. Its primary design goal is to protect key material by ensuring it is never exposed in plaintext outside of the SM or the enclave itself. If, for any reason, the key is exposed, the system offers the option to revoke it.

The KSM is designed as a software layer that provides cryptographic services to applications, abstracting the complexities of key management and cryptographic operations. It achieves this by creating secure wrappers around the mbedTLS cryptographic library that becomes part of the SM and leveraging the secure memory area only accessible by the SM.

When designing such a system with its layers, we followed these core principles:

- **Zero Plaintext Exposure:** Cryptographic keys in their plaintext form shall never leave the defined cryptographic boundary.
- **Separation of Duties:** The system separates key management functions from application logic. Applications can request cryptographic operations but cannot access the key material directly (except for public keys).
- **Full Lifecycle Management:** The system manages all phases of a key's life, from generation to destruction.
- **Crypto-Agility:** The system is designed to be adaptable to new cryptographic algorithms and standards with minimal changes to the core architecture.

Regarding the use of the different cryptographic algorithms and key sizes, we follow general recommendations and include FIPS-approved or NIST-recommended cryptographic algorithms. They are also reflected in the NIST documents [20, 19]. We do not get into the specifics of the crypto as it is not the focus of this chapter.

Table 5.2 provides descriptions of the core functionalities performed by the REWIRE KMS with their associated interfaces, and it also partially addresses 1 as it describes some of the operations. It is similar to the table presented in a previous deliverable [26], but this time we map the functionality with our implementation instead of with the GlobalPlatform API [2]. We provide further details about the specifics of each SBI call in section 5.2.

Functionality	Description	Related SBI
Key Generation	While some keys for the enclaves are generated when the enclave is created (e.g. attestation keys), some others are created upon their request. This function generates cryptographic keys designed for secure enclaves and applications, using information from a provided structure. The keys adhere to specified cryptographic standards and are securely managed within the Security Monitor, which stores them in a reserved area allocated for that specific enclave. All inputs are validated to ensure that all fields are properly set.	SBI_KMS_GENERATE_KEY, SBI_KMS_DERIVE_KEY
Key Retrieval	Most of the keys never leave the Security Monitor; however, for public key cryptographic schemes, the public key should be shared. Therefore, we provide an interface to retrieve such keys. Note that not only can the enclaves make such requests, but any other application that needs to validate some data coming from the enclave might make use of such an interface.	SBI_KMS_EXPORT_PUBLIC_KEY, SBI_KMS_QUERY_KEY_STATUS, SBI_KMS_GET_SEALING_KEY
Cryptographic Operations	This function performs cryptographic operations such as encryption, decryption, signing, and verification. It supports various cryptographic algorithms and ensures that all operations are securely executed within the Security Monitor. It is mainly intended to create hashes or digests of data, its validation, and signature verification, although encryption and decryption are also supported. Note that, as in previous cases, the inputs are validated before executing any operation.	SBI_KMS_ENCRYPT, SBI_KMS_DECRYPT, SBI_KMS_VERIFY_SIGNATURE, SBI_KMS_SIGN_DATA
Sealing and Unsealing Operations	Although this function also performs cryptographic operations (encryption, decryption, signing, and verification) it is slightly different than the previous ones as the result (data encrypted with a symmetric key) can be directly saved to disc and it is partially binded to the device and identifier of the enclave. Keystone also includes sealing and unsealing functions, and this functionality is partially a wrapper over the functions implemented in Keystone	SBI_KMS_SEAL_DATA, SBI_KMS_UNSEAL_DATA

Table 5.2: Functionalities of the REWIRE Key Management System

5.2.1 Component Descriptions

Since we have already presented the requirements and expected use of the REWIRE HKMS, we focus in this section on the detailed architecture and interactions of its components. Figure 5.2 shows a high-level overview of the components of the Key Management System and its layers. Some components lie in the application layer, some in the Security Monitor, and finally some in the platform or hardware layer

- **Application Layer:** Any user application (enclave) that requires cryptographic services, such as encrypting a file, establishing a TLS session, or verifying a digital signature. It should interact exclusively with the exposed SBI calls to get the guarantees of the management system.
- **CKMS API:** The public-facing interface of the system (SBI calls). It exposes a set of functions for key management and cryptographic operations. Enclaves use key handles to refer to keys managed by the system. The same API is used by the enclave owner in case he needs to update some metadata referring to the enclave
- **Cryptographic Engine (mbedTLS Wrapper):** This component is a secure wrapper around the mbedTLS library. It translates CKMS API calls (e.g., Encrypt, Sign) into specific calls to mbedTLS functions. Crucially, all operations performed by this engine occur within the secure memory boundary, using key material retrieved directly from the Secure Key Store. It communicates with the key life cycle manager to check and update the states of the keys.
- **Key Lifecycle Manager:** The core logic engine of the CKMS. It tracks the state of every key (e.g.,

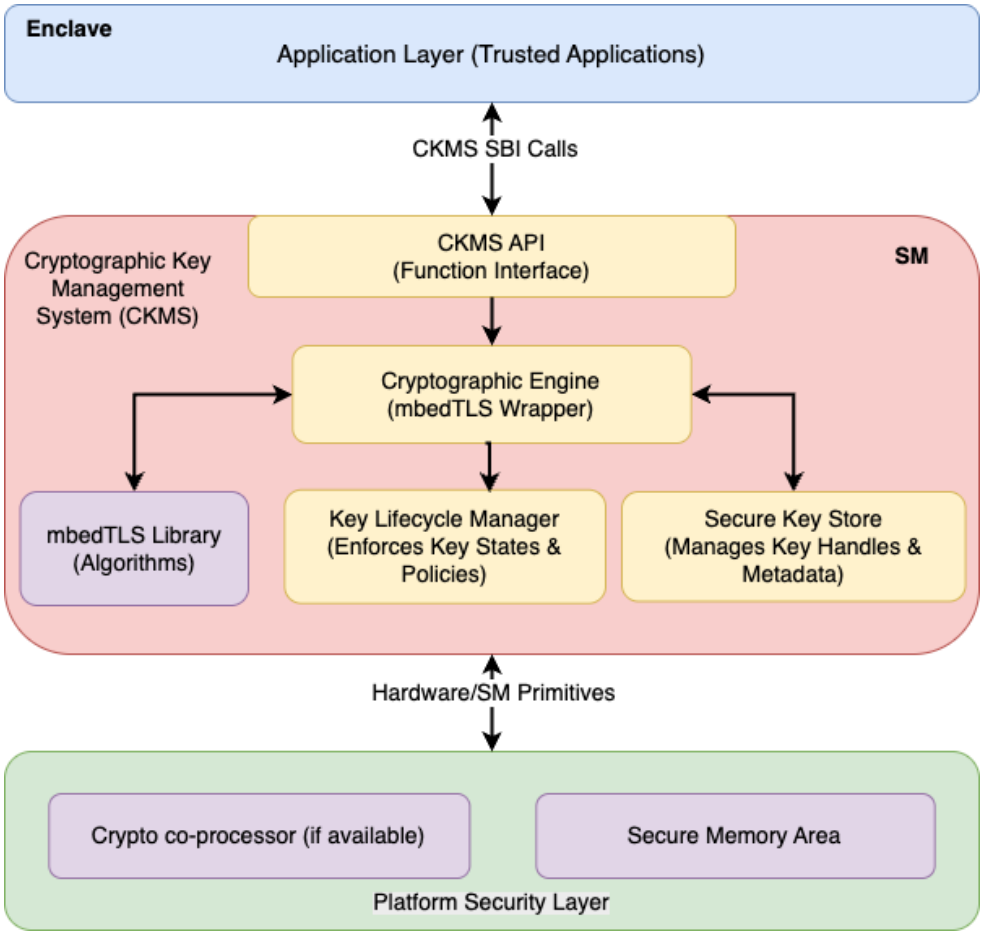

Figure 5.2: Diagram of the components of the Key management system

Active, Suspended, Destroyed) and enforces the rules for state transitions, as recommended by NIST SP 800-57. It ensures that a key can only be used in ways appropriate for its current state.

- **Secure Key Store:** This is the heart of the key protection mechanism. It is responsible for storing and managing all key material and associated metadata (e.g., key type, size, state, usage policy). All keys are protected directly by the SM and kept within its internal memory. If we need persistence, the cryptographic material should be encrypted at rest using a master key only accessible by the SM or stored directly within the hardware-protected memory.
- **mbedTLS Library:** The raw, unmodified cryptographic library that provides the core algorithms. We just compile the modules that we need to use within the KMS.
- **Crypto Co-processor:** Some RISC-V processors might be equipped with such co-processor that executes different algorithms in hardware, isolated from the rest of the components. We detect if there is such a component available at boot time, and if so, the SM will manage it and use it instead of mbedTLS when possible.
- **Secure Memory Area:** A hardware-enforced isolated region of memory. In the case of REWIRE, we do not have a specific dedicated Secure Element (SE), or non-volatile memory such as Trusted Platform Modules (TPMs). We only rely on the PMPs to protect the keys in use. In the absence of a secure or dedicated hard drive, one could persist data with a key managed by an external party that only sends it back to an authenticated SM, though this is a weaker security posture.

5.2.2 SBI Calls

As a result of the design of the Key Management System, we have added different SBI calls to the SM. This section summarizes the exposed SBI calls and briefly describes what they do.

SBI_KMS_GENERATE_KEY: This call generates a new key of specified length for the supported algorithms using the security monitor's secure random number generator (mbedTLS or RoT). The enclave specifies the key size and intended purpose, while the security monitor validates the request and generates fresh key material. The generated key is stored securely within the security monitor's memory space and associated with a unique handle that's returned to the enclave. The enclave uses this handle for subsequent operations, rather than having direct access to the key material. This ensures that keys remain protected within the trusted execution environment while still being available for cryptographic operations. When referring to asymmetric keys, this call generates new ECC key pairs for specific purposes within the enclave. While the enclave gets a master key pair during creation, additional key pairs might be needed for different cryptographic operations or protocols. This enables the enclave to have multiple cryptographic identities for different purposes while maintaining the security guarantees of the trusted execution environment. All the keys created during runtime start in a pre-activation state, and all the metadata described in previous sections is stored securely altogether with the keys.

SBI_KMS_DERIVE_KEY: This call derives keys from existing key material using key derivation functions, for example, AES keys using HKDF or PBKDF2 functions. The enclave provides a master key handle and derivation context, and if the request is properly formed and the enclave is authorized, the security monitor performs the derivation to produce a new key. This is particularly useful for creating session keys, per-operation keys, or hierarchical key structures. The derivation process ensures that related keys are cryptographically linked while maintaining independence for security purposes. The derived key can be used immediately (if it moves to an active state) or stored for later use, with the security monitor handling all the underlying cryptographic operations. The derived key material (handlers) is returned to the calling enclave, ensuring that each enclave gets unique keys even for the same derivation context. This is essential for isolating cryptographic operations between different enclaves.

SBI_KMS_EXPORT_PUBLIC_KEY: This call allows enclaves to retrieve their public keys for sharing with other parties. As with the other calls, the Security Monitor validates the requests, and if everything is OK, it returns the public key portion of the enclave's ECC key pair in a standard format that can be used for key exchange protocols, certificate generation, or identity verification. This is essential for establishing trust relationships between enclaves and enabling secure communication channels in the key management system. These SBI calls form the foundation of the ECC-based key management system within the REWIRE framework. Each call maintains the security boundaries of the trusted execution environment while providing the necessary cryptographic primitives for secure key management operations.

SBI_KMS_ENCRYPT: This call performs encryption using a specified key handle and mode of operation. The key has to be active, and the requester should be authorized. The enclave provides the key handle, plaintext data, initialization vector (if required), and specifies the encryption mode (CBC, GCM, CTR, etc.). The security monitor performs the encryption operation using the securely stored key material and returns the ciphertext to the enclave. This call supports various AES modes, including authenticated encryption modes like GCM that provide both confidentiality and integrity protection. The security monitor handles all the low-level AES operations while keeping the key material secure.

SBI_KMS_DECRYPT: This call performs decryption as the counterpart to the encryption operation. The enclave provides the key handle, ciphertext, and any necessary parameters like initialization vectors or authentication tags. The security monitor performs the decryption and returns the plaintext data. For authenticated encryption modes, the security monitor also verifies the authentication tag and returns an error if the integrity check fails. This ensures that encrypted data hasn't been tampered with during storage or transmission. Note that for decryption, the key could be deactivated, the SM will still proceed with the decryption and issue a warning.

SBI_KMS_SIGN_DATA: This call provides digital signature capabilities using the enclave's ECC private key. The enclave passes a hash of the data to be signed, and the security monitor performs ECDSA signature generation using the enclave's private key. The resulting signature is returned to the calling enclave. This functionality is crucial for authentication and integrity verification in the key management

system. The enclave can use this to sign certificates, authenticate messages, or prove the origin of data. The security monitor handles all the low-level ECC operations while keeping the private key material secure within the monitor's memory space.

SBI_KMS_VERIFY_SIGNATURE: This call allows enclaves to verify ECDSA signatures using the provided public keys. The enclave passes the public key, the original data hash, and the signature to be verified. The security monitor performs the cryptographic verification and returns the result. This is essential for validating signatures from other enclaves or external parties in the key management system. It enables trust establishment and message authentication without requiring the enclave to implement the complex ECC verification algorithms directly.

SBI_KMS_GET_SEALING_KEY: This call derives a sealing key that allows enclaves to encrypt data for persistent storage. The sealing key is derived from the enclave's measurement and the platform's root key material, ensuring that only the same enclave running on the same platform can decrypt the sealed data. This is critical for the key management system as it allows secure storage of sensitive key material across reboots and system restarts. The derived sealing key is deterministic based on the enclave's identity, so the same enclave will always get the same sealing key, enabling consistent access to encrypted storage.

SBI_KMS_SEAL_DATA: This call performs encryption of the given data using the corresponding sealing key of the enclave. The enclave provides the key handle and plaintext data. The security monitor performs the encryption operation using the securely stored key material and returns the ciphertext to the enclave.

SBI_KMS_UNSEAL_DATA: This call is complementary to the previous one. The enclave provides the encrypted data, and the security monitor decrypts that data and returns the plaintext to the enclave. Note that REWIRE keeps track of the updates applied to the enclaves, so it might be the case that an enclave tries to unseal data coming from a previous version that used a different key and it should be able to do so if that key is still in memory.

SBI_KMS_INVALIDATE_KEY: This call marks a specific key as compromised and invalid without immediately destroying it. The enclave provides a key identifier or key handle, and the security monitor updates the key's status to mark it as compromised. This allows for a grace period where the key can still be used for decryption of existing data but cannot be used for new encryption or signing operations. The security monitor maintains a revocation list internally and checks this list before allowing any cryptographic operations. This is particularly useful when a key might be compromised but needs time to transition to new keys or decrypt existing data. The invalidated key remains in memory but is flagged to prevent future use in sensitive operations (e.g., encryption).

SBI_KMS_DESTROY_KEY: This call performs immediate and secure destruction of a specific key. The enclave provides the key identifier, and the security monitor locates the key material in its secure memory and overwrites it with zeros to prevent recovery. This is more aggressive than invalidation and should be used when it is certain that a key is compromised and no longer needed. The security monitor removes all traces of the key from its internal data structures and ensures that any cached copies in registers or temporary storage are also securely wiped. This call is irreversible and should be used with caution since destroyed keys cannot be recovered. The SM keeps some internal identifiers of such keys for audit purposes.

SBI_KMS_REVOKE_ALL_KEYS: This call invalidates all keys associated with a specific enclave when there's evidence of a complete compromise. The security monitor iterates through all keys belonging to the calling enclave and marks them as compromised or destroys them based on the parameters of the request. This is an emergency response mechanism for when an enclave detects that its security has been fundamentally compromised. The call can operate in different modes, such as immediate destruction or gradual invalidation with a grace period. After this call, the enclave would need to regenerate all its cryptographic material from scratch.

SBI_KMS_QUERY_KEY_STATUS: This call allows enclaves to check the current status of their keys. The security monitor returns information about whether keys are valid, invalid, expired, or scheduled

for destruction. This includes metadata about when keys were created, last used, and any pending policy actions. The enclave can use this information to make informed decisions about key management operations and to detect potential security issues. The status information helps with audit trails and compliance requirements while enabling proactive key management.

SBI_KMS_UPDATE_KEY_PARAMS: This call allows enclaves to update some parameters of their keys. The security monitor will update the internal metadata used to manage the lifecycle of the keys. For example, the owner of one specific key might decide to allow more users to use that key or to extend its expiration date if it has not been compromised. This is an authenticated call that might be triggered by the enclave owner.

As it can be seen from the list, there are some calls that focus on the cryptographic operations, and some others that are specifically there for the management of the keys lifecycle.

5.2.3 Example: Sealing and unsealing

As mentioned above, data sealing is the process of encrypting data with a key that is uniquely bound to the specific software enclave that performs the operation and optionally to the device itself. This allows an enclave to store secrets in untrusted memory (e.g., on disk), with the guarantee that only the same enclave can decrypt (unseal) it later.

The Sealing Key is derived at runtime inside the secure memory from a Root Key.

- If a hardware root of trust is available (e.g., a key fused into the CPU), it will be used as the Root Key.
- If no hardware root of trust is detected, the CKMS generates and securely manages a software-based Root Key. This key acts as the foundation of trust for the sealing process on that specific device instance.

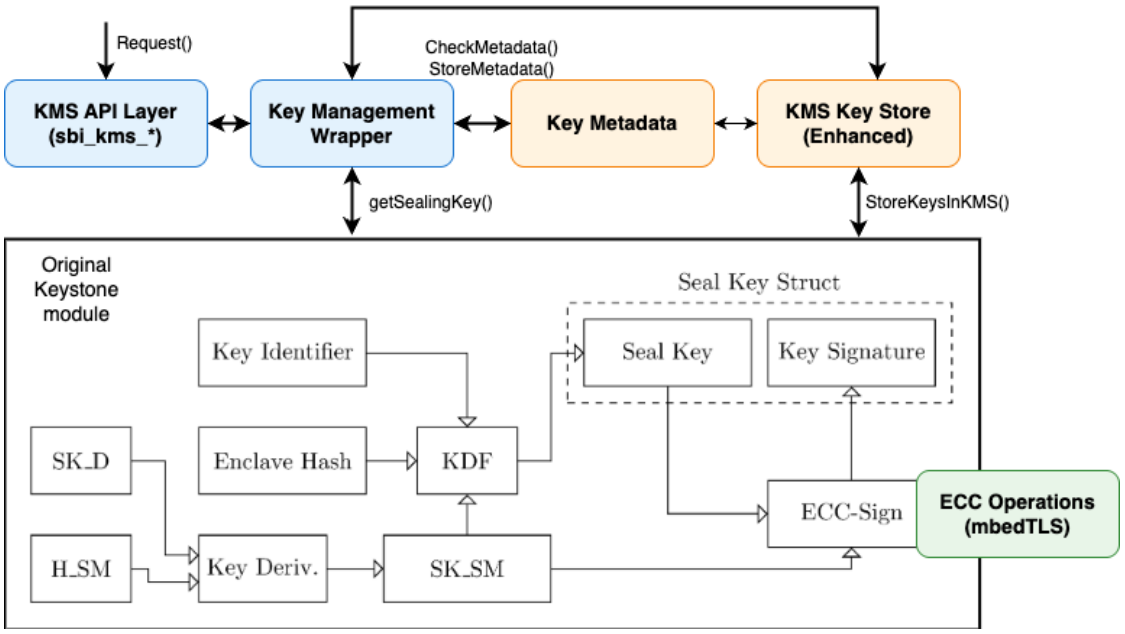

Figure 5.3: Diagram showing the integration of the functionalities of Keystone and the designed Key management system

This Root Key is combined with measurements of the enclave's code and data (its identity) to derive the final Sealing Key, which is never exposed outside the trusted boundary. The *sealing*, *unsealing*, and *getsealingkeys* functions are a special case in the REWIRE KMS as the Keystone project already comes with its own sealing, unsealing and key generation functions. We create wrappers around them, so we

can store the key material and metadata (e.g. when the key was used) within our KMS, but we maintain the same internal logic. The only exception is unsealing from previous versions of the same enclave. In that scenario, we use the previously stored keys instead of the derivation function.

Figure 5.3 shows the whole procedure; all requests are routed through the SBI calls in the KMS, which in turn forwards it to the appropriate components. Some metadata is stored in the internal memory of the KMS and the resulting final key that will be used to seal and unseal data is also stored in the secure area of the Security Monitor. Note that in the figure, the black blocks are the original ones in the Keystone project, whereas the coloured ones are the ones we added.

5.3 Implementation & Evaluation Roadmap

At the time of writing this deliverable, the implementation of the REWIRE Harmonised Key Management System (HKMS), as designed in this chapter, has been finalized. The final technical task remaining is the integration and loading of the ‘mbedtls‘ cryptographic library directly into the Security Monitor, which will complete the TCB-level instantiation of the KMS.

With the implementation phase concluding, the focus now shifts to performance evaluation. To precisely measure the overhead and efficiency of the individual key management functions, we have opted to evaluate the KMS through a series of standalone test cases. This microbenchmarking approach allows us to isolate the performance of each operation, providing clear and accurate latency metrics for the new SBI calls. The complete implementation path report, the final specifications of all SBI calls, and the detailed results of these microbenchmarks will be fully documented in Deliverable D6.2 [28].

Chapter 6

REWIRE Zero-Touch Onboarding

The secure and scalable lifecycle management of devices within a System-of-Systems (SoS) begins with a robust onboarding process. Zero-Touch Onboarding (ZTO) is the mechanism by which a new device establishes its identity, proves its integrity, and acquires the necessary cryptographic material to participate securely in its target domain. It is the critical first step in bootstrapping trust in a zero-trust environment, enabling a device to securely interact with other entities such as the Privacy CA, Domain Manager, and its peers. This chapter details the final cryptographic protocols that constitute the REWIRE ZTO framework, expanding upon the initial specifications in D4.2 [26] and D2.2 [29].

A key innovation in the final REWIRE architecture is the formalisation of two distinct ZTO flows, each tailored to different trust assumptions and operational privacy requirements. The choice between these flows allows REWIRE to adapt to the specific security posture of a given domain, ranging from a standard zero-trust model to a more stringent ”Below Zero Trust” paradigm.

The first flow is designed for environments demanding the highest level of privacy and assurance and operates under a **Below Zero Trust model**. This paradigm addresses a potential architectural gap in Zero Trust where credentials held on an endpoint could be over-exposed in various interactions. In this variant, the Privacy CA issues a comprehensive Verifiable Credential (VC) containing a broad set of the device’s attributes. The REWIRE security extensions running within the device’s Root-of-Trust (RoT) are then trusted to generate Verifiable Presentations (VPs) that *selectively disclose* only the minimal subset of attributes required for a given interaction. This approach provides the strongest privacy guarantees by ensuring that no unnecessary information is ever revealed.

The second flow, grounded in a conventional **Zero-Trust model**, provides strong baseline privacy through a different cryptographic mechanism. This flow assumes that verifying possession of the correct set of credentials is sufficient, without needing the fine-grained, context-specific disclosure of the other model. Here, the Privacy CA issues attributes tailored for the onboarding context, and the device then generates a **zero-knowledge proof (ZKP) of attribute ownership** for the Domain Manager. This ZKP proves that the device possesses the correct, authoritative attributes without revealing their actual values, thus satisfying the ”verify, then trust” principle in a private manner.

This former approach is a key innovation of the REWIRE project and aligns directly with emerging concepts in the IETF RATS working group, particularly the principles for ”Attestation Results for Secure Interactions” [32]. This model acknowledges that endpoint credentials can be manipulated and seeks to fill this gap by creating compound, hardware-rooted credentials that are resistant to spoofing, even by a privileged administrator. By operationalizing this ”Below Zero Trust” model through ZKPs, REWIRE is among the first frameworks to provide a practical solution for scenarios where trust in the runtime integrity of an endpoint cannot be persistently assumed.

The remainder of this chapter provides a detailed examination of these processes.

6.1 REWIRE Zero-Touch Onboarding for Runtime Operational Assurance

A core offering of the REWIRE framework is the provision of mechanisms to equip a device with the required cryptographic material for achieving runtime trust establishment. This is accomplished through a **Zero-Touch Onboarding (ZTO)** process, which is performed when a device enters a new domain. The ZTO mechanism is founded upon a novel and efficient **Attribute-Based Signcryption (ABSC)** scheme, which merges the benefits of Attribute-Based Signatures (ABS) and Attribute-Based Encryption (ABE) to allow a device to verifiably demonstrate its attributes prior to enrolment.

The ZTO process facilitates the device's entry into the service graph chain (SGC) of a target domain after the Design-Time configurations have been finalized and before runtime operations commence. In this context, REWIRE extends the conventional Extensible Authentication Protocol (EAP) by introducing new cryptographic protocols to authenticate and enrol a device based on its proven ownership of specific attributes. This section provides a high-level overview of this operational flow, adapted from the description in D2.2, which sets the stage for the detailed cryptographic protocols and security analyses presented in the subsequent sections of this chapter. Figure 6.1 illustrates the zero-touch onboarding process of an edge device into the REWIRE infrastructure.

The REWIRE ZTO framework supports two distinct variants to accommodate different privacy requirements during the domain enrolment phase:

1. The first variant is based on **Verifiable Credentials (VCs)** and enables strong privacy protection through **selective disclosure**. In this flow, the device can generate Verifiable Presentations (VPs) that contain only a subset of attributes required for a specific interaction, rather than revealing its entire credential.
2. The second variant provides an alternative privacy model where, instead of employing selective disclosure, the device furnishes a **zero-knowledge proof** of all attributes contained within its Verifiable Credential. This proves ownership without revealing the attribute values themselves.

While these two approaches differ in the final domain enrolment step, the initial credential issuance phase is common to both. The complete high-level process is described as follows:

The process begins when a device initializes itself for onboarding by generating an attestation key pair using a predefined pairing-friendly elliptic curve group. With its keys established, the device asynchronously requests separate challenges (nonces) from two authoritative entities: the **Privacy CA** and the **Manufacturer**. Upon receiving both challenges, the device computes two distinct cryptographic digests. The first digest, for the Privacy CA, is a hash of the concatenated challenges. The second, for the Manufacturer, is a hash of the challenges concatenated with the device's public attestation key. To prove its authenticity, the device signs the Manufacturer-specific digest using its unique Root ID Key (a secret known only to the device and Manufacturer) and signs the Privacy CA-specific digest with its private attestation key.

Through a trusted and authenticated communication channel, the Manufacturer validates the device's identity by verifying the HMAC signature with the corresponding Root ID Key. Once authenticated, the Manufacturer signs the device's public key and forwards it, along with the original challenge from the Privacy CA, to the Privacy CA for final verification.

The Privacy CA proceeds to verify the Manufacturer's signature on the device's public key. If successful, it uses this now-validated public key to verify the device's signature over the challenge digest. Successful completion of these steps confirms the device's authenticity. Before generating attribute keys, the Privacy CA fetches the device's attributes from its MUD profile. These attributes are then used to construct a Verifiable Credential (VC) that is cryptographically bound to the device's attestation public key. The resulting VC and associated attribute keys are then transmitted securely back to the device.

After this initial onboarding, the device proceeds to the domain enrolment phase by submitting its desired

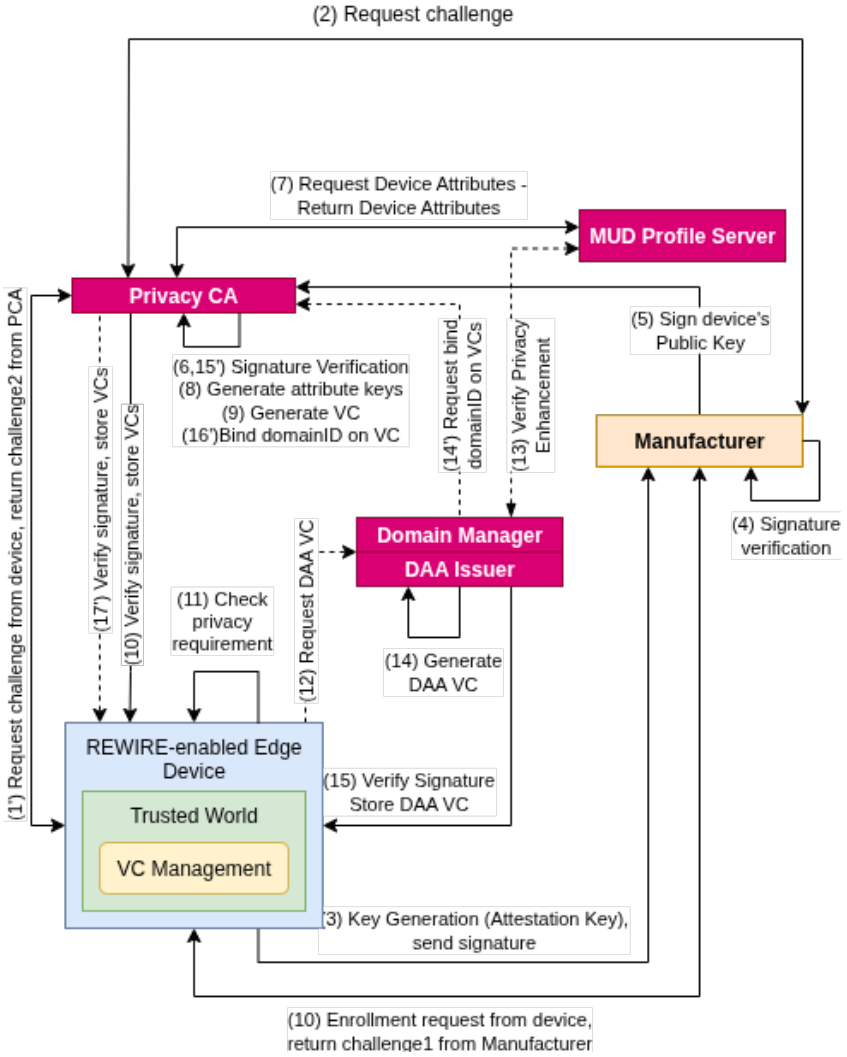

Figure 6.1: REWIRE Zero-Touch Onboarding

Domain ID to the **Domain Manager**. At this stage, the process diverges based on the required privacy flavour:

- **In the first flavour**, aligning with the principle of selective disclosure, the device creates a Verifiable Presentation (VP) containing only a subset of attributes necessary for the verification by the Domain Manager.
- **In the second flavour**, the device creates a proof of knowledge by presenting its VC, disclosing all attributes defined in its MUD profile. This proof is constructed using its attestation key and includes a signature over the Domain ID.

Upon successful verification of either the proof or the VP, the Domain Manager consults the device's MUD profile to determine if further privacy-preserving measures are required. If so, it may forward the device's proof to a Domain Attestation Authority to issue a randomized DAA Verifiable Credential. If not, it requests the Privacy CA to bind the device's Domain ID to its existing VC, and the updated VC is sent back to the device for secure storage.

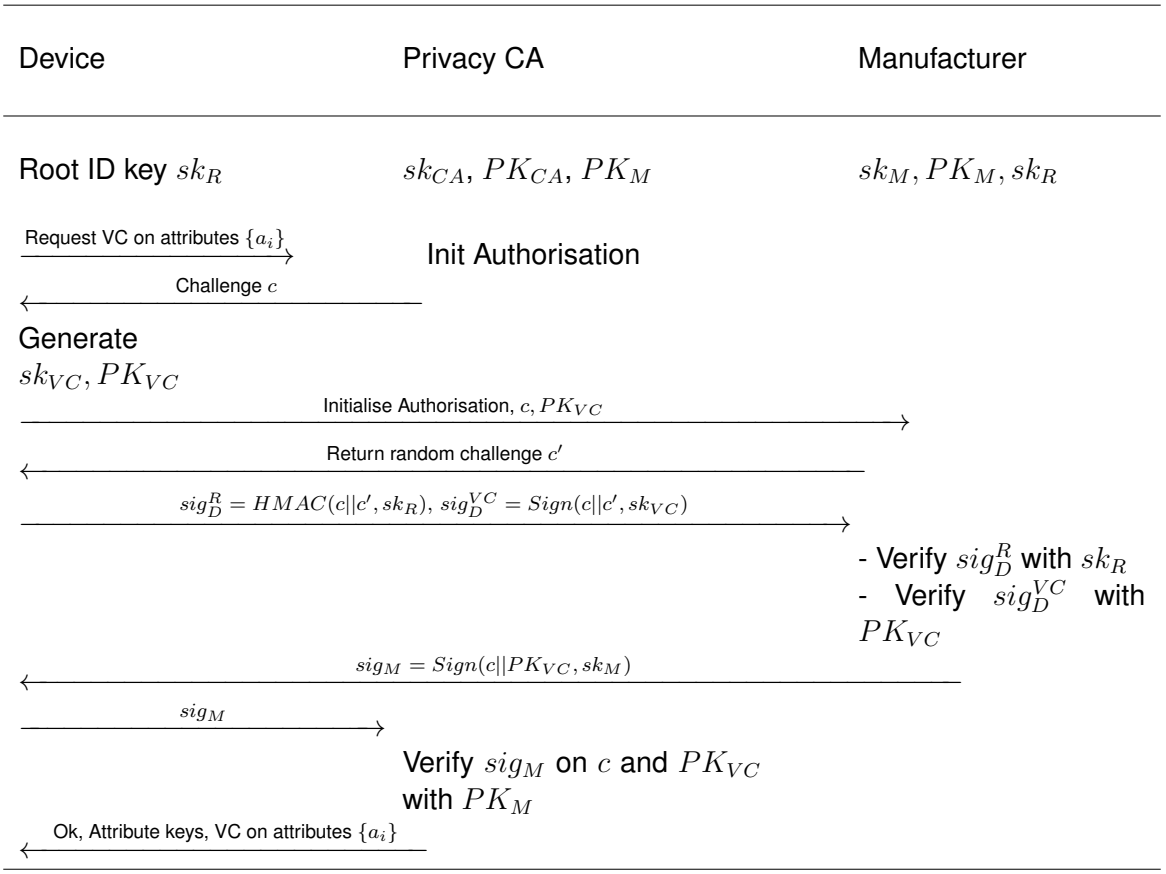

Figure 6.2: Device Authorisation Flow

6.2 REWIRE Zero-Touch Onboarding Based on Selective Disclosure

Expanding on the high-level description of the onboarding and enrolment phases, in what follows we provide the details of the crypto operations that take place at each step of the process. The device will authenticate with the Privacy CA and retrieve a set of attribute keys and a Verifiable Credential over its attributes. To be authenticated by the Domain Manager, the device will generate an Attestation key, which, in the case that privacy is required, it will be converted to a Direct Anonymous Attestation (DAA) Key for enabling enhanced unlinkability of produced Verifiable Presentations. The device will then request the DM's policies hashes, based on which the key restriction usage policies for its Attestation (or DAA) key will be build. To finalise its domain enrolment, the DAA issuer will generate an ABS/ABSC signature (as a credential) based on the policy from the DM and send it to the device.

In REWIRE's targeted use cases, a device will establish a secure and authenticated communication channel with other device within a domain by using Attribute Based Signatures and authenticated ephemeral symmetric encryption keys. On the other hand, communication with devices between different domains will be secured with Attribute Based Signcryption. To enforce the distinction between inner and outer domain communication, the device will retrieve from the Privacy CA an attribute key for the unique identifier of the domain it wants to enrol. This will permit the inclusion of the domain identifier as part of the Attribute Based Signature and Signcryption access policies.

In any case, the device will provide proofs of ownership of the attributes used as part of the attribute based scheme. In the absence of privacy requirements from the Domain Manager, those proofs will be comprised from the device's VC, signed with its Attestation key. On the other hand, if a Domain requires privacy preserving communication, the device will convert its Attestation key to a DAA secret key and

request a DAA credential bound to it, over the same attribute included in its VC. The attribute ownership proofs will then be generated using the DAA Verifiable Presentations.

In the rest of this section, we will examine the authentication of the device by the Privacy CA, with the support of the Manufacturer. Then, we will detail the device's Domain enrolment procedure, making use of Attribute Based Signatures, and depending on the Domain's privacy requirement's, either utilising the device's VC and Attestation key or, in the case that privacy is required, the DAA protocol.

6.2.1 VC Issuance

To be authenticated and retrieve a VC over its attributes, the device will request a random challenge from the VC Issuer, which will return signed by the Manufacturer. On its turn, the Manufacturer will only sign the challenge if the device can showcase possession of its Root ID Key. Furthermore, the device will generate a asymmetric key pair, which, given that the previous steps completed successfully, the Manufacturer will sign, as to allow its inclusion to the verifiable credential. This adds an additional layer of security, by establishing a binding, between the device and its VC. More specifically, the steps with which the device will be authenticated by the VC Issuer in order to receive a credential are the following;

- The device will initiate the authentication process by requesting a random challenge c from the VC Issuer.
- After retrieving c, the device will generate a key pair (sk_{VC}, PK_{VC}), where $PK_{VC} = g^{sk_{VC}}$ and initiate an authentication flow with the Manufacturer, including the returned random challenge c and the public key PK_{VC}.
- The Manufacturer will respond with a new random challenge c'. The device will then use its root ID key sk_R to sign the Manufacturer's challenge c' using $HMAC$, as $sig_D^R = HMAC(c||c', sk_R)$. The device will also use its generated sk_{VC} to sign the received challenge and get the signature $sig_D^{VC} = Sign(c||c', sk_{VC})$. It will return both sig_D^R and sig_D^{VC} to the Manufacturer.
- If able to verify sig_D^R using the device's Root ID key, and sig_D^{VC} using the received PK_{VC}, the Manufacturer will sign the initial challenge c and the aforementioned public key, to produce the signature $sig_M = Sign(c||PK_{VC}, sk_M)$, which they will return to the device.
- The device will return the public key PK_{VC} and the signature sig_M to the Privacy CA, who will first check that they have not issued a VC bound to the received PK_{VC} before. If successful, they will attempt to verify sig_M over the challenge c and the PK_{VC}. If both signatures verify correctly, the Privacy CA will consider the device authenticated, and generate for it the attribute keys SK_{Att}. See Section 3.2 in the deliverable D4.2, for details on the generation of those keys.

Upon successful completion of the above steps, the Privacy CA will generate the VC to be returned to the device, over a set of attributes $Att = \{a_i = H(att_i)\}_{1 \leq i \leq n}$. To that end, it will calculate the BBS signature (A, e), where $e \xleftarrow{\$} \mathbb{Z}_p$ randomly sampled scalar, and $A = (g_0 PK_{VC} \prod_{1 \leq i \leq n} h_i^{a_i})^{1/sk_{CA}+e}$. Finally, the Privacy CA will return the VC=$(Att, (A, e))$ and the attribute keys SK_{Att} to the Device. Note that this information will be structured as a W3C verifiable credential. For simplicity however, we will consider the verifiable credential here as the tuple $(Att, (A, e))$ and provide the details of organising that information as a W3C VC in the following versions of the protocol. To verify the Privacy CA's response, the device will check that $\hat{h}(A, g_2^e PK_{CA}) = \hat{h}(g_0 h^{sk_{VC}} \prod_{1 \leq i \leq n} h_i^{a_i}, g_2)$, where $h^{sk_{VC}} \prod_{1 \leq i \leq n} h_i^{a_i}$ can be public to everyone but sk_{VC} is known by the device and each a_i are known by both the device and the privacy CA, $\hat{h}$ is a type 3 pairing operation.

6.2.2 Domain Enrolment

At an abstract level, the device will first retrieve from the Privacy CA an attribute key (D_d, D'_d) for the **identifier of the Domain** it wants to be enrolled into (ID_{domain}), as to allow the domain to be part of

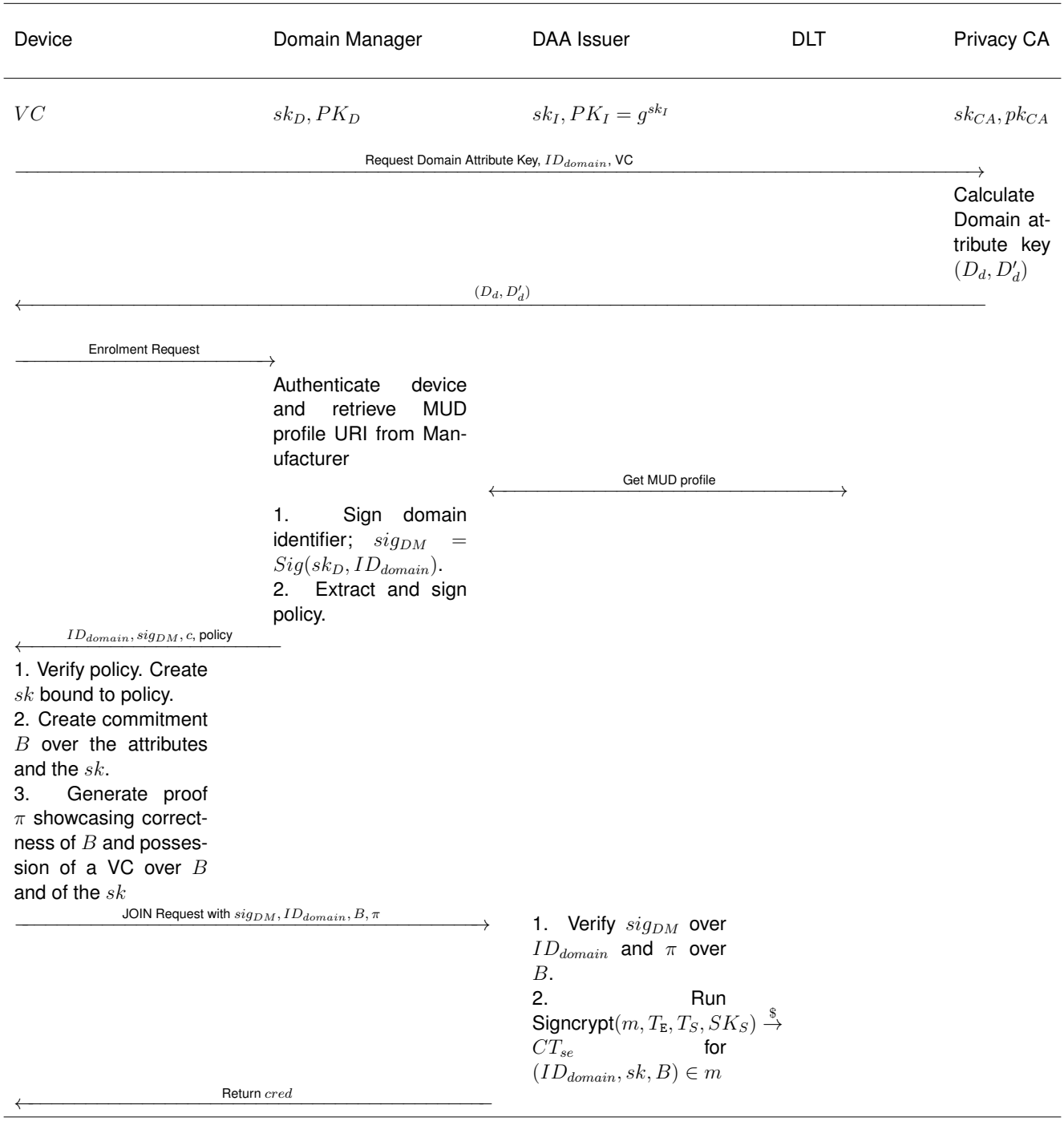

Figure 6.3: Device Enrolment Flow

the Attribute Based Encryption and Signcryption policies (details of the attribute keys generation are presented in Section 3.2 in the deliverable D4.2). Then, it will request the Domain's privacy policies, based on which its key restriction usage policies will be build. To finalise its Domain enrolment, the device will prove that it has the required characteristics, by constructing an Attribute Based Signature, using the Attribute Keys retrieved during the VC Issuance phase of the protocol and an access policy issued by the Domain Manager.

Furthermore, the device will provide evidence of ownership of the attributes used, by utilising either their VC or DAA credential, depending on the Domain's privacy requirements. If privacy is not required, those evidence will be generated using the device's already issued VC. On the other hand, if privacy is required, the device will convert its Attestation key to a DAA key, and execute the DAA $JOIN$ protocol, to recover a DAA credential bound to that key, over the same attributes included in the VC. Note that, besides the

device's DAA credential, each enclave running on that device will require a DAA credential on its own, as to allow for the layered certification architecture.

The steps with which the device will request and retrieve its Domain attribute key and the privacy policies are the following:

- The device will first request the domain attribute key (D_d, D'_d) from the Privacy CA by submitting it's VC and ID_{domain} for the Domain it wants to be enrolled.
- The Privacy CA will verify the received VC, retrieve the necessary, device specific, information for the attribute key generation and calculate the key (D_d, D'_d), which will be returned to the device.
- After receiving the domain attribute key from the VC Issuer, the device will initiate the enrolment request to the Domain manager by submitting its device identifier ID_{device}, signed by the manufacturer.
- The Domain Manager, will need to retrieve the device's MUD profile. To do that, the steps are the following;
    1. After the Domain Manager verifies the device's identifier, will return a random challenge c to the device.
    2. The device will sign and return the received challenge, using its root ID key sk_R, to produce the signature $sig_D(c)$.
    3. The Domain Manager will then request the device specific MUD profile UID ID_{MUD} from the manufacturer, using the device identifier ID_{device}, the challenge c and the received signature $sig_D(c)$.
    4. If the manufacturer is able to verify sig_D on c using the sk_R corresponding to ID_{device}, will return ID_{MUD} to the Domain Manager.
    5. With that, the Domain Manager will retrieve from the DLT the MUD profile of the device, from which it will extract the device policy hashes, in which it will append its own privacy policy hash.
- If the above steps execute successfully, the Domain Manager will return the generated policy and the domain identifier ID_{domain}, signed to the device.

After receiving the Domain Manager's response, the device will attempt to validate the received policy, and if successful, to extract the Domain's privacy requirements. The device will then generate its Attestation key, which, in the presence of privacy requirements, will be converted to a DAA key. In both cases, the device will bind the generated key to the access key restriction policies, constructed from the policy hashes received from the Domain Manager.

In the case that the Domain does not require privacy preserving communication, the device will generate an Attribute Based Signature over the Domain Manager's challenge c, using its attribute keys and the Domain specific access tree. To finalise its enrolment to the Domain, the device will submit the resulting signature and their VC, signed with both its VC key sk_{VC} and its Attestation key to the Domain Manager. The device will be enrolled to the Domain if the Domain Manager is able to verify the VC, the signature from the device's Attestation key and the received Attribute Based Signature.

For the rest of this section, we will examine the case where the Domain Manager established privacy requirements. In this case, the device will convert its Attestation key to a DAA key sk_{DAA}. It will then need to prove to the DAA issuer that it possess a set of attributes, signed by the Privacy CA, without revealing their values, while at the same time allowing the issuance of a DAA credential over both these attributes, as well as the unique domain identifier.

At the start of the DAA JOIN flow, the device will generate a commitment over its attribute set $\{a_i\}_{1 \leq i \leq n}$ and its DAA key sk_{DAA}, by calculating

$$B = h_0^s g^{sk_{DAA}} \prod_{1 \leq i \leq n} h_i^{a_i}$$

for randomly sampled $s \xleftarrow{\$} \mathbb{Z}_p$. It will then showcase the correct construction of B by generating a zero-knowledge proof-of-knowledge π_1 as follows;

$$\pi_1 = SPK\{(s, sk_{DAA}, a_1, ..., a_n) : B = h_0^s g^{sk_{DAA}} \prod_{1 \leq i \leq n} h_i^{a_i}\}$$

To prove that the device also possesses a VC over the attributes $\{a_i\}$, it will first choose random scalars $\tilde{a}_1, \tilde{a}_2 \xleftarrow{\$} \mathbb{Z}_p$ and then calculates $\bar{A} = A^{\tilde{a}_1 \tilde{a}_2}$, $D = (g_0 h_k^{sk_{tpm}} \prod_{1 \leq i \leq n} h_i^{a_i})^{\tilde{a}_2}$, $K = D^{\tilde{a}_1} \bar{A}^{-e}$, $\tilde{a}_2' = 1/\tilde{a}_2$. Here, in a DAA protocol, there are two signers, one is the TPM holding the secret key sk_{tpm} and the other one is helper signer host holding the credential. Here, if we consider the device as one signer, $sk_{tpm} = sk_{DAA}$. Finally, the device will generate the proof π_2 as follows;

$$\pi_2 = SPK\{(e, sk_{tpm}, \tilde{a}_1, \tilde{a}_2', a_1, ..., a_n) : K = \bar{A}^{-e} D^{\tilde{a}_1} \wedge g_0 = D^{\tilde{a}_2'} h_k^{-sk_{tpm}} \prod_{1 \leq i \leq n} h_i^{-a_i}\}$$

Note that the two proofs will be generated as to allow the DAA Issuer to verify their AND composition, i.e., that $\pi = \pi_1 \wedge \pi_2$ is valid, or more specifically, that the proof π is valid, where π is calculated as follows;

$$\pi = SPK\{(s, sk_{DAA}, e, \tilde{a}_1, \tilde{a}_2', a_1, ..., a_n) :$$
$$B = h_0^s g^{sk_{DAA}} \prod_{1 \leq i \leq n} h_i^{a_i} \wedge K = \bar{A}^{-e} D^{\tilde{a}_1} \wedge g_0 = D^{\tilde{a}_2'} h_k^{-sk_{DAA}} \prod_{1 \leq i \leq n} h_i^{-a_i}\}$$

Finally the device will generate a JOIN request as follows;

- The device will submit to the DAA Issuer the request $(B, \bar{A}, K, D, \pi, ID_{domain}, sig_{DM}(ID_{domain}))$.
- The DAA Issuer will verify the request by checking that $\hat{h}(\bar{A}, PK_{DAA}) = \hat{h}(K, g_2)$, verifying π and finally verifying the Domain Manager's signature $sig_{DM}(ID_{domain})$ over the domain identifier ID_{domain}.
- If successful, the DAA Issuer will generate a DAA credential by calculating $A_{DAA} = (g_0 h_d^{H(ID_{domain})} B)^{1/sk_I + e_{DAA}}$, where $e_{DAA} \xleftarrow{\$} \mathbb{Z}_p$, and return (A_{DAA}, e_{DAA}) to the device.

If the above steps complete successfully, the device will retrieve a DAA credential $cred = (A_{DAA}, e_{DAA})$, which will verify by checking that $\hat{h}(A_{DAA}, g_2^{e_{DAA}} PK_I) = \hat{h}(g_0 h_0^s g^{sk_{DAA}} h_d^{H(ID_{domain})} \prod_{1 \leq i \leq n} h_i^{a_i}, g_2)$.

To finalise its enrolment procedure, the device will generate an ABS signature using the required policy for that domain, wrapped with a Verifiable Presentation, generated with its received DAA credential. If the Domain Manager is able to verify both the Attribute-based Signature and the DAA Verifiable Presentation, the device will be enrolled to the Domain.

6.3 Design of Attribute-based Signcryption

Before introducing the proposed ABS/ABSC scheme, we list all notations for inputs/outputs of all algorithms in the Table 6.1.

As we mentioned before, during the domain enrolment procedure, we can make use of ABS/ABSC as a credential. As ABSC is an extension of ABS, we firstly introduce the proposed ABS and then present the ABSC scheme to establish a trust relationship for different REWIRE domains.

Table 6.1: Notation used in the ABS/ABSC scheme

Notation	Meaning
msk	master secret key
mpk	master public key
(M, π) in ABS/ABS	policy for the sender, M is the matrix, π is a set of attributes used in signing
i	the index of attribute
$\mathcal{U}$	the attribute universe
$\mathcal{S}$ in ABS/ABSC	attribute set for sender's key generation
$\mathcal{S}'$ in ABS	attribute set for receiver's key generation
sk	secret signing key associated with attributes
msg	the message to be signed
σ	the ABS/ABSC signature
(M', π') in ABSC	policy for the receiver, M' is the matrix, π' is a set of attributes used in signing

The workflow of ABS. When the communication is within the same domain, the ABS signature is needed. As shown in Figure 6.4, the DAA issuer generates a signature σ signed on the message msg by using the secret key sk, a set of attributes $\mathcal{S}$ and the corresponding policy (M, π). Then the DAA issuer sends the signature σ (also considered as the credential $cred$ in Figure 6.3) to the device. After receiving the signature σ, the device can run the verify algorithm Verify$(\mathsf{mpk}, (M, \pi), \sigma, \mathsf{msg})$. If the output is "1", it means the device accepts the signature σ ($cred$). Otherwise, the device rejects the signature σ.

The details of the proposed ABS scheme are as follows:

- Setup$(\lambda) \rightarrow (\mathsf{mpk}, \mathsf{msk})$: This setup algorithm is run by the DAA issuer by taking the security parameter λ as input and outputting the master public/secret key pair. Pick $\alpha \leftarrow_\$ \mathbb{Z}_p$, $\mathsf{H}_0 : \{0,1\}^* \rightarrow \mathbb{Z}_p$ and $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$, define $\mathcal{G} := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$. Finally, the master public key $\mathsf{mpk} := (\mathcal{G}, \mathsf{H}_0, \mathsf{H}_1, X = e(g_1, g_2)^\alpha)$, the master secret key msk := α.
- KeyGen$(\mathsf{msk}, (M, \pi), \mathcal{S}) \rightarrow \mathsf{sk}$: This key generation algorithm is run between the CA and the device. The CA selects the attributes assigned to the device and sends every attribute $\pi(i) \in \mathcal{S}$ to the device. The device computes $\sigma_i = \mathsf{H}_1(i||h_m||\pi(i))^x$, where $h_m = \mathsf{H}_0(M)$, (x, g_2^x) is the device's root private/public key pair. Then the CA verifies each σ_i under the g_2^x. If the verification fails, aborts. Otherwise, picks $r \leftarrow_\$ \mathbb{Z}_p$, $\vec{V} \leftarrow_\$ \mathbb{Z}_p^{n_2-1}$ and computes $\mathsf{sk}_1 := g_2^r$, $\mathsf{sk}_{2,i} := g_1^{M_i(\alpha||\vec{V})^\mathsf{T}} \cdot \sigma_i^r$ for each row $i \in \mathcal{S}$. Finally, outputs $\mathsf{sk} := (\mathsf{sk}_1, (\mathsf{sk}_{2,i})_{i\in\mathcal{S}})$.
- Sign$(\mathsf{sk}, \mathcal{S} \subseteq \mathcal{U}, (M, \pi), \mathsf{msg}) \rightarrow \sigma$: This signing algorithm is run by the DAA issuer.
  1. Picks $s, t, r_\alpha, (r_i)_{i\in[n_1]} \leftarrow_\$ \mathbb{Z}_p$;
  2. computes $h_m = \mathsf{H}_0(M)$, $\sigma_1 := \mathsf{sk}_1^{xt}$, $\sigma_2 := \prod_{i\in\mathcal{I}}(\mathsf{sk}_{2,i})^{\gamma_i s}$, $\sigma_3 := \prod_{i\in\mathcal{S}} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s} = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s}$, $\sigma_4 := g_2^t$, $Y = X^{s\cdot t}$, $Z = X^{r_\alpha}$;
  3. $c = \mathsf{H}_0(\sigma_1, \sigma_2, \sigma_3, \sigma_4, Y, Z, W, \mathsf{msg})$;
  4. $s_\alpha = r_\alpha - s \cdot t \cdot c$, $\forall_{i\in\mathcal{S}}$, $s_i = r_i - \gamma_i \cdot s \cdot c$, $\forall_{i\in[n_1]\setminus\mathcal{I}}, s_i = r_i$;
  5. Outputs $\sigma := (\sigma_1, \sigma_2, \sigma_3, \sigma_4, c, s_\alpha, (s_i)_{i\in[n_1']})$
- Verify$(\mathsf{mpk}, (M, \pi), \sigma, \mathsf{msg}) \rightarrow 0/1$: This verification algorithm is run by the device.
  1. $h_m = \mathsf{H}_0(M)$;

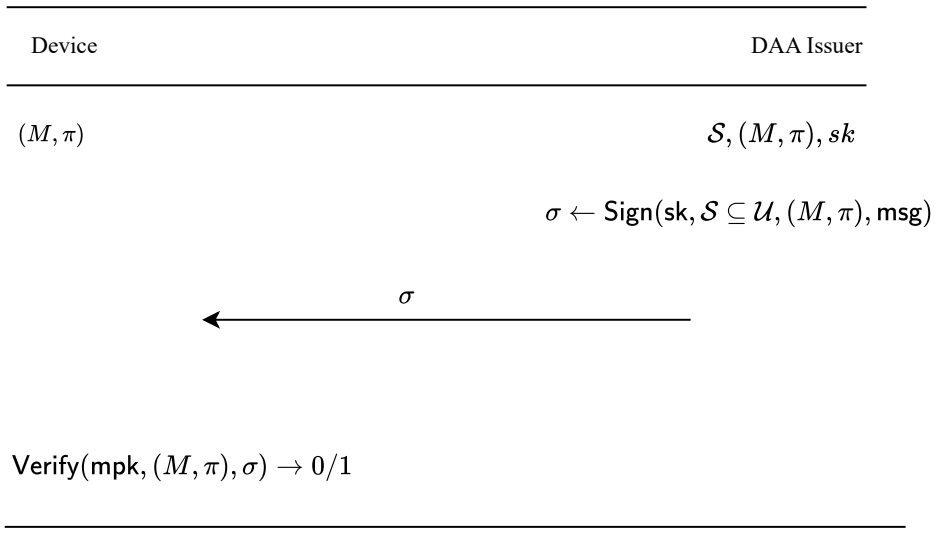

Figure 6.4: The workflow of ABS between the device and the DAA issuer

2. $Y' = e(\sigma_2, \sigma_4)/e(\sigma_3, \sigma_1)$;
3. $Z' = X^{s_\alpha} \cdot Y^c$ and $W' = \prod_{i \in [n_1]} \mathsf{H}(i||h_m||\pi(i))^{s_i} \cdot \sigma_3^c$;
4. $c' = \mathsf{H}_0(\sigma_1, \sigma_2, \sigma_3, \sigma_4, Y', Z', W', \mathsf{msg})$;
5. If $c' \neq c$, output 0; otherwise, output 1.

The workflow of ABSC. When the communication happened among different domains, the ABSC signature is necessary to generate a signature and encrypt the message to ensure the message confidentiality. Following the workflow in Figure 6.3, here, we give the workflow of ABSC between the device and the DAA issuer in Figure 6.5. Similar to the workflow of the ABS, in ABSC, the DAA issuer generates a signature σ signed on the message msg by using the secret key sk, two set of attributes $(\mathcal{S}, (\mathcal{S}')$ and the corresponding policies (M, π) and (M', π'). Then the DAA issuer sends the signature σ (also considered as the credential $cred$ in Figure 6.3) to the device. After receiving the signature σ, the device can run the verify decryption algorithm VerifyDec. If the output is "1", it means the device accepts the signature σ ($cred$). Otherwise, the device rejects the signature σ.

The ABSC scheme supports the signer's attribute anonymity, preventing a verifier from learning the attribute set used to create an ABS. In this scheme, the signer encrypts the signed message (attestation data) under a certain policy through an attribute-based encryption (ABE) scheme. The decryptor should satisfy the decryption policy to allow a successful decryption of the ciphertext.

- $\mathsf{Setup}(1^\lambda) \rightarrow (\mathsf{mpk}, \mathsf{msk})$. It takes the security parameter λ as input. The issuer picks $\alpha \leftarrow_\$ \mathbb{Z}_p$, $\mathsf{H}_0 : \{0,1\}^* \rightarrow \mathbb{Z}_p$ and $\mathsf{H}_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$. The issuer publishes $\mathcal{G} := (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ and its master public key $\mathsf{mpk} := (\mathcal{G}, \mathsf{H}_0, \mathsf{H}_1, X = e(g_1, g_2)^\alpha)$. The master secret key is msk:= α.
- $\mathsf{sKeyGen}(\mathsf{msk}, (M, \pi), \mathcal{S}) \rightarrow \mathsf{sk}$. The ABSC key generation algorithm outputs a set of attribute keys for someone who plays the role of the sender. The CA sends a set of attributes $\mathcal{S}$ to be certified for the user. Then the user with a root key $(SK = x, PK = g_2^x)$, replies with $\sigma_i = \mathsf{H}_1(i||h_m||\pi(i))^x$, where $h_m = \mathsf{H}_0(M)$ for all $i \in \mathcal{S}$, where $\mathcal{S}$ represents the set of indices of the attribute in $\mathcal{S}$. The Privacy CA verifies each σ_i under PK. Then proceeds as follows: The CA picks $r \leftarrow_\$ \mathbb{Z}_p$, $\vec{V} \leftarrow_\$ \mathbb{Z}_p^{n_2 - 1}$ and computes $h_m = \mathsf{H}_0(M)$ and the attribute keys $\mathsf{sk}_1 := g_2^r$. $\forall_{i \in \mathcal{S}}\ \mathsf{sk}_{2,i} := g_1^{M_i(\alpha||\vec{V})^\intercal} \cdot \sigma_i^r$. Finally, the CA outputs $\mathsf{sk} := (\mathsf{sk}_1, (\mathsf{sk}_{2,i})_{i \in \mathcal{S}})$.
- $\mathsf{rKeyGen}(\mathsf{msk}, (M', \pi')) \rightarrow \mathsf{sk}'$. This ABSC key generation algorithm outputs a set of attribute keys for someone who plays the receiver role. The Privacy CA sends a set of attributes $\mathcal{S}'$ to be

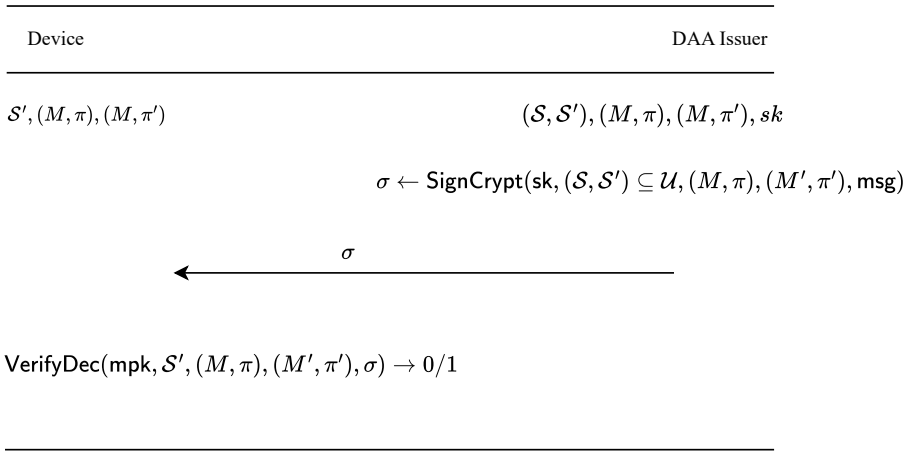

Figure 6.5: The workflow of ABSC between the device and the DAA issuer

certified for the user. Then the user with a root key $(SK' = x', PK' = g_2^{x'})$, replies with $\sigma'_i = \mathsf{H}_1(i||h'_m||\pi'(i))^x$, where $h'_m = \mathsf{H}_0(M')$ for all $i \in \mathcal{S}'$, where $\mathcal{S}'$ represents the set of indices of the attribute in $\mathcal{S}'$. The Privacy CA verifies each σ'_i under PK'. The CA picks $r' \leftarrow_\$ \mathbb{Z}_p$, $\vec{V}' \leftarrow_\$ \mathbb{Z}_p^{n'_2-1}$. Compute $h'_m = \mathsf{H}_0(M')$ and the attribute keys $\mathsf{sk}'_1 := g_2^{r'}$, $\forall\ i \in \mathcal{S}'$ $\mathsf{sk}'_{2,i} := g_1^{\vec{M}'_i(\alpha||\vec{V}')} \cdot \sigma'^{r'}_i$. The CA outputs $\mathsf{sk}' := (\mathsf{sk}'_1, \mathsf{sk}'_{2,i})\ \forall\ i \in \mathcal{S}'$.

- $\mathsf{SignCrypt}(\mathsf{sk}, (\mathcal{S},\mathcal{S}') \subseteq \mathcal{U}, (M,\pi), (M',\pi'), \mathsf{msg}) \rightarrow \sigma$: This algorithm is run by the encryptor/signer. Note that the signature and the encryption parts are glued together using the same parameter t to prove that the signer is the encryptor to avoid man-in-the-middle attacks.
  1. Pick $s, t, r_\alpha, (r_i)_{i\in[n_1]} \leftarrow_\$ \mathbb{Z}_p$;
  2. Compute $h_m = \mathsf{H}_0(M)$ and $h'_m = \mathsf{H}_0(M')$;
  3. $\sigma_1 := \mathsf{sk}_1^{xt}$;
  4. $\sigma_2 := \prod_{i\in\mathcal{S}}(\mathsf{sk}_{2,i})^{\gamma_i s}$;
  5. $\sigma_3 := \prod_{i\in\mathcal{S}} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s} = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s}$ since we choose $\forall\ i \in [n_1]\backslash\mathcal{S}\ \gamma_i = 0$;
  6. $\sigma_4 := g_2^t$;
  7. $\forall_{i\in\mathcal{S}'}\ \sigma_{5,i} = \mathsf{H}_1(i||h'_m||\pi'(i))^t$;
  8. $\sigma_6 = X^t \cdot \mathsf{msg}$;
  9. $Y = X^{s\cdot t}$;
  10. $Z = X^{r_\alpha}$;
  11. $W = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{r_i}$;
  12. $c = \mathsf{H}_0(\sigma_1, \sigma_2, \sigma_3, \sigma_4, (\sigma_{5,i})_{i\in\mathcal{S}'}, \sigma_6, Y, Z, W)$;
  13. $s_\alpha = r_\alpha - s \cdot t \cdot c$;
  14. $\forall_{i\in\mathcal{S}},\ s_i = r_i - \gamma_i \cdot s \cdot c, \forall_{i\in[n_1]\backslash\mathcal{S}}\ s_i = r_i$;
  15. $\sigma := (\sigma_1, \sigma_2, \sigma_3, \sigma_4, (\sigma_{5,i})_{i\in\mathcal{S}'}, \sigma_6, c, s_\alpha, (s_i)_{i\in[n_1]})$
- $\mathsf{VerifyDec}(\mathsf{mpk}, \mathcal{S}', (M,\pi), (M',\pi'), \sigma) \rightarrow 0/1$: This algorithm is run by the decryptor/verifier. It takes a signcrypt and returns a verification result of 0 or 1.
  1. Parse σ to $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, (\sigma_{5,i})_{i\in\mathcal{I}'}, \sigma_6, c, s_\alpha, (s_i)_{i\in[n_1]})$;

2. Compute $h_m = \mathsf{H}_0(M)$ and $h'_m = \mathsf{H}_0(M')$;
3. Compute $Y' = e(\sigma_2, \sigma_4)/e(\sigma_3, \sigma_1)$;
4. Compute $Z' = X^{s_\alpha} \cdot Y^c$ and $W' = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{s_i} \cdot \sigma_3^c$;
5. Compute $c' = \mathsf{H}_0(\sigma_1, \sigma_2, \sigma_3, \sigma_4, (\sigma_{5,i})_{i\in\mathcal{I}'}, \sigma_6, Y', Z', W')$;
6. If $c' \neq c$, output 0; otherwise, compute msg $= \dfrac{\sigma_6 \cdot e\left(\prod_{i\in\mathcal{S}'} \sigma_{5,i}^{\gamma'_i}, \mathsf{sk}_1'^{x'}\right)}{e\left(\prod_{i\in\mathcal{S}'} \mathsf{sk}_{2,i}'^{\gamma'_i}, \sigma_4\right)}$

6.3.1 REWIRE ZTO-II: Proof-of-Ownership of Attribute Space for Domain Enrolment

In the context of the REWIRE framework, we have adopted a Zero Touch Onboarding (ZTO) scheme designed to align with ETSI standards and to support varying privacy requirements across different operational domains. To address these varying requirements, the framework defines two distinct flavors of the ZTO protocol. Although both flavors are grounded in the same cryptographic and security primitives, they diverge in how attributes are disclosed by the device during the credential presentation phase.

The first flavor leverages signcryption and enables the device to selectively disclose its attributes. This approach is particularly suited to domains with heightened privacy constraints, allowing the device to reveal only those attributes that are strictly necessary for authentication or authorization within a given context. The second flavor, by contrast, requires the device to provide a proof of knowledge that discloses the entirety of its attributes. This full disclosure approach may be more appropriate in domains with lower sensitivity to privacy concerns or where comprehensive attribute verification is required for compliance or operational integrity.

A representative example of such a domain can be found in smart city infrastructures, particularly in the deployment of vehicular networks and smart satellites monitoring urban intersections. In these environments, devices such as roadside units (RSUs), autonomous vehicles, and communication relays often need to authenticate with control infrastructure to establish trust and enable coordination. However, the primary concern in this context is typically the accuracy, timing, and consistency of the data rather than the protection of device-specific attributes. As such, it may be acceptable—or even preferable—for devices to disclose all their relevant attributes (e.g., device type, manufacturer, domain authority, capabilities) in a verifiable manner, as this facilitates interoperability and efficient trust decisions across multiple stakeholders and jurisdictions.

Despite these functional differences, both variants of the ZTO scheme share the same underlying structure and cryptographic foundation, which is based on BBS signatures. The key distinction at the construction level lies in the verifiable credential (VC) issuance phase. In signcryption-based flavor, we use H_1 to map $i||h_m||\pi(i)$ to a group generator in $\mathbb{G}_1$, in order to create the the public attribute keys and the public domain key. This co-generation facilitates the selective disclosure mechanism, as it enables more flexible and privacy-preserving proofs without altering the trust or integrity assumptions of the scheme.

The protocol proceeds through a common set of phases that include verifiable credential issuance, domain enrollment, signing, and verification. As the differences between the two ZTO flavors are largely confined to the key generation process during credential issuance, the remaining phases of the protocol retain a unified structure. Therefore, in the remainder of this chapter, we focus on describing the signing and verification processes, which are shared by both flavors and demonstrate the core cryptographic functionality of the onboarding scheme.

Zero-Knowledge Proof of Knowledge over the Entire Attribute Space

The following construction implements a zero-knowledge proof of knowledge (ZKPoK) over the full attribute space of a device in the context of a Zero Touch Onboarding (ZTO) scheme. The protocol is designed to demonstrate possession of a valid verifiable credential and all corresponding attributes without revealing sensitive details beyond what is necessary for verification. It builds upon bilinear group-based cryptographic primitives, including structure-preserving BBS signatures, and integrates domain-bound commitments to prevent cross-domain replay or misuse.

Attribute-Based Key Generation

Let $\mathbb{G}_1$ be a cyclic group of prime order p, and let $\mathcal{A} = \{\text{attr}_1, \text{attr}_2, \ldots, \text{attr}_n\}$ be the set of attributes and $g \in \mathbb{G}_1$.
For every attribute $\text{attr}_i \in \mathcal{A}$, we:

- Sample a private attribute key $a_i \in \mathbb{Z}_p$
- Compute the public attribute key as $pk_i = h_1^{a_i} \in \mathbb{G}_1$.

The process begins with the prover sampling random values $\hat{a_1}, \hat{a_2}, \hat{b}, k \in \mathbb{Z}_p$, which serve as ephemeral blinding factors for the proof. These randomnesses are used to compute key intermediate commitments, beginning with $\bar{Y}$, which binds the domain identifier ID_{DOMAIN} and the full set of attributes $\{a_i\}$ into a single exponentiated group element under h_1. This construction ensures that any proof is domain-specific, enforcing contextual integrity.

The computation $\bar{A}_{DAA}^{\hat{a_1}\hat{a_2}}$ reflects a transformed version of the BBS DAA credential under blinding exponents, maintaining unlinkability and privacy. Similarly, the values $R = h_1^k$ and $\bar{R} = (Rh_1^{\hat{b}})^{\hat{a_2}}$ introduce randomized commitment components to support proof freshness and non-malleability.

DAA VC

Let $\mathbb{G}_1$ be a cyclic group of prime order p, then:

- $h_0 \in \mathbb{G}_1$
- $h_1 \in \mathbb{G}_1$

$cre_{DAA} = (A_{DAA}, e_{DAA})$, where:

- $A_{DAA} = (g_0 h_0^{sk_{DAA}} h_1^{H(ID_{DOMAIN})} \prod_{i\in[n]} h_1^{a_i})^{1/(sk_I + e_{DAA})}$
- $e_{DAA} \in \mathbb{Z}_p$

The term $D = (g_0 h_0^{sk_{DAA}})^{\hat{a_2}}$ encapsulates the secret key of the DAA signer, re-randomized under $\hat{a_2}$, while $B = D^{\hat{a_1}} \bar{Y}^{\hat{a_1}} \bar{A}_{DAA}^{-e_{DAA}}$ serves as the key component of the proof, combining credential, attributes, and domain context into a bound structure.

To prepare for the Fiat-Shamir heuristic, the protocol defines auxiliary values, including $\hat{a_2}' = \frac{1}{\hat{a_2}}$, and computes a message hash challenge value c. The response scalar γ incorporates the sum of attributes and domain hash, combined with the randomness k and $\hat{b}$, and weighted by the challenge hash c, ensuring non-interactive zero-knowledge compliance.

Next, a second set of randomizers $(r_\gamma, r_{a_1}, r_{a_2}, r_{\hat{b}}, r_{e_{DAA}}, r_{sk_{DAA}})$ is sampled to construct the corresponding proof commitments T_1 through T_4, each designed to hide the actual witness values while preserving algebraic consistency.

Finally, the challenge value ch is derived as a hash over the complete transcript and public parameters, and the proof responses $\hat{s}_1$ through $\hat{s}_6$ are computed by combining the randomizers and corresponding witness values scaled by the challenge. This structure guarantees soundness under the knowledge assumption, while preserving zero-knowledge through randomization.

This construction serves as a foundational building block for domains that require full attribute disclo-

sure during onboarding, providing a cryptographically secure and privacy-preserving proof of credential possession and attribute integrity.

1. Sample random $\hat{a_1}, \hat{a_2}, \hat{b}, k \in \mathbb{Z}_p$
2. $\bar{Y} = (h_1^{H(ID_{DOMAIN})} \prod_{i\in[n]} pk_i)^{\hat{a_2}}$
3. $\bar{A_{DAA}} = A_{DAA}^{\hat{a_1}\hat{a_2}}$
4. $R = h_1^k$
5. $\bar{R} = (Rh_1^{\hat{b}})^{\hat{a_2}}$
6. $D = (g_0 h_0^{sk_{DAA}})^{\hat{a_2}}$
7. $B = D^{\hat{a_1}} \bar{Y}^{\hat{a_1}} \bar{A_{DAA}}^{-e_{DAA}}$
8. $\hat{a_2'} = \frac{1}{\hat{a_2}}$ and $\gamma = \hat{a_2}((\sum_{i\in[n]} a_i + H(ID_{DOMAIN}))H(R||PK_I||h_1^{H(ID_{DOMAIN})} \prod_{i\in[n]} pk_i||m) + k + \hat{b})$
9. $c = H(R||PK_I||h_1^{H(ID_{DOMAIN})} \prod_{i\in[n]} pk_i||m)$
10. Sample random $r_\gamma, r_{a_1}, r_{a_2}, r_{\hat{b}}, r_{e_{DAA}}, r_{sk_{DAA}} \in \mathbb{Z}_p$
11. $T_1 = \bar{R}^{r_{a_2}} h_1^{-r_{\hat{b}}}$
12. $T_2 = h_1^{r_\gamma}$
13. $T_3 = D^{r_{a_1}} \bar{Y}^{r_{a_1}} \bar{A_{DAA}}^{-r_{e_{DAA}}}$
14. $T_4 = D^{r_{a_2}} h_0^{-r_{sk_{DAA}}}$
15. $ch = H(R||\bar{R}||\bar{Y}||D||B||\bar{A}||T_1||T_2||T_3||T_4||m)$
16. $\hat{s_1} = r_\gamma + ch * \gamma$
17. $\hat{s_2} = r_{a_1} + ch * \hat{a_1}$
18. $\hat{s_3} = r_{a_2} + ch * \hat{a_2'}$
19. $\hat{s_4} = r_{\hat{b}} + ch * \hat{b}$
20. $\hat{s_5} = r_{e_{DAA}} + ch * e_{DAA}$
21. $\hat{s_6} = r_{sk_{DAA}} + ch * sk_{DAA}$

Figure 6.6: Zero Knowledge Proof of Knowledge of the entire Attribute Space.

The verification procedure described in Figure 6.7 validates a non-interactive zero-knowledge proof of knowledge (ZKPoK) over a device's entire attribute space. This process ensures the correctness and integrity of the commitments and responses generated during the proof phase, and confirms that the prover indeed possesses valid credentials and corresponding attributes bound to the declared domain.

The procedure begins with a pairing equation check:

$$\hat{h}(\bar{A_{DAA}}, PK_I) \stackrel{?}{=} \hat{h}(B, g_2)$$

This bilinear equality ensures that the BBS+ signature $\bar{A_{DAA}}$ is valid with respect to the issuer's public key PK_I and the constructed element B, which binds the domain authority, the attribute commitment, and the credential structure. The equality check here is essential to prevent forgery and ensures that the credential is indeed issued by the legitimate authority.

Next, the verifier recomputes the original challenge c_v as:

$$c_v = H(R||PK_I||h_1^{H(ID_{DOMAIN})} \prod_{i\in[n]} pk_i||m)$$

This binds the credential proof to a specific message m, the issuer's public key, and the domain context. The verifier then reconstructs the proof commitments from the prover's responses using the following expressions:

$$\hat{T_1} = R^{-ch} \bar{R}^{\hat{s_3}} h_1^{-\hat{s_4}}$$

$$\hat{T2} = (\bar{R}\bar{Y}^{c_v})^{-ch} h_1^{\hat{s_1}}$$

$$\hat{T_3} = B^{-ch} (D\bar{Y})^{\hat{s_2}} \bar{A_{DAA}}^{-\hat{s_5}}$$

$$\hat{T_4} = g_0^{-ch} D^{\hat{s_3}} h_0^{-\hat{s_6}}$$

Each of these reconstructed terms serves as a consistency check for a particular component of the original proof. The algebraic relationships ensure that the prover's claimed knowledge of secret values such as the private key, randomized scalars, and full attribute set are internally coherent and match the commitments.

Finally, the verifier computes a new challenge hash ch_v using the reconstructed proof elements:

$$ch_v = H(R||\bar{R}||\bar{Y}||D||B||\bar{A_{DAA}}||\hat{T_1}||\hat{T_2}||\hat{T_3}||\hat{T_4}||m)$$

The proof is accepted only if this computed challenge matches the one sent by the prover:

$$ch \stackrel{?}{=} ch_v$$

This final equality guarantees that the non-interactive proof has not been tampered with and that all elements correspond to a consistent witness, thereby confirming the prover's possession of valid credentials for the complete attribute space.

1. $\hat{h}(\bar{A_{DAA}}, PK_I) \stackrel{?}{=} \hat{h}(B, g_2)$
2. $c_v = H(R||PK_I||h_1^{H(ID_{DOMAIN})} \prod_{i\in[n]} pk_i||m)$
3. $\hat{T_1} = R^{-ch} \bar{R}^{\hat{s_3}} h_1^{-\hat{s_4}}$
4. $\hat{T_2} = (\bar{R}\bar{Y}^{c_v})^{-ch} h_1^{\hat{s_1}}$
5. $\hat{T_3} = B^{-ch} (D\bar{Y})^{\hat{s_2}} \bar{A_{DAA}}^{-\hat{s_5}}$
6. $\hat{T_4} = g_0^{-ch} D^{\hat{s_3}} h_0^{-\hat{s_6}}$
7. $ch_v = H(R||\bar{R}||\bar{Y}||D||B||\bar{A_{DAA}}||\hat{T_1}||\hat{T_2}||\hat{T_3}||\hat{T_4}||m)$
8. $ch \stackrel{?}{=} ch_v$

Figure 6.7: DAA Verification of The Entire Attribute Space.

6.4 Security Analysis of REWIRE Crypto Agility Layer

In this section, we firstly give the security requirements for an ABSC scheme and then give security analysis of the proposed ABSC scheme and the ZTO process.

6.4.1 Security requirements for an ABSC scheme

Because an ABSC scheme is an extension of an ABS, when talking about the security requirements of ABSC, we need to give the security requirements of ABS.

Firstly, for an ABS scheme, the goal of the adversary can be one of the following:

1. To forge a signature with a predicate that his attributes do not satisfy. We proved that the adversary's attributes $\mathcal{S}$ do not satisfy the policy. Therefore, the signer cannot generate a valid ABS that satisfies the policy matrix M. This is achieved in our ABS scheme because the keys that correspond to the attributes in $\mathcal{S}$ are not enough to verify the correctness of the pairing $Y = e(\sigma_2, \sigma_4)/e(\sigma_3, \sigma_1)$. Recall that the attribute keys have the following construction: $\mathsf{sk}_{2,i} := g_1^{M_i(\alpha||\vec{V})^\intercal} \cdot \sigma_i^r$ for each row $i \in \mathcal{I}$. Also, the sign algorithm in the ABS scheme shows that $\sigma_2 := \prod_{i\in\mathcal{I}} (\mathsf{sk}_{2,i})^{\gamma_i s}$ is a part of the ABS signature. If the attribute keys $\mathsf{sk}_{2,i}$ corresponding to the attributes in $\mathcal{S}$ are not enough to satisfy M as in our case, then the simplification of the $\prod_{i\in\mathcal{I}} g_1^{M_i(\alpha||\vec{V})^\intercal \gamma_i}$ is not possible because the equation $\sum \gamma_i M_i = (1, 0, ..., 0)$ cannot be valid. Hence, the ABS signature will not be successfully verified. A corrupt signer might get the attribute key corresponding to the `Secure Boot` from another signer so to have enough keys to build an ABS valid signature, but this would not work in our scheme since the attribute keys are bound to the root-key of the device and generated from different seeds r for different users. This prevents any collusion between signers or impersonating attacks. Similar unforgeability arguments follow for our ABSC schemes.

2. To distinguish which attributes were used to generate a signature or any other identifying information associated with the particular signer amongst the users satisfying the given predicate. This might lead to the *device fingerprinting* which in turn can disclose information to the adversary of the internal device's characteristics (e.g., Operating System version, whitlist of binaries loaded as part of the operational stack) that can further lead to implementation disclosure attacks. The security requirements can be summarized as unforgeability and attribute-signer privacy. For the ABSC, in addition to the above two targets, the ciphertext does not reveal any information about the encrypted message, which we call "Indistinguishability against Chosen Plaintext Attack (IND-CPA)" security.
3. To generate a valid signature via accessing some attribute key set but without knowing the identity key.

Based on the above descriptions, in the following, we give definitions of all security requirements, *attribute privacy*, *unforgeability*, *IND-CPA* and *attribute-identity key binding* in ABS/ABSC.

Definition 1 (**Attribute Privacy in ABS**)**.** Given two valid attribute-based signatures σ_1 and σ_2 generated from two different attribute sets $\mathbb{A}_1$ and $\mathbb{A}_2$ (associated with two attribute key sets SK_1 and SK_2 generated and certified by a trusted authority) and satisfying the same policy $\mathcal{P}$. Choose a random signature σ_i from $\{\sigma_1, \sigma_2\}$, the probability that an adversary knows which set of attributes in $\{\mathcal{S}_1, \mathcal{S}_2\}$ used to generate σ_i is $1/2$.

The privacy is achieved in our schemes by the construction of σ_3 and the Shonrr signatures as follows: $\sigma_3 := \prod_{i\in\mathcal{I}} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s} = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s}$ since we choose $\forall\ i \in [n_1]/\mathcal{I}\ \gamma_i = 0$. Note that a Shnorr signature is created on all the attributes in the matrix policy M that represents the policy for the Universe attribute set $\mathcal{U}$, this not only helps in verifying the construction of σ_3 and hence σ but also hides the attributes used by the signer to satisfy $\mathcal{P}$ into a larger set $\mathcal{U}$.

Definition 2 (**Unforgeability in ABS**)**.** We consider that the ABS scheme is existentially unforgeable against chosen predicate and message attacks. Its formal definition is based on the following game involving a challenger $\mathcal{B}$ and an adversary $\mathcal{A}$:

- Setup Phase: $\mathcal{B}$ runs Setup. $\mathcal{B}$ retains Privacy CA secret and public keys generated from Setup, sends the public key to $\mathcal{A}$;
- Query Phase: $\mathcal{A}$ can perform a polynomially bounded number of queries on $(\mathsf{msg}, \mathcal{S})$ to private key extraction oracle and signing oracle, respectively. These queries will be answered by $\mathcal{B}$ with a secret key.
- Forgery: Finally, $\mathcal{A}$ outputs an ABS signature σ^* on messages msg^* using an an attribute set ω^* that satisfies a policy $\mathcal{S}^*$.

We say that the adversary wins the game if σ^* is a valid signature on message msg^* for a policy $\mathcal{S}^*$, such that $(\mathsf{msg}^*, \mathcal{S}^*)$ has not been queried to the signing oracle and no attribute set ω such that there exists $\omega \subset \omega^*$ satisfying $\mathcal{S}^*$ has been submitted to the key generation oracle. The advantage of $\mathcal{A}$ is defined as the probability that it wins the game and should be negligible.

Definition 3 (**IND-CPA**)**.** a ciphertext does not reveal any information about the encrypted message, which we call "Indistinguishability against Chosen Plaintext Attack (IND-CPA)" security. We model the adaptive IND-CPA security in a game running between an adversary $\mathcal{A}$ and a challenger $\mathcal{B}$ as follows:

- Setup Phase: $\mathcal{B}$ runs Setup to obtain the Privacy CA public and secret keys. It sends the CA's public key to the adversary and keeps the secret part.
- Phase 1. $\mathcal{A}$ issues queries to a key generation oracle for a Key generation oracle that issues a set of attribute keys for an attribute set $\mathcal{S}$ under a certain access matrix $\mathbf{M}$ and sends the keys to $\mathcal{A}$.
- Challenge. $\mathcal{A}$ outputs a challenging attribute set $\mathcal{S}^*$ with the restriction that $\mathcal{S}^*$ cannot satisfy any $\mathcal{S}$ that has been queried in Phase 1 and two equal-length messages msg_0^* and msg_1^*. Then $\mathcal{B}$ selects a random bit $b \in \{0, 1\}$, runs the encryption algorithm to get the ciphertext CT_b^* on a message msg_b^* under $\mathcal{S}^*$ and returns the challenge CT_b^* to $\mathcal{A}$.

- Guess. $\mathcal{A}$ outputs $b' \in \{0, 1\}$ and wins the game if $b = b'$.

An ABSC scheme is adaptively IND-CPA secure if the advantage function defined by the probability $|b' = b| - 1/2$ is negligible.

For the key binding property, we need to discuss two cases, i.e., attribute-identity key binding and attestation-identity key binding.

Definition 4 (**Attribute-identity key binding**)**.** This property requires the Privacy CA to be honest. An ABS protocol holds the key binding property if every valid ABS generated using a set of attribute keys sk_{PK} and the device's secrete identity key x, which corresponds to the device public key PK, then $(sk_{PK} : (sk_1, sk_2), h_m, PK)$ is found in the CA's record, i.e., the attribute keys in sk_{PK} and PK should belong to the same device.

The key binding experiment $Exp_{ABSC}^{akeybind}$ allows the adversary $\mathcal{A}$ to corrupt some attribute keys. In this game, the adversary is given access to a corrupt device's attribute keys sk_{PK}^* (resp. $sk_{PK}'^*$) and generates an ABSC on behalf of the device using the attribute keys that can represent signing keys(resp. decryption keys) in sk_{PK}^* (resp. $sk_{PK}'^*$) without the knowledge of the secret identity key x^* (resp. x'^*) that corresponds to PK (resp. PK'). At the end of the execution. The outcome of the experiment is determined as follows. If the key (α, X) or $(x, PK)/(x', PK')$ has been corrupted, the experiment returns 0. Otherwise, if σ^* is valid, the experiment returns 1, else returns 0.

We define the advantage of the adversary $\mathcal{A}$ in the attribute-identity key binding game as $Adv_{ABSC}^{akeybind}(\nu) = Pr[Exp_{ABSC}^{akeybind}(\nu) = 1]$. Then, we say that the ABSC (ABS) protocol holds attribute-identity key binding if $Adv_{ABSC}^{akeybind}(\nu)$ is a negligible function of ν for all polynomial-time adversaries $\mathcal{A}$.

Except the **attribute-identity key binding**, we also need to give the definition of **attestation-identity key binding**. In the definition of **attestation-identity key binding**, the underline requirement is the binding of credential to the identity key. Therefore, here, we also need to prove the credential is unforgeable.

Definition 5 (**Credential unforgeability**)**.** The credential unforgeability is defined based the following game involving a challenger $\mathcal{B}$ and an adversary $\mathcal{A}$:

- Setup phase: $\mathcal{B}$ runs Setup. Then $\mathcal{B}$ retains the secret key and sends the public key to $\mathcal{A}$;
- Query Phase: $\mathcal{A}$ can perform a polynomial bounded number of queries on some attestation keys to the secret key extraction oracle and signing oracle, respectively. These queries will be answered by $\mathcal{B}$ with a secret key or a credential to the adversary.
- Forgery: the adversary $\mathcal{A}$ outputs a credential $cred^*$ on a device's public key PK_{VC}^*.

We say the adversary wins the game if $cred^*$ is a valid credential on PK_{VC}^* and $(cred^*, PK_{VC}^*)$ has not been queried to the signing oracle before. The advantage of $\mathcal{A}$ is defined as the probability that it wins the game. If the credential is unforgeable, the advantage of $\mathcal{A}$ should be negligible.

Definition 6. The Key binding Experiment: This property requires the device with an identity key sk_R to be honest. This property guarantees that if the device with an identity key sk_R holds a valid credential $cred$ on an Attestation key sk_{VC}, then sk_{VC} should be bound to the device's unique identity key sk_R.

The key binding experiment $Exp_{Att}^{keybind}$ allows the adversary $\mathcal{A}$ to corrupt some attestation keys. In this game, the adversary is given access to corrupt devices' attestation keys sk_{VC}^* and generates a self-certification $cred$ on behalf of an honest device using the attestation key sk_{VC}^* without the knowledge of the secret identity key sk_R of the honest device.

At the end of the execution, the outcome of the experiment is determined as follows: if the key (sk_{VC}, PK_{VC}) has been corrupted, the experiment returns 0. Otherwise, the experiment returns 1.

Definition 7. Attestation-identity Key binding: We define the advantage of the adversary $\mathcal{A}$ in the key binding game as: $Adv_{Att}^{keybind}(\nu) = Pr[Exp_{Att}^{keybind}(\nu) = 1]$ and say that the protocol holds key binding if $Adv_{Att}^{keybind}(\nu)$ is a negligible function of ν for all polynomial-time adversaries $\mathcal{A}$.

In ZTO process, there is an extra property that can be satisfied, called **selective disclosure of attributes**. It means the device can reveal a subset of attributes while keeping the remaining attributes hidden. This

property can be achieved in the ABSC scheme, which means the device can sign/encrypt some attributes to be hidden while leave remaining attributes open. It is trivial to realize this property.

6.4.2 Security Analysis of Attribute-based SignCryption

In this section, we will give the security analysis of the proposed ABS/ABSC scheme based on the above security requirements described in section 6.4.1.

Hardness assumptions. *Bilinear Diffie-Hellman Inversion (BDHI) assumption.* We use a simplified version of the BDHI problem introduced in [13] with a Type-III setting.

We define par $:= (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$, $\alpha, \beta \xleftarrow{\$} \mathbb{Z}_p$, $\mathsf{D} = (g_1^\alpha, g_2^\alpha, g_1^\beta, g_2^\beta)$. The target is to compute $\mathsf{T} = e(g_1, g_2)^{\alpha^2 \cdot \beta}$. We denote the advantage of an algorithm $\mathcal{A}$ in solving the computational BDHI problem by

$$\mathrm{Adv}_{\mathcal{A}}^{\mathrm{BDHI}}(\lambda) := \left| \Pr\left[\mathcal{A}(\mathsf{par}, \mathsf{D}, \mathsf{T}) = 1\right] \right|$$

Which should be negligible in λ. We say that an algorithm $\mathcal{A}(t, \epsilon)$ solves the BDHI problem in $\mathbb{G}_T$ if $\mathcal{A}$ runs in time at most t, $\mathrm{Adv}_{\mathcal{A}}^{\mathrm{BDHI}}$ is at least ϵ.

Theorem 1. *Our ABS scheme satisfies unforgeability in the random oracle model under the BDHI assumption.*

Proof. Suppose a PPT adversary $\mathcal{A}$ can break our ABS scheme in the selective unforgeability game. There exists an algorithm $\mathcal{B}$ that solves the BDHI problem. The simulator $\mathcal{B}$ is given a BDHI tuple $(\hat{g_1}, \hat{g_2}, \hat{g_1}^\alpha, \hat{g_2}^\alpha, \hat{g_1}^\beta, \hat{g_2}^\beta)$. $\mathcal{B}$' s objective is to compute $e(\hat{g_1}, \hat{g_2})^{\alpha^2\beta}$, it works by interacting with $\mathcal{A}$ as in the following game settings:

Setup. Let $l^* + 1$ denote the number of challenged attributes in $\mathcal{S}^*$, including the identity attribute on which the adversary aims to create a forged ABS. Let $g_1 = \hat{g}_1^\alpha$ and $g_2 = \hat{g}_2$. $\mathcal{B}$ sets the master public key $X \leftarrow e(g_1, g_2)^\alpha = e(\hat{g_1}^\alpha, \hat{g_2})^\alpha = e(\hat{g_1}, \hat{g_2})^{\alpha^2}$. The master secret key is implicitly set as α, and g_1^α is implicitly set as $\hat{g_1}^{\alpha^2}$.

$\mathcal{B}$ samples a uniformly random vector $\vec{V} \in \mathbb{Z}_p^{n_2-1}$. If the attribute corresponding to the row $\vec{M}_i$ belongs to the challenge set $\mathcal{S}^*/id$, the simulator calculates $\xi_i = \vec{M}_i.(1|\vec{V}) \in \mathbb{Z}_p$. Note that if the attributes corresponding to the rows $\vec{M}_i$ satisfy the policy $\mathbf{M}$, then $\sum \xi_i = 1$. Assume that the attributes in $\mathcal{S}^*/id$ satisfies $\mathbf{M}$. The adversary aims to forge an ABS signature on $\mathcal{S}^*$ without having the attribute key corresponding to the identity (not necessarily identity can be any attribute that the adversary doesn't have its key).

Let f_i be random secret integers randomly sampled by the simulator for each challenged attribute u_i and let $f = \sum_{i \in [1, l^*]} f_i$. The simulator samples a random integer d.

$\mathcal{B}$ sets the random oracle $\mathsf{H}_1 : \{0,1\}^* \to \mathbb{G}_1$ that takes an attribute as input and sets the output as follows:

- If the attribute corresponding to the row $\vec{M}_i$ belongs to the challenge set $\mathcal{S}^*/id$, $\mathcal{B}$ sets:

$$\mathsf{H}_1 : \{0,1\}^* \to \hat{g_1}^{f_i - \xi_i d\alpha}$$

- If the attribute is the identity attribute, then $\mathcal{B}$ sets:

$$\mathsf{H}_1 : \{0,1\}^* \to \hat{g}_1^{d\alpha}$$

When $\mathcal{A}$ issues a query for generating a secret key on a set of attributes $\mathcal{S} \subseteq \mathcal{S}^*/id$. $\mathcal{B}$ runs the random oracle H_1 to obtain the corresponding $h_i \in \mathbb{G}_1$ for each attribute u_i where the random oracle is defined as before. $\mathcal{B}$ defines $h_m = \mathsf{H}_0(M)$ for all $i \in \mathcal{I}$, where $\mathcal{I}$ represents the set of indices of the attributes in $\mathcal{S}$,

we denote $\mathcal{I}$ by $[1, l]$ for some integer $l < l^*$. Let r be a random integer in $\mathbb{Z}_p$ sampled by the simulator. The attribute keys are set as follows:

$$\mathsf{sk}_1 := {g_2}^r$$

$$\mathsf{sk}_{2,i} := \hat{g_1}^{\alpha^2/l^*} \, \hat{g_1}^{(-d\alpha/l^*)r} \hat{g_1}^{r f_i}$$

Then $\mathcal{B}$ implicitly sets $r = \alpha/d \; \forall_{i \in \mathcal{I}}$, then the simulator keys will be:

$$\mathsf{sk}_1 := {g_2}^r = \hat{g_2}^{\frac{\alpha}{d}}$$

$$\begin{aligned} \mathsf{sk}_{2,i} &:= \hat{g_1}^{\alpha^2/l^*} \hat{g_1}^{(-d\alpha/l^*)\frac{\alpha}{d}} \hat{g_1}^{f_i \frac{\alpha}{d}} \\ &= \hat{g_1}^{f_i \frac{\alpha}{d}} \end{aligned}$$

Signing oracle. When $\mathcal{A}$ issues a query for a signature with an access policy $\mathcal{P} = (M \in \mathbb{Z}_p^{n_1 \times n_2}, \; \pi, \; \{\pi(i)\}_{i \in [n_1]})$ and a specific attribute **subset** of $\mathcal{S} = \{u_i\}_{i \in [1,k]}$ on message msg. $\mathcal{B}$ runs the random oracle to obtain the corresponding h_i for each attribute $\pi(i)$. $\mathcal{B}$ proceeds as follows:

- Pick $s, t, r_\alpha, (r_i)_{i \in [n_1]} \leftarrow_\$ \mathbb{Z}_p$
- $\sigma_1 := \mathsf{sk}_1^t = \hat{g_2}^{\frac{\alpha}{d} t}$,
- $\sigma_2 := \prod_{i \in [1,k]} (\mathsf{sk}_{2,i})^{\gamma_i s} = \hat{g_1}^{s \sum_{i \in [1,k]} f_i (\alpha/d)}$,
- $\sigma_3 := \prod_{i \in [n_1]} \mathsf{H}_1(i || h_m || \pi(i))^{\gamma_i s} = \prod_{i \in [1,k]} \hat{g_1}^{s \sum_{i \in [1,k]} (f_i - \xi_i d\alpha)}$.
  If the attributes corresponding to $[1, k]$ rows satisfies the policy $\mathbf{M}$, then $\sigma_3 = \hat{g_1}^{\sum_{i \in [1,k]} s f_i - s d\alpha}$. Note that $\gamma_i = 0$ if $i \notin [1, k]$, otherwise $\gamma_i = 1$.
- $\sigma_4 := g_2^t = \hat{g_2}^t$
- $Y = X^{s \cdot t}$,
- $Z = X^{r_\alpha}$,
- $W = \prod_{i \in [n_1]} \mathsf{H}_1(i || h_m || \pi(i))^{r_i}$,
- $c = \mathsf{H}_0(\sigma_1, \sigma_2, \sigma_3, \sigma_4, Y, Z, W, \mathsf{msg})$,
- $s_\alpha = r_\alpha - s \cdot t \cdot c$, and
- $\forall_{i \in [n_1]} s_i = r_i - \gamma_i \cdot s \cdot c$.
- Output $\sigma := (\sigma_1, \sigma_2, \sigma_3, \sigma_4, c, s_\alpha, (s_i)_{i \in [n_1]})$

The signature verification passes if $\mathcal{S}$ satisfies M. It is easy to see that:

$$e(g_1, g_2)^{\alpha s t} = e(\hat{g_1}, \hat{g_2})^{\alpha^2 s t} = Y' = e(\sigma_2, \sigma_4)/e(\sigma_3, \sigma_1)$$

=

$$\frac{e(\hat{g_1}^{\sum_{i \in [1,k]} f_i s \frac{\alpha}{d}}, \hat{g_2}^t)}{e(\hat{g_1}^{\sum_{i \in [1,k]} s f_i - s d\alpha}, \hat{g_2}^{\frac{\alpha}{d} t})}$$

This indicates that $\mathcal{B}$ can simulate a valid signature σ for $\mathcal{A}$'s query without knowing the answer to the BDHI problem.

Forgery. When $\mathcal{A}$ outputs a valid forgery $\sigma^* = (\sigma_1^*, \sigma_2^*, \sigma_3^*, \sigma_4^*, s_\alpha^*, \{s_i^*\}_{i \in [n_1^*]}, c^*)$ on a message msg^* that satisfies the verification process, there exists $r^*, s^*, t^* \in \mathbb{Z}_p$ such that:

$$\sigma_1^* = \hat{g_2}^{r^* t^*}$$

$$\sigma_2^* = \prod_{i \in [1,l^*]} (\hat{g_1}^{s^* \alpha^2 / l^*} \hat{g_1}^{-r^* s^* d\alpha / l^*} \ \hat{g_1}^{r^* s^* f_i}) . \hat{g_1}^{r^* s^* \alpha d}$$

$$= \hat{g_1}^{\alpha^2 s^*} . \hat{g_1}^{\sum_{i \in [1,l^*]} f_i r^* s^*}$$

$$\sigma_3^* = \prod_{i \in [n_1^*]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i^* s^*}$$

$$= \prod_{i \in [1,l^*]} (\hat{g_1}^{s^* f_i - s^* \xi_i \alpha d}) g^{s^* \alpha d}$$

$$= \hat{g_1}^{\sum_{i \in [1,l^*]} s^* f_i}$$

Note that $\gamma_i^* = 0$ if $i \notin \mathcal{S}^*$ and 1 otherwise.

$$\sigma_4^* = \hat{g_2}^{t^*}$$

It proceeds by creating Shnorr signatures, as in the ABS protocol. It sets $s_\alpha^* = r_\alpha^* - s^* \cdot t^* \cdot c^*, \forall i \in \mathcal{I}^* : s_i^* = r_i^* - \gamma_i^* \cdot s^* \cdot c^*, \forall i \notin \mathcal{I}^* : s_i^* = r_i^*$ and out puts σ^*.

The signature verification passes if $\mathcal{S}^*$ satisfies $\mathbf{M}$. It is easy to see that:

$$e(g_1, g_2)^{\alpha s^* t^*} = e(\hat{g_1}, \hat{g_2})^{\alpha^2 s^* t^*} = Y'^* = e(\sigma_2^*, \sigma_4^*) / e(\sigma_3^*, \sigma_1^*)$$

=

$$\frac{e(\hat{g_1}^{\alpha^2 s^*} . \hat{g_1}^{\sum_{i \in [1,l^*]} f_i r^* s^*}, \hat{g_2}^{t^*})}{e(\hat{g_1}^{\sum_{i \in [1,l^*]} s^* f_i}, \hat{g_2}^{r^* t^*})}$$

Then $\mathcal{B}$ uses the forking lemma [10] to extract the value $s^* t^*$ hidden in s_α^*, and the value $\{\gamma_i^* s\}_{i \in \mathcal{I}}$ hidden in s_i^*: After a polynomial reply of $\mathcal{A}$, $\mathcal{B}$ obtains two valid signatures $(s_{\alpha_1}^*, s_{i1}^*, c_1^*)$ and $(s_{\alpha_2}^*, s_{i2}^*, c_2^*)$, where $c_1^* \neq c_2^*$. According to the scheme, $s_{\alpha_1}^* = r_\alpha^* - s^* t^* \cdot c_1^*$, $s_{\alpha_2}^* = r_\alpha^* - s^* t^* \cdot c_2^*$, then we can get $s_{\alpha_1}^* - s_{\alpha_2}^* = s^* t^* \cdot (c_2^* - c_1^*)$. Therefore, $s^* t^* = \frac{s_{\alpha_1}^* - s_{\alpha_2}^*}{c_2^* - c_1^*}$. Similarly, we can obtain $s_{i1}^* - s_{i2}^* = \gamma_i^* s^* \cdot (c_2^* - c_1^*)$, so $\gamma_i^* s^* = \frac{s_{i1}^* - s_{i2}^*}{c_2^* - c_1^*}$. By leveraging the knowledge of $s^* t^*$, $\{\gamma_i^* s^*\}_{i \in \mathcal{I}^*}$ and $\{\gamma_i^*\}_{i \in \mathcal{I}^*}$, $\mathcal{B}$ can first calculate the value of s^* and t^* separately, and then compute $\hat{g_2}^{r^*} = \sigma_1^{\frac{1}{t^*}}$. Finally, $\mathcal{B}$ can extract the answer to the BDHI problem from the forgery by the following:

$$\mathsf{T} = \frac{e(\sigma_2{}^{\frac{1}{s^*}}, \hat{g_2}^{\beta})}{e(\hat{g_1}^{\beta}, \hat{g_2}^{r^*})^{\sum_{i \in [1,l^*]} f_i}}$$

□

Theorem 2. *Our A-KP-ABE scheme used in the ABSC is adaptively IND-CPA secure under the generic group model by modeling the hash function* H_1 *as a random oracle.*

Proof. In the IND-CPA game, the only ciphertext component that is related to the two challenged messages is $\sigma_6 = e(g_1, g_2)^{\alpha t}.msg$. Therefore, the adversary $\mathcal{A}$ attempts to win the game by distinguishing $\sigma_6 = e(g_1, g_2)^{\alpha t}.msg_0^*$ from $\sigma_6 = e(g_1, g_2)^{\alpha t}.msg_1^*$.

For $\tau \leftarrow \mathbb{Z}_p$, the probability of distinguishing $e(g_1, g_2)^{\alpha t}.msg_0^*$ from $e(g_1, g_2)^{\tau}$is equal to that of distinguishing $e(g_1, g_2)^{\tau}$ from $e(g_1, g_2)^{\alpha t}.msg_1^*$. Therefore, if $\mathcal{A}$ has advantage ϵ in winning the IND-CPA game, then it has advantage $\epsilon/2$ in distinguishing $e(g_1, g_2)^{\alpha t}$ from $e(g_1, g_2)^{\tau}$. Thus, we consider the game where$\mathcal{A}$ can distinguish $e(g_1, g_2)^{\alpha t}$ from $e(g_1, g_2)^{\tau}$. The modified game is simulated as follows:

Setup. The challenger $\mathcal{B}$ chooses $\alpha \leftarrow \mathbb{Z}_p$ and sends the public key $pk; (g_1, g_2, X = e(g_1, g_2)^{\alpha})$ to $\mathcal{A}$.

Phase 1. In phase 1, $\mathcal{A}$ can make oracle queries to the random oracle and a key generation oracle as follows:

Random Oracle: When $\mathcal{A}$ makes a query, $\mathcal{B}$ outputs

$$\mathsf{H}_1 : \{0,1\}^* \rightarrow \hat{g_1}^{d_i}$$

Key generation oracle: When $\mathcal{A}$ makes a key query for an access policy, $\mathcal{B}$ picks $r \leftarrow \mathbb{Z}_p$ and a vector $\vec{V} \leftarrow \mathbb{Z}^{n_2-1}$. Let $m_i = M(\alpha||\vec{V})$ Note that the m_i is random since it depends on the random distribution of α and $\vec{V}$. Then $\mathcal{B}$ generates the secret key as the following: $sk_1 = g_2^r$, $\forall\ i \in \mathcal{I}^*$ $\mathsf{sk}_{2,i} := g_1^{m_i + d_i r}$ and sends them to $\mathcal{A}$.

Challenge:$\mathcal{A}$ outputs an attribute set $\mathcal{S}^*$ and two messages msg_1^*, msg_1^* that it intends to attack. $\mathcal{B}$ checks if $\mathcal{S}^*$ satisfies any access policy queried in Phase 1. If yes, $\mathcal{B}$ aborts. Otherwise, $\mathcal{B}$ chooses $t, \tau \leftarrow \mathbb{Z}_p$. $\mathcal{B}$ chooses $\mu \leftarrow \{0,1\}$. If $\mu = 0$, it generates the challenge ciphertext as follows: If $\mu = 0$, then $\sigma_6 : e(g_1, g_2)^{\alpha t}$, $\sigma_4 : g_2^t$, $\forall_{i \in \mathcal{I}'}\ \sigma_{5,i} = g^{d_i}$. Otherwise, $\sigma_6 : e(g_1, g_2)^{\tau}$, $\sigma_4 : g_2^t$, $\forall_{i \in \mathcal{I}'}\ \sigma_{5,i} = g_1^{d_i t}$. $\mathcal{B}$ sends the ciphertext to $\mathcal{A}$.

Phase 2. It is the same as in Phase 1, with the restriction the input access policy $\mathbf{M}$ is not satisfied by the challenge attribute set $\mathcal{S}^*$.

If $\mathcal{A}$ can construct $e(g_1, g_2)^{\delta \alpha t}$ for some $\delta \in \mathbb{Z}_p$ that can be combined from the oracle outputs already queried, then $\mathcal{A}$ can use it to distinguish $e(g_1, g_2)^{\alpha t}$ from $e(g_1, g_2)^{\tau}$. $\mathcal{A}$ can construct $e(g_1, g_2)^{\delta \alpha t}$ with negligible probability since $\mathcal{A}$ needs to cancel the term m_i to get g_1^{α}. However, it is impossible to reconstruct α since any input access policy $\mathbf{M}$ cannot be satisfied by the attribute sets $\mathcal{S}^*$. Therefore, $\mathcal{A}$ cannot gain a non-negligible advantage in the IND-CPA game.

□

Theorem 3. *Our ABS/ ABSC schemes have the attribute-identity key binding property if BLS signature scheme is EU-CMA (Existential Unforgeability under a Chosen Message Attack) secure and the DH problem is hard.*

Proof. Suppose that an adversary gets access to an attribute key set $sk^* : (sk_1^*, sk_2^*)$ and wants to create a signature without the knowledge of the secret identity key x^* that corresponds to PK^* under which the attribute keys were issued by the Privacy CA.

Case 1: The adversary proceeds as follows: Picks $s^*, t^*, r_{\alpha}^*, (r_i^*)_{i \in [n_1]} \leftarrow_{\$} \mathbb{Z}_p$. $\mathcal{A}$ then computes $h_m = \mathsf{H}_0(M)$ and $h'_m = \mathsf{H}_0(M')$, however we assumed that the adversary has no information about x^*, r^*, then the adversary chooses a random f for x^* and proceeds by setting $\sigma_1^* := \mathsf{sk}_1^{ft^*}$.

Case 2: The adversary proceeds as follows: Picks $s^*, t^*, r_{\alpha}^*, (r_i^*)_{i \in [n_1]} \leftarrow_{\$} \mathbb{Z}_p$. $\mathcal{A}$ then computes $h_m = \mathsf{H}_0(M)$ and $h'_m = \mathsf{H}_0(M')$, $\sigma_1^* := \mathsf{sk}_1^{x^* t^*} = PK^{r^* t^*}$, however we assumed that the adversary has no information about x^*, r^*, then the adversary chooses a random f for r^* and proceeds by setting $\sigma_1^* := PK^{ft^*}$.

The adversary then proceeds in both cases as normal ABS: $\sigma_2^* := \prod_{i\in\mathcal{I}}(\mathsf{sk}_{2,i}^*)^{\gamma_i s^*}$, $\sigma_3^* := \prod_{i\in\mathcal{I}} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s^*}$ $= \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s}$ since we choose $\forall\, i \in [n_1] \setminus \mathcal{I}\ \gamma_i = 0$, $\sigma_4^* := g_2^{t^*}\ \forall_{i\in\mathcal{I}'}\ \sigma_{5,i}^* = \mathsf{H}_1(i||h'_m||\pi'(i))^{t^*}$ and $\sigma_6^* = X^{t^*} \cdot \mathsf{msg}$.

$\mathcal{A}$ calculates $Y^* = X^{s^*\cdot t^*}$, $Z^* = X^{r_\alpha^*}$, $W^* = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{r_i}$ and the challenge $c^* = \mathsf{H}_0(\sigma_1^*, \sigma_2^*, \sigma_3^*, \sigma_4^*, (\sigma_{5,i}^*)_{i\in\mathcal{I}'}, \sigma_6^*, Y^*, Z^*, W^*)$. It then creates proof of knowledge of the parameters $s_\alpha^* = r_\alpha^* - s^* \cdot t^* \cdot c^*$. Finally, $\sigma^* := (\sigma_1^*, \sigma_2^*, \sigma_3^*, \sigma_4^*, (\sigma_{5,i}^*)_{i\in\mathcal{I}'}, \sigma_6^*, c^*, s_\alpha^*, (s_i^*)_{i\in[n_1]})$. We want to argue that σ^* is accepted if σ_1^* is well constructed, and this cannot be completed without the knowledge of x^* or r^*. The adversary proceeds without the knowledge of r^* or x^* as follows:

Assuming that the adversary gets the required attribute keys, the signature is only verified if $Y' = e(\sigma_2^*, \sigma_4^*)/e(\sigma_3^*, \sigma_1^*)$ and $Z' = X^{s_\alpha^*}\cdot Y^{*c^*}$ and $W' = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{s_i^*}\cdot\sigma_3^{*c^*}$. $c' = c^* = \mathsf{H}_0(\sigma_1^*, \sigma_2^*, \sigma_3^*, \sigma_4^*, (\sigma_{5,i}^*)_{i\in\mathcal{I}'}, \sigma_6^*, Y', Z', W')$. This means that $Y^* = Y'$ equals

$$Y' = \frac{e(\prod_{i\in\mathcal{I}}(g_1^{\gamma_i s^* M_i(\alpha||\vec{V})^\intercal} \cdot \mathsf{H}_1(i||h_m||\pi(i))^{x^* r^* \gamma_i s^*}, g_2^{t^*})}{e(\prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s^*}, \mathsf{sk}_1^{ft^*})}$$

or

$$Y' = \frac{e(\prod_{i\in\mathcal{I}}(g_1^{\gamma_i s^* M_i(\alpha||\vec{V})^\intercal} \cdot \mathsf{H}_1(i||h_m||\pi(i))^{x^* r^* \gamma_i s^*}, g_2^{t^*})}{e(\prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s^*}, PK^{ft^*})}$$

In both cases, this is only true if $f = r^*$, which contradicts our assumption that the Privacy CA is not corrupt, or $f = x^*$, which contradicts that the device identity key is unknown to the adversary. The adversary is not able to find $g_2^{x^* r^*}$ (due to the hardness of the DH) even with the knowledge of $g_2^{x^*}$ and $g_2^{r^*}$. Hence, the adversary succeeds in this game by randomly guessing x^* (due to the unforgeability of the BLS signature $\mathsf{H}_1(i||h_m||\pi(i))^{x^*}$) or r^* (due to hardness of the DL) with a probability of $1/q + 1/q = 2/q$, which is low for a large enough q. □

□

Theorem 4. *Our credential is unforgeable if the BBS signature is unforgeable.*

Proof. Our credential is generated by the BBS signature algorithm. Therefore, the credential is unforgeable due to the unforgeability of the BBS signature. □

Theorem 5. *Our scheme offers attestation-identity key binding, if the credential is unforgeable, the hash function is collision-resistant and the the zero-knowledge proof π_1 is simulation-sound extractable.*

Proof. Suppose an adversary gains access to a corrupt attestation key sk_{VC} and attempts to create a signature without knowledge of the secret identity key sk_R. There are two cases according two steps in the protocol, i.e., the device requests to join the domain by proving the possession of the attestation sk_{VC} using π_1 and the credential issuance from the issuer.

Case 1:

- In the request to join the domain step, the device sends the VC request on attributes $\{a_i\}$ to the Privacy CA.
- The Privacy CA returns a challenge c to the device. After receiving the challenge c, the device generates the attestation key sk_{VC} and PK_{VC} (which can also be converted to DAA secret/public key pair). c and PK_{VC} are sent to the manufacturer, who returns a random challenge c' to the device.
- Then the device compute a signature using the hash MAC and a signature algorithm $Sign$. In particular, sig_D^R is generated by the sk_R. If the adversary cannot access to the sk_R, he cannot generate a valid sig_D^R even he can using a random choosing value sk_R^* due to the collision-resistance of the hash function. The probability that this case happened is negligible.

Case 2: If the adversary got a valid sig_M during communications between the device and manufacturer, the adversary can send it to the Privacy CA and obtain a VC. During this process, the adversary needs to prove the possession of the secret key and attributes using the zero-knowledge proof. Because the adversary does not know the secret key from sig_M the zero-knowledge proof is simulation-sound extractable, the adversary cannot generate a valid proof by only knowing the attestation key. The probability that this case happened is negligible.

Case 3: In the last step, the Privacy CA will send the VC to the device. The VC is generated using the BBS signature, which binds attestation key with the credential. Because the BBS signature is unforgeable, the adversary cannot replace the valid key with a forged key to forge a valid VC. The probability that this case happened is negligible.

To summarize, our scheme offers attestation-identity key binding. □

To conclude, the scheme offers the attestation-identity key binding and attribute-key binding properties. Based on the transitive of these two key binding properties, identity key, attribute key and attestation key are all bounded together.

6.4.3 Security Model and Analysis of Zero-Touch Onboarding (ZTO)

The ZTO process in REWIRE consists of two stages, i.e., VC issuance and domain enrolment. As we have described the primitives used in ZTO and associated security proofs. In this section, based on the security requirements in subsection 6.4.1 and security analysis in subsection 6.4.2, we give the security analysis of the whole ZTO protocol. The ZTO protocol is supposed to satisfy *attribute privacy*, *unforgeability*, *attribute-identity key binding* and *attestation-identity key binding*, all of which are defined in subsection 6.4.1.

System model. The system model considers the following entities:

- **Device**. They are potentially untrusted that can interact with the Privacy CA, manufacturer, domain manager and DAA issuer.
- **Privacy CA**. The Privacy CA takes charge of issuing attributes for the device. Moreover, the Privacy CA can be corrupt.
- **Manufacturer**. Manufacturer is trusted and can access to the device's root ID key, which can be used to verify device's signatures.
- **DAA Issuer**. The DAA issuer is fully trusted and in charge of issuing a credential for the device.
- **Domain Manager**. Domain manager is fully trusted, who decides whether accepting the device's enrolment request or not.

Threat model. In the threat model, we consider an adversary can corrupt devices, Privacy CA.

Firstly, the adversary's target is to forge a signature that can bypass the attestation procedure and evade detection. The adversary can try to forge a signature with a predicate that his attributes do not satisfy.

Secondly, except the unforgeability, the adversary also can try to distinguish which attributes were used to generate a signature or any other identifying information associated with the particular signer amongst the users satisfying the given predicate. This might lead to the device fingerprinting which in turn can disclose information to the adversary of the internal device's characteristics, e.g., operating system version, whitlist of binaries loaded as part of the operational stack, that can further lead to implementation disclosure attacks.

Thirdly, to break the key-binding, the adversary can try to generate a valid signature by accessing some attribute key set without knowing the identity key.

Security requirements: In the ZTO protocol, there are some security requirements, i.e., **attribute privacy**, **unforgeability**, **IND-CPA**, **attribute-identity key binding** and **attestation-identity key binding (credential-identity key binding)**. All of them have been defined in subsection 6.4.1.

Theorem 6. *The ZTO protocol offers attribute privacy if the zero-knowledge proof (the Schnorr signature) can offer zero-knowledge property.*

Proof. The attribute privacy is achieved in our schemes by the construction of σ_3 and the Schnorr signatures as follows:
$\sigma_3 := \prod_{i\in\mathcal{I}} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s} = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s}$ since we choose $\forall\ i \in [n_1]/\mathcal{I}\ \gamma_i = 0$. Note that a Shnorr signature is created on all the attributes in the matrix policy M that represents the policy for the Universe attribute set $\mathcal{U}$, this not only helps in verifying the construction of σ_3 and hence σ but also hides the attributes used by the signer to satisfy $\mathcal{P}$ into a larger set $\mathcal{U}$. □

Theorem 7. *The ZTO protocol is unforgeable if the hash function is collision-resistant used during the VC issuance, the zero-knowledge proof π is simulation-sound extractable, the ABS algorithm used in ABSC is unforgeable, the KP-ABE scheme used in the ABSC is adaptively IND-CPA secure.*

Proof. In this proof, we need to prove that the advantage of an adversary $\mathcal{A}$ in forging a valid signature without knowing a valid device's identity key and attribute keys is negligible.

During the VC issuance, the manufacturer is trusted, the device uses the hash function to generate the digest as a signature sig_D^R. Because the hash function is collision-resistant, the adversary $\mathcal{A}$ cannot forge a valid sig_D^R* to make it equal to sig_D^R with a different root ID key sk_R^*. This case happens with a negligible probability.

Then in the device enrolment phase, the domain manager and privacy CA are not trusted. The device needs to request to join with sending the zero-knowledge proof of a VC over B and sk. The adversary cannot get the knowledge of sk due the simulation-sound extractability. Further, the DAA issuer is trusted, suppose the adversary can output a valid credential $cred$, which means $\mathcal{A}$ can either break the BDHI assumption that the ABS signature is not unforgeable or get the secret key of the DAA issuer that the KP-ABE is not adaptively IND-CPA. The probability of these cases happen is negligible. Therefore, the ZTO protocol can offer unforgeability. □

Theorem 8. *The ZTO protocol provides attribute-identity key binding, i.e., it is computationally infeasible for an adversary to replace/exclude any of the device's signatures with a set of attribute keys but without knowing a valid identity key if the ABS/ABSC scheme offers attribute-identity key binding property.*

Proof. Suppose an adversary only knows a set of attribute keys without knowing the identity key, then the adversary wants to generate a valid signature. This means the adversary can break the key binding property in the ABS/ABSC scheme.

Concretely, suppose that an adversary gets access to an attribute key set $sk^* : (sk_1^*, sk_2^*)$ and wants to create a signature without the knowledge of the secret identity key x^* that corresponds to PK^* under which the attribute keys were issued by the Privacy CA.

Case 1: The adversary proceeds as follows: Picks $s^*, t^*, r_\alpha^*, (r_i^*)_{i\in[n_1]} \leftarrow_\$ \mathbb{Z}_p$. $\mathcal{A}$ then computes $h_m = \mathsf{H}_0(M)$ and $h'_m = \mathsf{H}_0(M')$, however we assumed that the adversary has no information about x^*, r^*, then the adversary chooses a random f for x^* and proceeds by setting $\sigma_1^* := \mathsf{sk}_1^{ft^*}$.

Case 2: The adversary proceeds as follows: Picks $s^*, t^*, r_\alpha^*, (r_i^*)_{i\in[n_1]} \leftarrow_\$ \mathbb{Z}_p$. $\mathcal{A}$ then computes $h_m = \mathsf{H}_0(M)$ and $h'_m = \mathsf{H}_0(M')$, $\sigma_1^* := \mathsf{sk}_1^{x^*t^*} = PK^{r^*t^*}$, however we assumed that the adversary has no information about x^*, r^*, then the adversary chooses a random f for r^* and proceeds by setting $\sigma_1^* := PK^{ft^*}$.

The adversary then proceeds in both cases as normal ABS: $\sigma_2^* := \prod_{i\in\mathcal{I}}(\mathsf{sk}_{2,i}^*)^{\gamma_i s^*}$, $\sigma_3^* := \prod_{i\in\mathcal{I}} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s^*}$ $= \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s}$ since we choose $\forall\ i \in [n_1] \setminus \mathcal{I}\ \gamma_i = 0$, $\sigma_4^* := g_2^{t^*}\ \forall_{i\in\mathcal{I}'}\ \sigma_{5,i}^* = \mathsf{H}_1(i||h'_m||\pi'(i))^{t^*}$ and $\sigma_6^* = X^{t^*} \cdot \mathsf{msg}$.

$\mathcal{A}$ calculates $Y^* = X^{s^* \cdot t^*}$, $Z^* = X^{r^*_\alpha}$, $W^* = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{r_i}$ and the challenge $c^* = \mathsf{H}_0(\sigma^*_1, \sigma^*_2, \sigma^*_3, \sigma^*_4, (\sigma^*_{5,i})_{i\in\mathcal{I}'}, \sigma^*_6, Y^*, Z^*, W^*)$. It then creates proof of knowledge of the parameters $s^*_\alpha = r^*_\alpha - s^* \cdot t^* \cdot c^*$. Finally, $\sigma^* := (\sigma^*_1, \sigma^*_2, \sigma^*_3, \sigma^*_4, (\sigma^*_{5,i})_{i\in\mathcal{I}'}, \sigma^*_6, c^*, s^*_\alpha, (s^*_i)_{i\in[n_1]})$. We want to argue that σ^* is accepted if σ^*_1 is well constructed, and this cannot be completed without the knowledge of x^* or r^*. The adversary proceeds without the knowledge of r^* or x^* as follows:

Assuming that the adversary gets the required attribute keys, the signature is only verified if $Y' = e(\sigma^*_2, \sigma^*_4)/e(\sigma^*_3, \sigma^*_1)$ and $Z' = X^{s^*_\alpha} \cdot Y^{*c^*}$ and $W' = \prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{s^*_i} \cdot \sigma^{*c^*}_3$. $c' = c^* = \mathsf{H}_0(\sigma^*_1, \sigma^*_2, \sigma^*_3, \sigma^*_4, (\sigma^*_{5,i})_{i\in\mathcal{I}'}, \sigma^*_6, Y', Z', W')$. This means that $Y^* = Y'$ equals

$$Y' = \frac{e(\prod_{i\in\mathcal{I}}(g_1^{\gamma_i s^* M_i(\alpha||\vec{V})^\intercal} \cdot \mathsf{H}_1(i||h_m||\pi(i))^{x^* r^* \gamma_i s^*}, g_2^{t^*})}{e(\prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s^*}, \mathsf{sk}_1^{ft^*})}$$

or

$$Y' = \frac{e(\prod_{i\in\mathcal{I}}(g_1^{\gamma_i s^* M_i(\alpha||\vec{V})^\intercal} \cdot \mathsf{H}_1(i||h_m||\pi(i))^{x^* r^* \gamma_i s^*}, g_2^{t^*})}{e(\prod_{i\in[n_1]} \mathsf{H}_1(i||h_m||\pi(i))^{\gamma_i s^*}, PK^{ft^*})}$$

In both cases, this is only true if $f = r^*$, which contradicts our assumption that the Privacy CA is not corrupt, or $f = x^*$, which contradicts that the device identity key is unknown to the adversary. The adversary is not able to find $g_2^{x^* r^*}$ (due to the hardness of the DH) even with the knowledge of $g_2^{x^*}$ and $g_2^{r^*}$. Hence, the adversary succeeds in this game by randomly guessing x^* (due to the unforgeability of the BLS signature $\mathsf{H}_1(i||h_m||\pi(i))^{x^*}$) or r^* (due to hardness of the DL) with a probability of $1/q + 1/q = 2/q$, which is low for a large enough q. □

□

Theorem 9. *Our ZTO protocol offers attestation-identity key binding, if the credential is unforgeable, the hash function is collision-resistant and the the zero-knowledge proof π_1 is simulation-sound extractable.*

Proof. Suppose an adversary gains access to a corrupt attestation key sk_{VC} and attempts to create a signature without knowledge of the secret identity key sk_R. There are two cases according two steps in the protocol, i.e., the device requests to join the domain by proving the possession of the attestation sk_{VC} using π_1 and the credential issuance from the issuer.

Case 1:

- In the request to join the domain step, the device sends the VC request on attributes $\{a_i\}$ to the Privacy CA.
- The Privacy CA returns a challenge c to the device. After receiving the challenge c, the device generates the attestation key sk_{VC} and PK_{VC} (which can also be converted to DAA secret/public key pair). c and PK_{VC} are sent to the manufacturer, who returns a random challenge c' to the device.
- Then the device compute a signature using the hash MAC and a signature algorithm $Sign$. In particular, sig^R_D is generated by the sk_R. If the adversary cannot access to the sk_R, he cannot generate a valid sig^R_D even he can using a random choosing value sk^*_R due to the collision-resistance of the hash function. The probability that this case happened is negligible.

Case 2: If the adversary got a valid sig_M during communications between the device and manufacturer, the adversary can send it to the Privacy CA and obtain a VC. During this process, the adversary needs to prove the possession of the secret key and attributes using the zero-knowledge proof. Because the adversary does not know the secret key from sig_M the zero-knowledge proof is simulation-sound extractable, the adversary cannot generate a valid proof by only knowing the attestation key. The probability that this case happened is negligible.

Case 3: In the last step, the Privacy CA will send the VC to the device. The VC is generated using the BBS signature, which binds attestation key with the credential. Because the BBS signature is unforgeable, the adversary cannot replace the valid key with a forged key to forge a valid VC. The probability that this case happened is negligible.

To summarize, the ZTO protocol offers attestation-identity key binding. □

6.5 Experimental Set Up & Evaluation

Having described both flavors of the Zero-Touch Onboarding (ZTO) scheme, we now present a detailed evaluation of its performance when used as a mechanism for demonstrating proof of ownership across an entire attribute space.

For our experimental setup, we implemented the full ZTO scheme in C, leveraging the mbedTLS cryptographic library atop the Keystone framework—a trusted execution environment (TEE) for RISC-V platforms [30]. All evaluations were conducted on a StarFive VisionFive 2 board, equipped with a quad-core 1.5 GHz CPU and 4 GB of RAM. The protocol was executed 1,000 times to ensure consistency in the performance measurements.

To realistically reflect our target use cases, the evaluation scenario focused on scaling the number of attributes involved in the protocol—from 5 to 10, and up to 15 attributes, which represents typical configurations for applications in this domain.

For a granular performance analysis, we decomposed the ZTO protocol into nine distinct phases:

1. **Setup Parameters:** The generation of cryptographic keys—outlined in the setup section—is considered out of scope for the benchmarking results, as these keys are created only once during system initialization. Specifically, the Manufacturer and Privacy Certification Authority (PCA) each generate their respective public/private key pairs. A symmetric key is also established to secure communications between the edge device and the Manufacturer. Once these foundational components are in place, the device initiates an authenticated enrollment process.
2. **Authenticated Enrollment:** During authenticated enrollment, the device must demonstrate legitimate possession of an elliptic curve (EC) key pair. This is achieved through the generation and submission of three cryptographic signatures. The goal is to prove to the Manufacturer—and subsequently to the PCA that the presented public key genuinely belongs to the device, all while preserving device anonymity. To this end, the device responds to random challenges from both the PCA and the Manufacturer. It generates a fresh EC key pair and produces a Schnorr signature over these challenges, alongside an additional asymmetric signature intended for the PCA. The Manufacturer intermediates between the device and the PCA, validating signatures and forwarding verified information to the PCA, which ultimately confirms the authenticity of the device's key ownership.
3. **Attribute Key Creation:** In this phase, the PCA generates unique attribute keys for each attribute. For every attribute, a secure random number is used to generate a private key, from which a corresponding public key is derived.
4. **Verifiable Credential (VC) Creation:** The generated attribute keys serve as building blocks for Verifiable Credentials (VCs). The PCA selects a set of relevant attributes and issues a VC that enables selective disclosure of these attributes by the device. This credential operates in a fashion similar to BBS+ signature-based credentials. The PCA cryptographically binds the attribute keys to the device's authenticated public key, producing a composite signature that ensures both authenticity and linkage of credentials.
5. **VC Verification:** Upon receiving the VCs, the device performs local verification to confirm their integrity and authenticity before storing them for subsequent use. This involves computing a bilinear pairing between the received VCs and the PCA's public key

6. **Verifiable Presentation (VP) Generation:** Once VC verification is successful, the device proceeds to generate a Verifiable Presentation (VP) based on the verified credentials. The VP is then submitted as part of an enrollment request to the target domain.
7. **VP Verification:** The domain performs VP verification to ensure that the disclosed attributes match those used during credential issuance, and that the presentation originates from the correct device public key. This step relies on the chain of trust established during the PCA's issuance of VCs.
8. **Direct Anonymous Attestation (DAA) VC Creation:** If the domain enforces privacy-preserving requirements for membership, the domain manager (DM), after verifying the VP, triggers an upgrade of the credentials to variants compatible with Direct Anonymous Attestation (DAA). This ensures enhanced privacy when operating within sensitive environments.
9. **DAA VC Verification:** Finally, the device verifies the upgraded DAA credentials. If no privacy requirements are specified, the device instead receives a Domain ID Key from the domain manager, which enables its trusted participation within the domain.

In the tables below (Tables 6.2, 6.3, and 6.4), you can see the detailed results of our evaluation. By scaling the number of attributes from 5 to 10 and then to 15, we observe that only **two phases are affected** by the increase in attribute count: the **attribute key creation** phase, which is a sub-phase of the VC creation, and the **PCA VP generation** phase. The attribute key creation phase shows a clear **linear increase** as the attribute count grows. Specifically, the measured averages are **628.89 ms** for 5 attributes, **1256.57 ms** for 10 attributes, and **1885.16 ms** for 15 attributes. This is an expected result, as each attribute requires the generation and processing of a separate key, resulting in a proportional increase in computational workload.

On the other hand, the PCA VP generation phase exhibits only a **negligible increase** as the attribute count increases. The execution times observed are **704.02 ms** for 5 attributes, **707.92 ms** for 10 attributes, and **711.76 ms** for 15 attributes. This minor growth is due to the slightly larger amount of data involved in creating the presentation, but overall the performance impact remains almost constant, highlighting the **excellent scalability** of this phase.

All other phases of the protocol remain **unaffected** by the number of attributes. Their execution times are stable across all configurations tested, with only minimal fluctuations that fall well within the range of expected variance. This confirms that these parts of the ZTO protocol are independent of the attribute scaling and retain their efficiency even when deployed with larger attribute sets. In summary, the results demonstrate that, aside from the predictable linear growth in the key-generation phase, the overall ZTO scheme remains **highly efficient** and well-suited for practical deployments requiring a larger number of attributes.

Phase	Min(ms)	Max(ms)	Average(ms)	Std(ms)
set up parameters	13.39	13.63	13.43	0.0710
authedicated enrollement	334.84	336.36	335.73	0.46
attribute key creation	627.78	629.47	628.89	0.52
VC creation	695.99	698.29	697.33	0.67
verify PCA VCs	46.08	46.32	46.15	0.07
PCA VPs	702.92	705.17	704.02	0.69
VP verification	416.37	417.69	416.94	0.34
DAA VC creation	196.82	197.85	197.29	0.28
DAA VC verification	41.40	41.43	41.41	0.01

Table 6.2: ZTO Evaluation: 5 Attributes

Phase	Min(ms)	Max(ms)	Average(ms)	Std(ms)
set up parameters	13.37	13.43	13.40	0.01
authedicated enrollement	334.55	336.04	335.46	0.49
attribute key creation	1254.65	1258.79	1256.57	1.61
VC creation	1326.87	1331.14	1328.89	1.66
verify PCA VCs	49.97	50.05	50.02	0.02
PCA VPs	706.55	708.80	707.92	0.91
VP verification	415.88	417.50	416.68	0.54
DAA VC creation	196.88	197.75	197.31	0.28
DAA VC verification	41.27	41.39	41.36	0.03

Table 6.3: ZTO Evaluation: 10 Attributes

Phase	Min(ms)	Max(ms)	Average(ms)	Std(ms)
set up parameters	13.30	13.56	13.42	0.06
authedicated enrollement	334.68	336.62	335.36	0.55
attribute key creation	1882.41	1888.81	1885.16	2.09
VC creation	1958.50	1965.14	1961.39	2.19
verify PCA VCs	53.87	53.99	53.94	0.03
PCA VPs	710.24	712.87	711.76	1.05
VP verification	416.05	417.78	416.82	0.68
DAA VC creation	196.87	197.86	197.36	0.35
DAA VC verification	41.28	41.40	41.37	0.03

Table 6.4: ZTO Evaluation: 15 Attributes

Chapter 7

REWIRE Instantiation of ABSC

7.1 Attributes in REWIRE

7.1.1 Introduction

Entities in the REWIRE system have, or are assigned, a set of attributes. These attributes are the fundamental, verifiable facts that represent an entity's identity, capabilities, or current operational state. They serve as the basis for all policy decisions and are essential for the trust assessment processes within the framework.

The scope of attributes in REWIRE is comprehensive, designed to capture a holistic and verifiable view of a device's posture. This approach aligns with emerging standards for remote attestation, such as the 'trustworthiness-vector' model from the IETF RATS working group [32] which provides a structured way to represent different classes of evidence about a device. An attribute could, for instance, describe an immutable property of the device's hardware, represent the cryptographic measurement of a critical software component, or capture a dynamic property reflecting its live runtime configuration.

Given a device's collection of attributes, we must decide if it is sufficient to permit a specific operation. In REWIRE, these decisions are based on policies, $\mathcal{P}$. If a device's attributes satisfy the policy, it is authorized to proceed. For instance, a policy to authorize the execution of a critical enclave could depend on evidence spanning multiple aspects of the device's state. Such a policy might be expressed as:

$$\mathcal{P} = ((\text{hardware} - \text{label} : \text{soc\_type} = \text{rewire\_v1}) \wedge (\text{executables} - \text{label} : \text{sm\_hash} = \text{0xabc...}) \\ \wedge (\text{configuration} - \text{label} : \text{secure\_boot} = \text{enabled})). \quad (7.1)$$

This policy dictates that the action is only permitted on a device with a specific hardware type, running a trusted version of the Security Monitor (verified by its measurement), and which has been booted securely. This model provides granular, flexible, and automated control over critical lifecycle operations based on verifiable evidence.

While convenient for human interpretation, for automated processing these policies are converted into formal structures like access trees and monotone span programs. These are now described before we give details of how they are used for access control and for protecting data in the system.

7.1.2 Access Trees and Monotone Span Programs

Access trees and monotone span programs are used to express access policies.

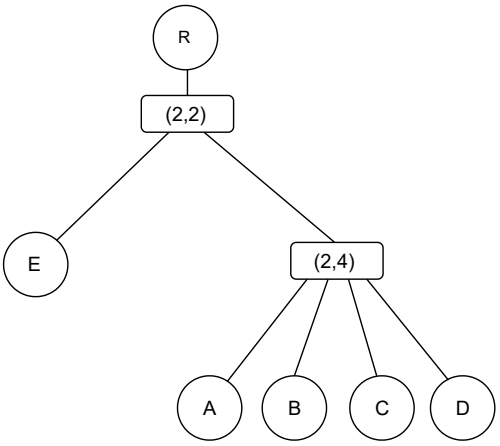

Figure 7.1: Access tree using threshold gates.

Access Trees An example access tree is shown in Figure 7.1 (taken from the example in [17]). At the root of the tree is a root value, R, while at the leaves are values associated with the attributes. How the R value is used will vary depending on the protocol being used, it may be a Boolean value, or a value to be used directly in the protocol. In all cases the root value will only be available if the user's attributes 'satisfy' the tree. In the example the access tree represents the condition that to obtain the value R the user should have attribute E and two of the attributes A, B,C and D. Given values for the attributes A, B, C and D we work up the tree (if the user does not have a value for a leaf attribute it is replaced by $\perp$). As we move up the tree if the children of a node meet the required threshold then that node's value can be passed up the tree, otherwise $\perp$ is passed up instead.

When the tree represents a Boolean circuit, then the attributes and the root value are all Boolean values (0 or 1). For the example shown $R = E \wedge (((A \wedge B) \vee (C \wedge D)) \vee ((A \vee B) \wedge (C \vee D)))$ and this is shown as a 'binary' tree in Figure 7.2.

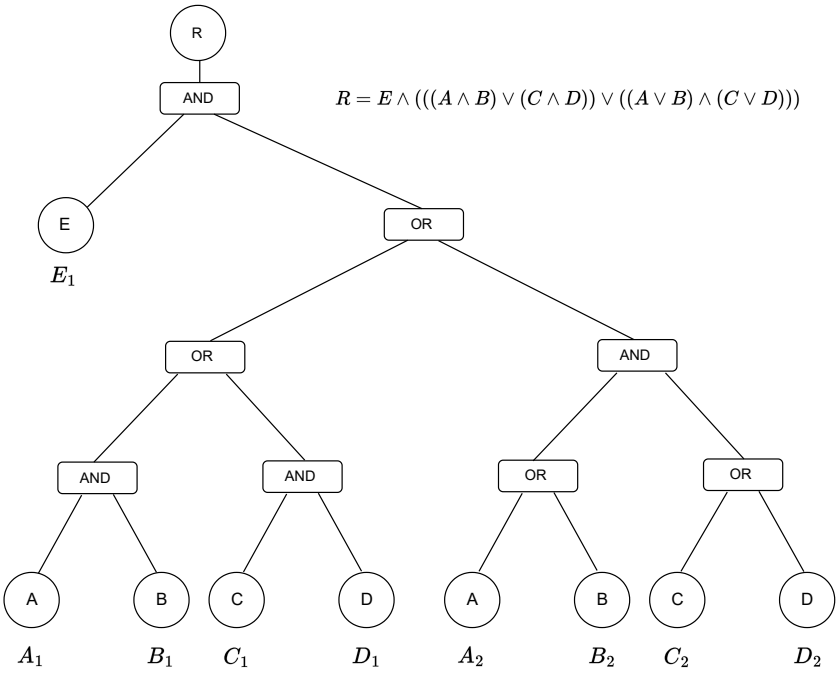

Figure 7.2: Access tree (Figure 3 from [17]).

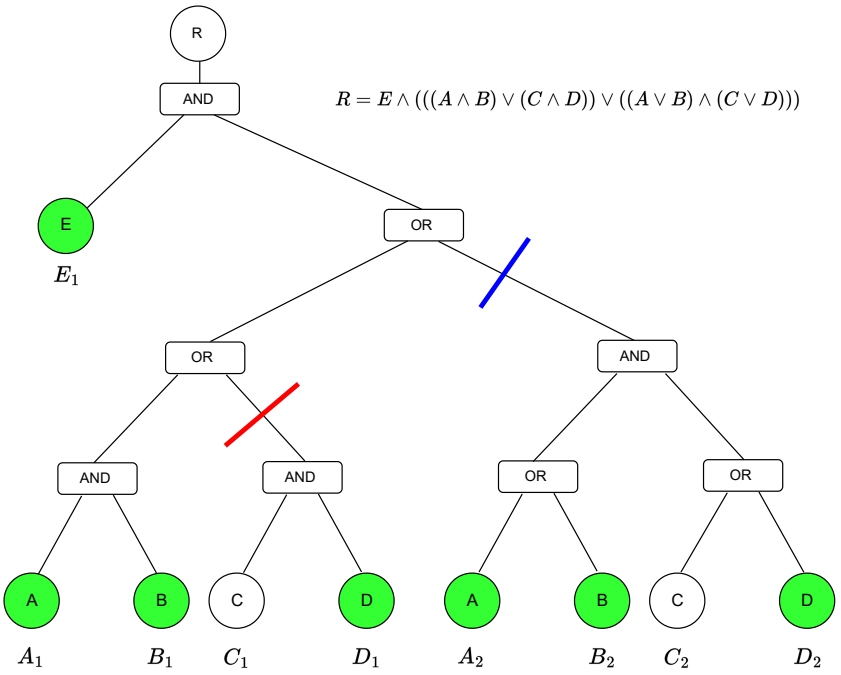

Figure 7.3: The pruned access tree.

As before to obtain the root value, we start at the leaves and work up the tree. For the example, suppose that a user has attributes E, A, B and D, we can find out whether they satisfy the policy by pruning the access tree (see Figure 7.3). we see that (E_1, A_1, B_1), (E_1, A_2, D_2) and (E_1, B_2, D_2) are solutions. The red line shows that we can prune the branch where $C \wedge D$ failed, while the blue line shows where we can prune the tree further as we already know that the condition is satisfied.

Monotone Span Programs Monotone span programs (MSPs) provide an alternative mechanism for attribute based access control [14][1]. An MSP is defined by the tuple

$$\langle M, \mu \rangle$$

.

$$M \in \mathbb{Z}_p^{n_1 \times n_2}$$

is a matrix where every row is associated with an attribute and the function μ gives the mapping between the attributes and the rows. For a Boolean circuit like that shown in Figure 7.2 there is a direct mapping between the leaves of the access tree and the rows of the MSP matrix.

As described below, a policy is satisfied, if a linear combination of the rows associated with a user's attributes is equal to the vector

$$(1, 0, \ldots, 0) \in \mathbb{Z}_p^{1 \times n_2}$$

. As an illustration of the principle involved when using an MSP, consider a vector

$$\vec{v} = (s, r_1, \ldots r_{n_2 - 1})$$

where s is a secret and $r_1, \ldots, r_{n_2-1}$ are chosen at random from $\mathbb{Z}_p$. Let the rows of M be M_i with $i \in \{1, \ldots, n_1\}$, then the n_1 elements of $M\vec{v}$ are $M_i\vec{v}$ and these 'hide' s. Let the linear combination of rows associated with a user's attributes that equals $(1, 0, \ldots, 0)$ be $\sum_j \gamma_j M_j$, then $\sum_j \gamma_j M_j \vec{v} = (1, 0, \ldots, 0)\vec{v} = s$ and s can be recovered. The details in the protocols that follow are different, but the underlying idea is

[1] Threshold access trees can be routinely converted into MSPs [17], as can access trees based on AND and OR gates.

the same.

More formally $M \in \mathbb{Z}_p^{n_1 \times n_2}$ and for $r \in [n_1]$, $\mu : r \to \mathcal{A}$. Note that an attribute may be associated with more than one row of M and so the mapping μ may not be injective. In this case, we define a further function $\rho(r) = |\{z \,|\, \mu(r) = \mu(z) \text{ with } z \leq r\}|$ to denote the $\rho(r)^{th}$ occurrence of attribute $\mu(r)$.

Given a set of attributes, $\mathcal{S} \subseteq \mathcal{A}$, access is confirmed if the vector $(1, 0, \ldots, 0) \in \mathbb{Z}_p^{1 \times n_2}$ lies in the span of the rows, M_i of M for which $\mu(i) \in \mathcal{S}$. If we let $\mathcal{I} = \{i : \mu(i) \in \mathcal{S}\}$, then we have

$$\sum_{i \in \mathcal{I}} \gamma_i M_i = (1, 0, \ldots, 0)$$

for some constants $\gamma_i \in \mathbb{Z}_p$. Conversely, $\langle M, \mu \rangle$ does not accept $\mathcal{S}$ if there exists a vector $\omega \in \mathbb{Z}_p^{n_2}$ which is orthogonal to all of the rows $\{M_i : i \in \mathcal{I}\}$ and not orthogonal to $(1, 0, \ldots, 0)$.

Note that writing $\boldsymbol{\gamma} \in \mathbb{Z}_p^{1 \times n_1}$ and setting $\boldsymbol{\gamma}_i = 0$ when $i \notin \mathcal{I}$ an alternative test for access is:

$$\boldsymbol{\gamma} M = (1, 0, \ldots, 0).$$

For the example above, the corresponding MSP (obtained using the Lewko-Waters algorithm [16]) is:

$$M = \begin{pmatrix} 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix} \begin{matrix} E \\ A \\ B \\ C \\ D \\ A \\ B \\ C \\ D \end{matrix}$$

For this MSP we see that, for example, $\mu(3) = B$ and $\rho(3) = 1$. We also have $\mu(7) = B$ and in this case, $\rho(7) = 2$.

As for the access tree example, suppose that the user has attributes E, A, B and D. In this case we have $\gamma_4 = \gamma_8 = 0$, using these in the equation gives:

$$\begin{pmatrix} \gamma_1 & \gamma_2 & \gamma_3 & \gamma_5 & \gamma_6 & \gamma_7 & \gamma_9 \end{pmatrix} \begin{pmatrix} 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Taking the transpose gives:

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ -1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_5 \\ \gamma_6 \\ \gamma_7 \\ \gamma_9 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Re-arranging into row-echelon form gives:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_5 \\ \gamma_6 \\ \gamma_7 \\ \gamma_9 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

We have seven unknowns and just five equations, extracting the null-space gives the solution as:

$$\begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_5 \\ \gamma_6 \\ \gamma_7 \\ \gamma_9 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} + \alpha \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ -1 \\ -1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

In this case, as the γ–values can only be 0, or 1, possible solutions are:

$$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} (E, A, B), \qquad \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} (E, A, D) \quad \text{and} \quad \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} (E, B, D)$$

Note.

- Given a set of attributes we just need to find one solution to confirm that the policy is satisfied. So for the example above, the user has attributes E, A, B and D, but attributes E, A and B are enough to either satisfy the access tree (see Figure 7.3) or solve the MSP equation and show that the policy is satisfied. Further solutions are un-necessary.
- Given a binary access tree, we can obtain the monotone span program using the Lewko-Waters algorithm [16]. The access tree together with the attributes of an entity can also be used to check whether a policy is satisfied and to obtain $\vec{\gamma}$, as pruning the tree is often easier than using linear algebra.

7.2 Benchmarking Elements of the ABSC Scheme

In this section we benchmark the different components of the ABSC scheme outside the context of a TEE (we will provide the TEE benchmarks in D6.2). Timings will depend on the precise details of a given policy, the timings given here are for policies that are simple conjunctions of N_A attributes, this represents the worst case for a policy with N_A attributes as the number of columns in the MSP matrix depends directly on the number of AND gates in the policy, and in addition, for this policy all rows of the MSP matrix are used in the calculations. The benchmarks given here, in Table 7.1, were obtained on a Dell Lattitude 5491 laptop, with an Intel Core i7-8850H Processor (6 Cores, 9M Cache, 2.6GHz) and 32GB of available memory. Timings were obtained using the Linux `clock_gettime` routines with the `CLOCK_PROCESS_CPUTIME_ID` clock. The tests were run on a single core without using Gramine TEE

emulation. As discussed previously the Gramine TEE will add some overhead and this will be reflected in the end-to-end measurements reported in D6.2.

The test program uses the MIRACL Core [6] cryptographic library and was compiled using GCC version

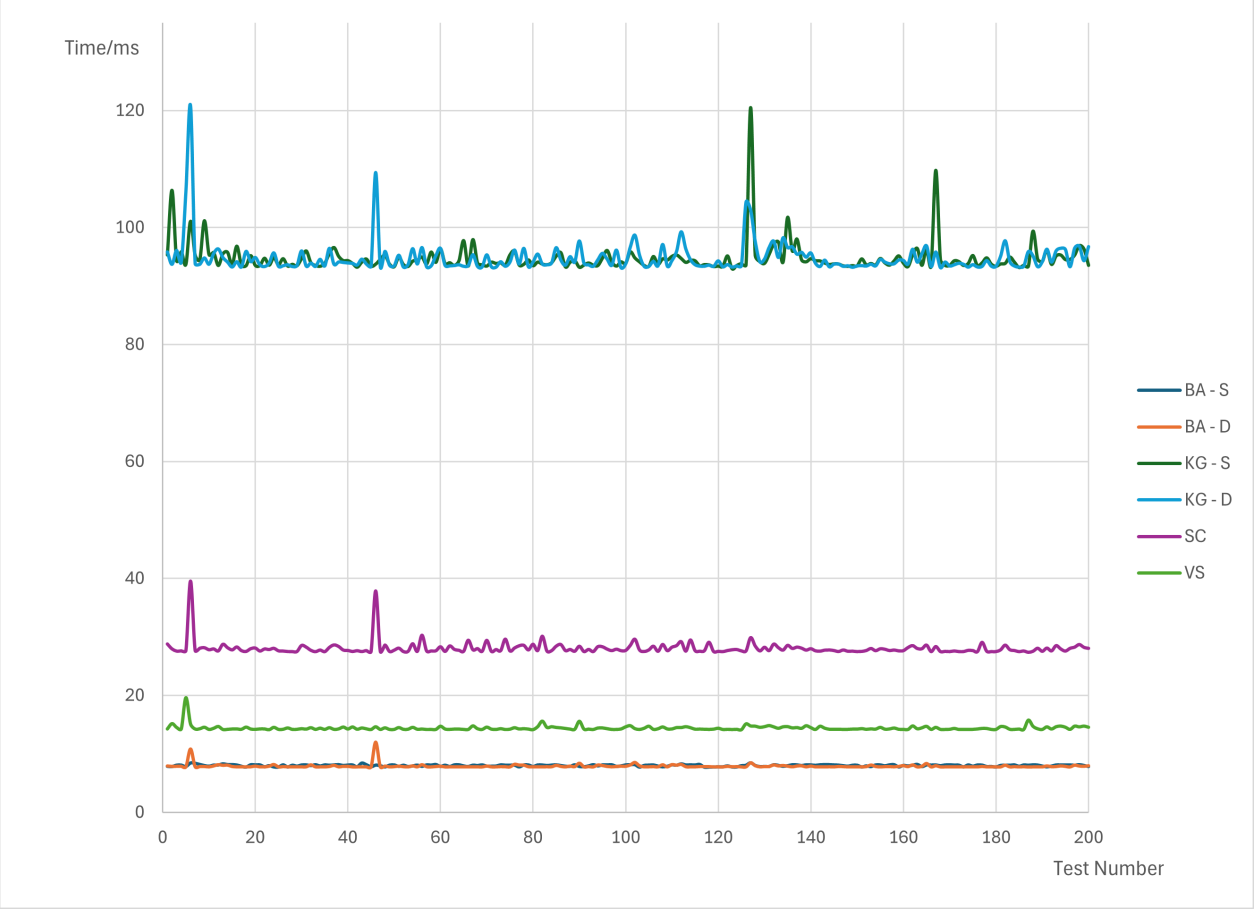

Figure 7.4: The Raw Timing Data for 30 Attributes.

11.4.0. The test program uses the FP256BN elliptic curve and runs the whole protocol checking that the signature verifies and that the shared secret is obtained correctly. The test program was run 200 times and the different functions in the protocol timed as outlined above. Figure 7.4 shows one of the timing sets, the traces are: BA-S – Bind-Attributes for the signer, BA-D – Bind-Attributes for the decryptor, KG-S – Key-Gen for the signer, KG-D – Key-Gen for the decryptor, SC – Sign-Crypt and VS – SignCrypt-Verify.

The traces are typically noisy at the start and then settle down. There are still odd spikes, presumably due to some system events that disrupt the process. To minimise these disruptions the laptop being used was disconnected from the network and the WiFi was disabled. However, as seen in Figure 7.4, disruptions still occur and so we average over 200 tests, this was enough to get consistent results. The timing averages and standard deviations obtained are given in Table 7.1. Normally the policies for signing and encryption will be different and different results for BA-S and BA-D would be obtained, similarly KG-S and KG-D would also be different. For these tests the signing and encryption policies were the same and so the two sets of 200 tests for Bind-Attributes and for Key-Gen are essentially the same (as highlighted in the Figures) and so they were combined for the table.

Figure 7.5 shows the results from the table plotted against the number of attributes.

Table 7.1: Times Measured for the Different ABSC Functions

Function	Number of Attributes	Average Time (ms)	Standard Deviation
Bind-Attributes	5	1.78	0.34
	10	3.01	0.19
	20	5.46	0.22
	30	7.95	0.29
Key-Gen	5	16.40	1.04
	10	31.98	0.92
	20	63.37	1.67
	30	94.78	2.71
Sign-Crypt	5	7.11	0.80
	10	11.21	0.55
	20	19.59	0.86
	30	28.05	1.2
SignCrypt-Verify	5	8.58	0.28
	10	9.69	0.31
	20	12.02	0.23
	30	14.40	0.46

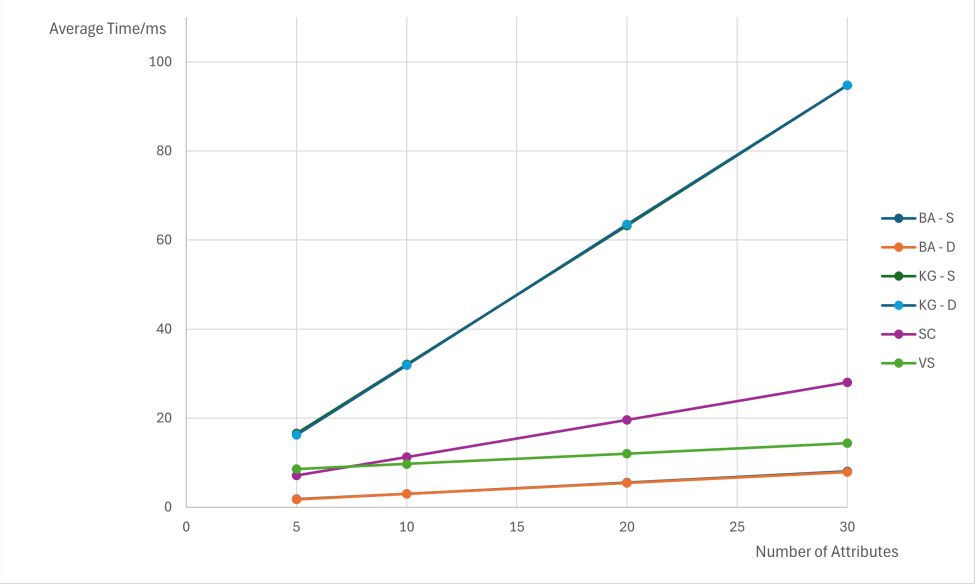

Figure 7.5: Averages vs Number of attributes N_A.

As expected from the function definitions the timings scale linearly with the number of attributes. The Key-Gen function takes much more time than the others as it involves checking pairings for each attribute and the pairings calculations are very expensive. In the protocols, a user only has to call Key-Gen (and Bind-Attributes) once for each policy that they need to use, the keys that they receive can then be stored securely and used as required and so this comparatively heavy cost should not be a problem. The SignCrypt-Verify function also requires pairings calculations, but however many attributes are involved there are only three pairings, there is some increase in the time required with the number of attributes, but this is not significant.

Chapter 8

REWIRE Tracing Capabilities

In order to enable devices to provide verifiable claims regarding their operational correctness, the REWIRE TCB is equipped with sophisticated **tracing capabilities**. These capabilities are the fundamental enabler for **Property-Based Attestation**. A core innovation of REWIRE is that the Design-Phase process, specifically the **Compositional Verification and Validation** toolchain, identifies the precise set of properties that cannot be formally verified and must therefore be attested at runtime. Since the target embedded devices often lack such inherent monitoring functions, the REWIRE Tracer provides this critical capability. This mechanism provides a *verifiable level of tracing*, which is characterized by two distinct security properties: **authenticity** and **integrity**. Authenticity ensures that the traces originate from a valid and authorized Tracer. This is achieved through key restriction usage policies enforced by the Attestation Agent; the public key of the legitimate Tracer is bound to the policy, ensuring that the Agent will only process evidence signed by the corresponding private key. Integrity, on the other hand, ensures that the collected evidence has not been tampered with. The safeguarding of this integrity is anchored to the device's hardware root of trust; the Tracer signs the evidence with its private Tracer Key (TK), which is a hardware-based key managed and protected by the Security Monitor. The REWIRE Tracer is designed to monitor both the configurational and operational profiles of the device's software stack. In its current implementation, documented in this deliverable, the Tracer focuses on monitoring the **static configuration** of a binary to provide the necessary evidence for the CIV scheme. Beyond this, and to accommodate the advanced **Process State Verification** service described in Section 4.2.2, REWIRE introduces a key innovation: enhanced tracing in the form of memory introspection for trusted applications. While most tracers are confined to the untrusted world, REWIRE extends the Security Monitor with additional SBI calls that enable secure, TCB-mediated introspection of enclave memory.

These tracing capabilities are the fundamental enabler for **Property-Based Attestation**. A fundamental challenge that must be overcome to achieve efficient runtime attestation is the identification of a minimal-yet-sufficient set of properties that, if attested and verified, can provide strong guarantees about the correct configuration and operational state of a device. The current convergence in the field is towards such property-based attestation, but the key question remains: how do we identify the required set of properties that correctly captures the volatile state of a device that can change during runtime. By continuously monitoring a specific set of properties, their attestation can be elevated to be equivalent to a proof of the device's correct state. REWIRE is one of the first frameworks to directly address this challenge by combining a **Compositional Verification and Validation** toolchain with runtime monitoring. This design-time process formally identifies the **trust boundary** of a system; any behaviour that can be formally verified is considered within this boundary. What cannot be modelled or verified is abstracted, and it is precisely these abstracted operations that yield the specific set of properties that must be attested during runtime. Since the embedded devices targeted by the project often lack the inherent functions to monitor these specific properties, the REWIRE Tracer is designed to provide this critical capability. As documented in Deliverable D4.2 [26], our survey of state-of-the-art tracing technologies led to the conclusion that a software-based solution, which is non-intrusive and weakens the assumption of requiring access to the

application's source code, is the most practical and scalable approach. Therefore, this chapter details the final REWIRE Tracer implementation, which is focused on extracting the configuration traces necessary to enable the CIV scheme based on these design choices.

While tracers can be based in software, hardware, or a co-design, REWIRE has opted for a less intrusive, **software-based component**. To ensure its own trustworthiness, the Tracer does not operate in complete isolation. Because the Tracer must monitor processes in the untrusted world, it cannot itself be fully instantiated within the trusted world of a secure enclave. To circumvent the security challenges this poses, and to ensure both its integrity and authenticity, the Tracer's trust is anchored directly to the TCB. We assume the Tracer has been equipped with a hardware-rooted **Tracer Key (TK)**, which it uses to cryptographically sign evidence before it is forwarded to the Attestation Agent. The secure transport of this evidence within the device is also a critical consideration. We assume the presence of protection measures, such as those defined by the Security Protection and Data Model (SPDM) protocol [8], for the authenticated encryption of traces when they are shared. SPDM defines a secure session and payload structure for conveying attestation evidence, which can be protected and transported between different layers of a device's software stack. In the REWIRE architecture, this ensures that the signed traces from the Tracer are securely conveyed to the Security Monitor for mediation. Therefore, the REWIRE Tracer is capable not only of verifying that traces originate from an authentic source but also that they have been collected and transmitted with high integrity. This entire security model is accomplished through the use of the pre-shared Tracer Key (TK), which is established with the Attestation Agent and the Verifiable Policy Enforcer as part of a key restriction usage policy enforced by the **Domain Manager**. The foundational hypothesis that a secure, pre-shared key is necessary to anchor the Tracer's trust was formally proven in the security analysis documented in Deliverable D3.3 [24].

8.1 Updates and Additional Features to the REWIRE Tracing Layer

The final design of the REWIRE Tracer has been significantly enhanced from its initial concept to focus on the practical challenges of securing embedded devices. As mentioned, its primary role is to provide the runtime evidence necessary for the Configuration Integrity Verification (CIV) scheme [26]. This evidence, once processed by other components like the Attestation Agent, can then be used to identify threats. The following updates distill the final version of the overall tracing fabric, comprising extendable tracing probes, which was re-scoped to a **daemon-based implementation**. This final design focuses on two key areas: the collection of advanced evidence through memory introspection and significant performance optimization.

A core update to the Tracer's design is its specific focus on collecting the evidence needed to detect sophisticated memory-based attacks that target binaries running in the untrusted world. An adversary leveraging memory-related vulnerabilities can attempt to either disrupt the static integrity of the code base or alter the runtime configuration and control flow of the target system. The REWIRE Tracer is designed to provide evidence for both, with a primary focus on the configuration integrity required for the CIV scheme. This capability is inspired by modern Endpoint Detection and Response (EDR) products, but adapted for the unique constraints of embedded systems which typically lack such ready-made monitoring solutions. Advanced attacks like buffer overflows and Return-Oriented Programming (ROP) directly manipulate the program's control flow. While mechanisms like Control-Flow Attestation (CFA) or Control-Flow Integrity (CFI) are designed to counter these, they are often resource-intensive and may not detect more subtle configuration attacks. For instance, a common exploitation technique is to circumvent code-signing and memory protections by tampering with the permissions of an existing memory segment to make it writable, injecting malicious shellcode, and then reverting the permissions back to executable. This type of attack is particularly insidious as it can bypass CFI by keeping the program's high-level execution flow largely intact [12]. Solely monitoring permissions at discrete intervals could easily miss the transient window where they were changed. The REWRITE Tracer's capabilities complement CFA/CFI by providing the evidence to detect such unauthorized executable code injections through persistent memory monitoring.

To counter such threats, the REWIRE Tracer implements a robust monitoring strategy that inspects both the **permissions** of memory a process is loaded into and the **structure** of that memory. This is achieved by parsing a process's Executable and Linkable Format (ELF) file [3], the standard format for Linux binaries. By inspecting the ELF header and its associated program headers at load time, the Tracer precisely identifies the intended boundaries of legitimate executable code segments (such as '.text'). This deep inspection of the ELF structure post-loading ensures that memory regions are correctly mapped and that critical structures like symbol tables and relocations align with the original binary's logic. By continuously verifying the permissions and structural integrity of these critical memory regions, the Tracer can provide the evidence needed to detect violations, such as those arising from memory-tampering attacks.

In addition to enhancing its evidence-gathering capabilities, we have identified a key path for future performance optimization. During development, it was observed that a primary performance bottleneck in runtime configuration integrity tracing is the repeated parsing of a process's in-memory representation to identify and exclude invalid offsets (e.g., relocation entries in shared libraries) that must be ignored when computing a cryptographic hash. To address this, an intelligent **software caching mechanism** will be implemented and evaluated in D6.2 [28]. The goal is to reduce the expensive parsing overhead so that it is performed only once per monitored process. *The underlying observation for this optimization is that once a (non-JIT) process and its dependent libraries are loaded into memory, their internal structure and the location of these (invalid) offsets should never legitimately change their in-memory representation*. The Tracer now calculates these offsets once upon the first inspection of a binary or library and stores them in a memory-efficient cache. For subsequent integrity checks of the same process, the Tracer reuses the cached offsets, eliminating the expensive parsing step. Crucially, this optimization can also extend to shared libraries[1]; the cached offsets for a library like 'libc.so' can be calculated once and then reused by every process that links against it. This dramatically reduces redundant computations and enhances the overall performance of continuous integrity tracing across the entire system, making it feasible to monitor multiple processes with minimal overhead. These updates culminate in a Tracer that is not only more secure but also significantly more efficient and practical for deployment on resource-constrained edge devices.

8.2 REWIRE SW-based Tracing Capabilities

This section details the requirements, high-level design, and implementation path for the REWIRE Tracer. The core design philosophy behind the Tracer is that it must be non-intrusive and have a minimal impact on the operational and safety profile of the processes it is monitoring. To meet this primary goal, the **Tracer is engineered to run automatically and seamlessly**, meaning a monitored process does not need to interact with it, nor does it need to take any action to facilitate the monitoring process. The Tracer operates as a **silent observer from the perspective of the target application**.

To achieve this, the REWIRE Tracer must first identify which processes require monitoring. This is not a static configuration but is driven by the active **MSPL security policies** that dictate the type of attestation controls to be executed at runtime. As defined in the work of WP3 and documented in D3.3 [24], these policies explicitly include the set of binaries to be traced as part of a given attestation action. Once a target process is identified, the Tracer must be able to read the executable code and static data segments that the process has loaded into memory. A crucial aspect of this design is the ability to delineate the relevant parts of the process's memory footprint (specifically the code and read-only data) while correctly ignoring dynamic memory regions like the stack and heap, which are expected to change continuously during normal execution.

Furthermore, as aforementioned, a core dimension is the **verifiability** of the extracted traces. This is

[1] This is dependent to how the underlying compiler enables the management of shared libraries. In the case, of each process loading their own version of the shared library, each will have a unique set of invalid offsets. Still though the parsing overhead is reduced to only the first runtime integrity tracing request.

achieved by safeguarding two distinct properties: the **integrity** of the evidence and the **authenticity** of its source. **Integrity**, which ensures the traces have not been altered, is guaranteed by rooting the signing process to the underlying secure element in the context of REWIRE, the Security Monitor. The Tracer's private **Tracer Key (TK)** is securely managed by the TCB, and the cryptographic signing operation itself is mediated by the SM. To protect the traces in transit *before* they are signed, we assume the presence of a secure channel mechanism, such as that defined by the **Security Protocol and Data Model (SPDM)** [8]. In this model, the Tracer can establish a secure session with the Security Monitor to transport the collected evidence. Once the SM receives the traces over this protected channel, it signs them using the TK on behalf of the Tracer before forwarding them to the Attestation Agent. **Authenticity**, which ensures the traces originate from the legitimate Tracer, is verified by the Attestation Agent. This is accomplished through the key restriction usage policy that is enforced by the Domain Manager. As introduced in D4.2 [26] and formally verified in D3.3 [24], the public part of the unique Tracer Key (TK) is integrated into this policy. This allows the Attestation Agent to cryptographically verify that the signature on the received evidence was generated by the legitimate TK, confirming the traces' origin. The security of this entire process relies on both *what* evidence is sent and *how* it is cryptographically protected and verified.

8.2.1 High-level Design

The high-level design of the REWIRE Tracer is architected to meet the core requirements of being non-intrusive, automated, and secure. To ensure that it runs automatically and seamlessly, the Tracer is implemented as a daemon that operates continuously in the background of the untrusted world. The activation and operation of the Tracer are driven by security policies enforced by the trusted components of the REWIRE TCB, as depicted in the sequence diagram in Figure 8.1.

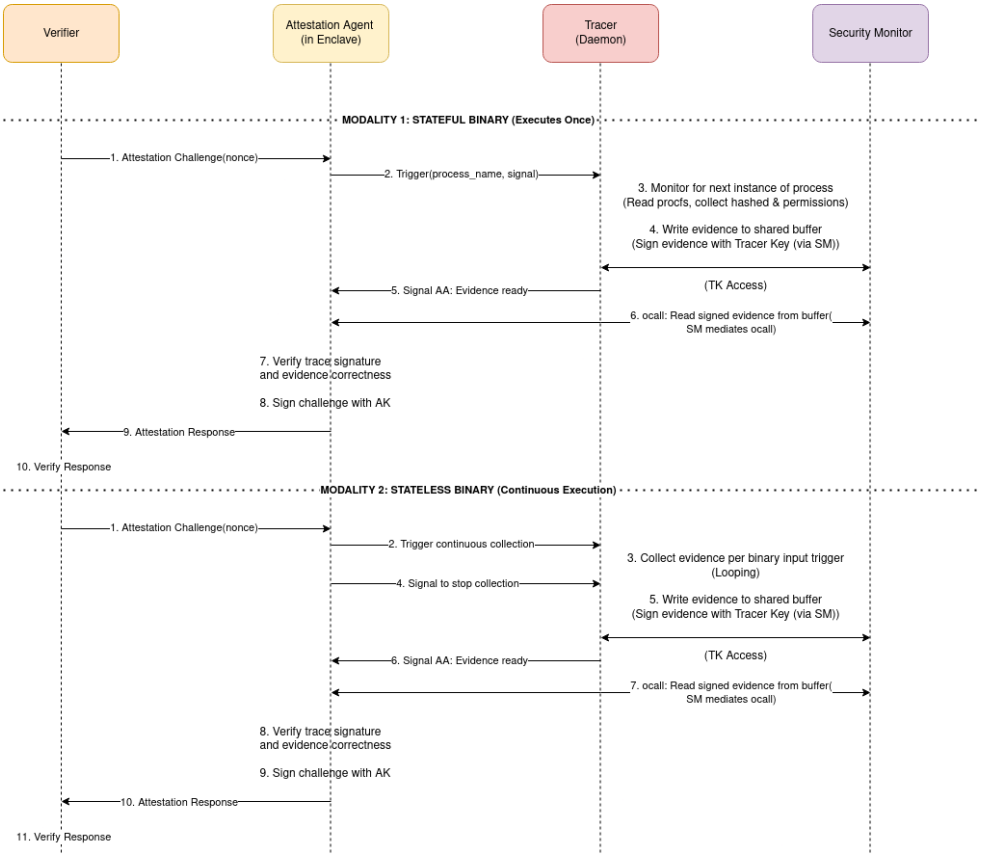

Figure 8.1: High-Level Tracer Flow

The operational flow begins when the untrusted **Facility Layer** of the device decoding the attestation

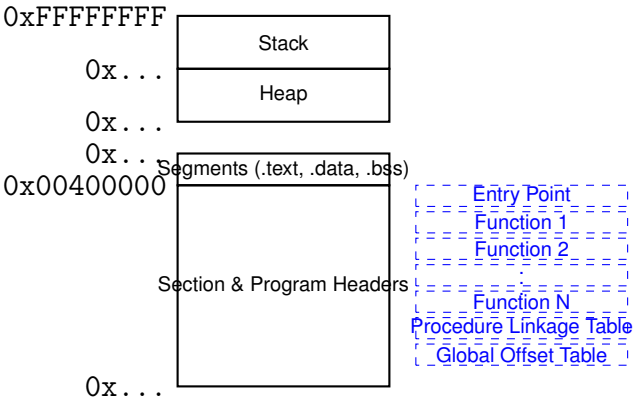

Figure 8.2: Memory Layout of an ELF Binary with Function Symbols and Metadata

execution logic derived from an enforced security policy through the **Policy Orchestrator**. This policy defines the type of attestation control (e.g., CIV or Control-Flow Integrity, etc.) that must be executed and for which properties of the device. The Facility Layer instructs the trusted **Attestation Agent (AA)**, running within its own secure enclave, to act upon this policy. To initiate the tracing task, the AA triggers the Tracer daemon by writing the name of the target process—which can be either stateful or stateless—into a specific configuration file located in the untrusted world. Access to this configuration file is strictly controlled to ensure that only the Attestation Agent and other privileged processes can modify it, thereby protecting the system from an attacker who might attempt to alter the monitoring targets.

Upon activation, the Tracer daemon reads the target process from the configuration file and begins its inspection using the ‘procfs‘ virtual filesystem. This standard interface in Unix-like systems allows for the detailed inspection of a process’s memory layout without requiring special kernel access or any interaction from the monitored process itself. By parsing ‘procfs‘, the Tracer can identify which sections of the process’s ELF file have been mapped into memory, where they are located, and their specific permissions. The Tracer’s primary objective is to verify the integrity of the executable code segments. These sections are meant to be loaded with ‘executable‘ and ‘read-only‘ permissions and must not change during the entire lifetime of the process. If the Tracer detects a violation of these properties—for example, if an executable section’s permissions are tampered with to become writeable, or if its content changes—it registers this as an integrity failure.

Once the Tracer has collected the necessary evidence (e.g., cryptographic hashes of the monitored memory sections), it communicates the results securely to the Attestation Agent. While this is currently manifested through the use of **shared buffers** where the Tracer outputs the monitored traces (signed with the Tracr’s Key managed by the SM), this can be elevated to the establishment of an authenticated encrypted channel through the use of the Secure Protocol and Data Model (SPDM) [8]. The SPDM protocol, defines a payload structure for conveying attestation Evidence contained in certificates and the SPDM Measurement Block. Processing certificates containing Evidence implies Evidence is extracted from the certificate as part of evidence appraisal. Attesters that implement REWIRE tracing typically protect layer-specific evidence in layer-specific certificates that are dynamically generated when a device boots. One of the layers typically implements the SPDM protocol that is used to transport the certificates to the SPDM Requester. The REWIRE layer that implements the SPDM protocol may also be an Attesting Environment that collects additional measurements from one or more Target Environments that are conveyed using an SPDM Measurement Block.

The Attestation Agent, from within its enclave, then performs a secure ‘ocall‘. This ‘ocall‘ is mediated by the **Security Monitor**, which allows the AA to safely read the signed traces from the shared buffer. The AA can then verify the signature on the traces, ensuring that the evidence is both authentic (originating from the correct Tracer) and has maintained its integrity throughout the process.

Persistent State Monitoring: The persistent state captures the stable (Figure 8.2), structural properties of the binary as it is instantiated in the target device. This includes the **in-memory layout derived from the ELF binary**. More specifically, the ELF-based structures include segments such as .text, .data, and .bss, along with resolved program symbols and relocation entries. REWIRE can perform a deep inspection of this structure post loading, ensuring that memory regions are correctly mapped and that symbols and relocations align with the original binary's logic. Any anomalies —- such as unexpected symbol resolution, corrupted relocation — would indicate a compromised or failed migration. Thus, persistent state verification establishes the foundational trust that the binary is genuine, correctly instantiated, and logically aligned with expected behaviour.

Optimizations: To minimize the performance impact of continuous runtime monitoring, the final design will incorporate a critical optimization: **the caching of invalid memory offsets.** A significant bottleneck was identified in repeatedly parsing the memory maps of processes and their loaded libraries to exclude dynamic sections (e.g., relocation data) that should not be part of an integrity hash. Since the layout of a process and its shared libraries is static once loaded, the Tracer now performs this expensive parsing only once. It computes the set of invalid offsets for a given binary or library and stores them in a cache. For all subsequent integrity checks, the Tracer reuses this cached information, avoiding redundant work. This is particularly effective for shared libraries, as their offset information can be calculated once and reused by every process that links against them, dramatically improving the efficiency and scalability of the tracing mechanism on the device.

8.2.2 Implementation Path Report

The REWIRE image is built using Buildroot. Accordingly, the REWIRE Tracer is installed via a Buildroot package. This package is responsible for both building the tracer program and installing it into the Buildroot image. The following files are installed:

- **/etc/init.d/S60monitor**: An init script describing how the daemon can be started and stopped.
- **/usr/share/monitor/.monitor-config**: The configuration file used to specify which processes should be monitored.
- **/usr/bin/monitor**: The executable used to monitor other processes.

Compilation process with buildroot:

```
set(MONITOR_INCLUDE_DIR "${CMAKE_FIND_ROOT_PATH}/usr/include")
set(MBEDTLS_INCLUDE_DIR "${CMAKE_FIND_ROOT_PATH}/usr/include/mbedtls")

set(MBEDCRYPTO_STATIC_LIB "${CMAKE_FIND_ROOT_PATH}/usr/lib/libmbedcrypto.a")
set(MBEDTLS_STATIC_LIB "${CMAKE_FIND_ROOT_PATH}/usr/lib/libmbedtls.a")
```

Config file:

```
config BR2_PACKAGE_MONITOR
        bool "monitoring-daemon"
        help
                Monitoring daemon from Secura, checks the integrity of a process
                and communicates this to the trusted world
```

When the REWIRE image boots, the daemon process is started by the init system. While running, it waits for a SIGINT signal, which is typically sent by the Attestation Agent operating in the trusted world. However, this is not a strict requirement—any process is capable of signaling the Tracer.

The input and output of the Tracer are secured by ensuring that the corresponding files can only be read from and written to by privileged processes. The Tracer's output is written to a shared buffer after being

signed with the Tracer Key. The Security Monitor acts as a mediator, allowing the Attestation Agent to retrieve and verify this data.

Upon receiving the signal, the Tracer reads its configuration file and begins monitoring the configured processes. It identifies these processes by name, as listed in the configuration file, and uses pidof to obtain their process IDs (PIDs). For each identified process, the Tracer reads the corresponding ELF file from procfs. The actual name of the program that launched the process is confirmed by reading the symbolic link /proc/PID/exe. This is important because the program name may differ from the process name.

For instance, in many embedded systems, busybox is used—a single binary that implements various Unix utilities. These utilities are typically invoked via symbolic links to the busybox binary. When executed through such a link, the first argument to busybox is the name of the utility being invoked. As a result, the process name reflects the utility, while /proc/PID/exe points to busybox. This distinction is critical for attestation, since integrity checks must be performed on the actual executable binary rather than on a utility name or process name.

The second stage of reading from procfs involves parsing /proc/PID/maps to list the memory-mapped sections of the executed program. Again, it is important to distinguish between the process name and the actual program, as the executable's ELF section headers also describe these memory sections. The Tracer parses this mapping and filters for executable sections. For each executable section, it records the mapped addresses and creates a memory dump. These memory dumps are then hashed using the SHA-256 implementation provided by the mbedtls library.

The resulting data—both the permissions of each section and the SHA-256 hashes of the corresponding memory dumps—are saved to the file /tmp/monitor-hashes. The information is stored in JSON format to ensure compatibility and ease of parsing. Once the results have been saved, the tracer sends a signal back to the process that originally signaled it. While this is expected to be the facility layer in the trusted world, it is not a strict requirement.

Instantiation of Tracer output (example):

```
{
     "results" : [
    {
        "name": "openssl",
        "exec_sections": [
            {"openssl": {"permissions":"PERMS", "hash":"HASH"}},
            {"libc": {"permissions":"PERMS", "hash":"HASH"}}
        ]
    },
    {
        "name": "bash",
        "exec_sections": [
            {"bash": {"permissions":"PERMS", "hash":"HASH"}},
            {"libc": {"permissions":"PERMS", "hash":"HASH"}}
        ]
    }
]
}
```

Example usage of the Tracer:

When running the REWIRE image, the Tracer runs as a daemon. We can get its PID as follows:

```
# cat /run/monitor.pid
131
```

With the PID we can send a SIGINT signal to the process of the daemon, triggering it to run:

```
# kill -SIGINT 131
# ^C
```

After sending the signal, another SIGINT signal is received in response, indicated by the ^C in the terminal. This means the Tracer has finished collecting its evidence. We can also check the configuration file to verify which processes the Tracer monitors:

```
# cat /usr/share/monitor/.monitor-config
/usr/sbin/dropbear
udhcpc
```

Indeed the output of the Tracer contains the memory permissions and hashes of the mapped sections from these processes:


```
{
    "results": [{
        "name": "/usr/sbin/dropbear",
        "exec_sections": [{
            "/usr/sbin/dropbear": {
                "permissions": "r-xp",
                "hash": "e1...0b"
            }
        }, {
            "/lib/libc.so.6": {
                "permissions": "r-xp",
                "hash": "2d...19"
            }
        }, {
            "/lib/libcrypt.so.1": {
                "permissions": "r-xp",
                "hash": "a3...7d"
            }
        }, {
            "/lib/ld-linux-riscv64-lp64d.so.1": {
                "permissions": "r-xp",
                "hash": "f7...1e"
            }
        }]
    }, {
        "name": "/bin/busybox",
        "exec_sections": [{
            "/bin/busybox": {
                "permissions": "r-xp",
                "hash": "c0...b8"
            }
        }, {
            "/lib/libc.so.6": {
                "permissions": "r-xp",
                "hash": "2d...19"
            }
        }, {
            "/lib/libresolv.so.2": {
```

```
                "permissions": "r-xp",
                "hash": "20...86"
            }
        }, {
            "/lib/ld-linux-riscv64-lp64d.so.1": {
                "permissions": "r-xp",
                "hash": "f7...1e"
            }
        }]
    }]
}
```

If we check the logs of the Tracer, more details can be found of the steps it took to gather this evidence.

```
Received signal 2 from process with PID 132, starting analysis now
Number of targets: 2
PID of target 0: 128
PID of target 1: 123

Monitoring process with PID 128 and process name /usr/sbin/dropbear

aaaaaacf2aa000-aaaaaacf2ed000 r-xp 00000000 fe:00 444 /usr/sbin/dropbear
aaaaaacf2ed000-aaaaaacf2ef000 r--p 00042000 fe:00 444 /usr/sbin/dropbear
aaaaaacf2ef000-aaaaaacf2f0000 rw-p 00044000 fe:00 444 /usr/sbin/dropbear
aaaaaacf2f0000-aaaaaacf2f1000 rw-p 00000000 00:00 0   [heap]
aaaaaacf2f1000-aaaaaacf312000 rw-p 00000000 00:00 0   [heap]
ffffffabc3e000-ffffffabc40000 rw-p 00000000 00:00 0
ffffffabc40000-ffffffabd82000 r-xp 00000000 fe:00 143 /lib/libc.so.6
ffffffabd82000-ffffffabd83000 ---p 00142000 fe:00 143 /lib/libc.so.6
ffffffabd83000-ffffffabd86000 r--p 00142000 fe:00 143 /lib/libc.so.6
ffffffabd86000-ffffffabd88000 rw-p 00145000 fe:00 143 /lib/libc.so.6
ffffffabd88000-ffffffabd95000 rw-p 00000000 00:00 0
ffffffabd95000-ffffffabd9c000 r-xp 00000000 fe:00 144 /lib/libcrypt.so.1
ffffffabd9c000-ffffffabd9d000 ---p 00007000 fe:00 144 /lib/libcrypt.so.1
ffffffabd9d000-ffffffabd9e000 r--p 00007000 fe:00 144 /lib/libcrypt.so.1
ffffffabd9e000-ffffffabd9f000 rw-p 00008000 fe:00 144 /lib/libcrypt.so.1
ffffffabd9f000-ffffffabdcf000 rw-p 00000000 00:00 0
ffffffabdcf000-ffffffabdd1000 r--p 00000000 00:00 0   [vvar]
ffffffabdd1000-ffffffabdd3000 r-xp 00000000 00:00 0   [vdso]
ffffffabdd3000-ffffffabdf2000 r-xp 00000000 fe:00 138 /lib/ld-linux-riscv64-lp64d.so.1
ffffffabdf3000-ffffffabdf4000 r--p 0001f000 fe:00 138 /lib/ld-linux-riscv64-lp64d.so.1
ffffffabdf4000-ffffffabdf6000 rw-p 00020000 fe:00 138 /lib/ld-linux-riscv64-lp64d.so.1
ffffffdf33f000-ffffffdf360000 rw-p 00000000 00:00 0   [stack]

File: /usr/sbin/dropbear
Start address: 0xaaaaaacf2aa000
End address: 0xaaaaaacf2ed000
Permissions: r-xp

File: /lib/libc.so.6
Start address: 0xffffffabc40000
End address: 0xffffffabd82000
```

```
Permissions: r-xp

File: /lib/libcrypt.so.1
Start address: 0xffffffabd95000
End address: 0xffffffabd9c000
Permissions: r-xp

File: /lib/ld-linux-riscv64-lp64d.so.1
Start address: 0xffffffabdd3000
End address: 0xffffffabdf2000
Permissions: r-xp

SHA256 hash of executable section: e1[...]0b
SHA256 hash of executable section: 2d[...]19
SHA256 hash of executable section: a3[...]7d
SHA256 hash of executable section: f7[...]1e

...
Analysis finished, sending SIGINT to process 132
```

We can get the offsets where the executable sections of the '/usr/sbin/dropbear' file are located using readelf:

```
# readelf -S /usr/sbin/dropbear
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
...
  [11] .plt              PROGBITS         0000000000005c00  00005c00
       00000000000009e0  0000000000000010  AX       0     0     16
  [12] .text             PROGBITS         00000000000065e0  000065e0
       00000000000326c8  0000000000000000  AX       0     0     4
...
```

8.3 Finalized Action Workflow for the REWIRE Tracer

This section provides the final, detailed action workflow of the REWIRE Tracer operating in the context of a runtime attestation process. This flow incorporates the design updates detailed throughout this chapter, including the daemon-based implementation, the policy-driven trigger mechanism, and the secure communication channel between the trusted and untrusted worlds. As depicted in the sequence diagram in Figure 8.1, the Tracer supports two primary modalities for monitoring stateful and stateless binaries.

The action workflow for the **tracing of a Stateful Binary (Modality 1)**, which executes once and terminates, consists of the following steps:

1. A remote **Verifier** initiates an attestation process by sending a challenge, containing a fresh *nonce* and a *timestamp*, to the **Attestation Agent (AA)** running within its secure enclave on the target device.
2. The AA, acting on the execution logic dictated by the security policy, triggers the **Tracer daemon**. It does this by writing the name of the target stateful binary into the '.monitor-config' file and then sending a 'SIGINT' signal to the daemon.

3. The Tracer daemon awakens, reads the configuration file, and begins monitoring for the *next running instance* of the specified binary. Using the 'procfs' filesystem, it inspects the process's memory map to identify its executable segments.
4. The Tracer collects the evidence, which consists of the **permissions** and the **SHA-256 hashes** of the identified executable memory segments.
5. The Tracer writes the collected evidence into a dedicated **shared buffer**. It then performs a secure call to the **Security Monitor (SM)** to access its private **Tracer Key (TK)** and signs the contents of the buffer.
6. The Tracer signals the AA to indicate that the signed evidence is ready for collection.
7. The AA performs a secure **ocall**, mediated by the SM, to read the signed traces from the shared buffer.
8. The AA first verifies the signature on the traces using the Tracer's public key (which is bound to the security policy). If the signature is valid, it compares the collected evidence against the expected reference values.
9. If the evidence is correct, the AA uses its own **Attestation Key (AK)** to sign the original challenge from the Verifier. This signed response, often wrapped in a Verifiable Credential (VC), is sent back to the Verifier.
10. The Verifier validates the signature on the response. A successful verification confirms that the Prover device was in a correct configuration state at the time of attestation.

The action workflow for the **tracing of a Stateless Binary (Modality 2)**, which runs continuously, follows a similar pattern but with a different collection logic:

1. A remote **Verifier** initiates an attestation process by sending a challenge to the Prover's **Attestation Agent (AA)**.
2. The AA triggers the **Tracer daemon** by writing the target binary's name to the configuration file and sending a signal. The governing security policy dictates that monitoring should be continuous or periodic.
3. The Tracer begins monitoring the target binary. It collects evidence (memory permissions and hashes) each time it detects an **input trigger** for the binary (e.g., a new request from the AA), as this signifies the start of a new execution loop. This collection continues until a stop signal is received from the AA.
4. Upon receiving a signal from the AA to conclude the attestation, the Tracer consolidates the collected evidence.
5. The Tracer writes the consolidated evidence into the **shared buffer** and uses its **Tracer Key (TK)** via the SM to sign the buffer's contents.
6. The Tracer signals the AA that the evidence is ready.
7. The AA performs a secure **ocall** to read the signed evidence from the shared buffer.
8. The AA verifies the signature and the integrity of the collected traces against the policy's reference values.
9. If the verification is successful, the AA signs the Verifier's original challenge with its **Attestation Key (AK)** and returns the signed response.
10. The Verifier validates the response, confirming the device maintained its integrity throughout the monitored period.

8.4 Test Cases

The foundational capabilities of the REWIRE Tracer—specifically, its ability to monitor a single process by reading its ELF structure and memory segments via 'procfs'—have been successfully implemented and functionally tested. Building upon this baseline, the following unit tests have been defined to validate the more advanced functionality and the correct integration of the Tracer within the REWIRE TCB. These test cases focus on scenarios involving multiple processes and the end-to-end workflow required for the CIV scheme. While these unit tests validate specific capabilities, the Tracer is an integral component of the REWIRE TCB. As such, it will be comprehensively evaluated as part of the complete, integrated framework in the context of the project's use cases. The final results and detailed analysis of both these unit tests and the integrated system performance will be presented in Deliverable D6.2 [28].

Table 8.1: Unit Test UT_TRACE_1: Multiple Memory Segment Monitoring

Test Case ID	UT_TRACE_1
Title	Multiple Memory Segment Monitoring
Description	Test the Tracer's ability to monitor multiple distinct executable memory segments within a single process. Since a single ELF binary can map several executable sections into memory, this test ensures the Tracer correctly identifies all such segments, monitors their permissions for unauthorized changes, and correctly computes the integrity hashes for each one.
Component	REWIRE Tracer
Input	A single target process known to have at least two distinct executable memory segments.
Output	A pass/fail result. A 'pass' indicates the Tracer correctly reports the state (permissions and hashes) for all monitored executable segments.
Status & Results	**[TO BE DONE]** - The tests will be performed and documented in D6.2.

Table 8.2: Unit Test UT_TRACE_2: Multiple Process Instance Monitoring

Test Case ID	UT_TRACE_2
Title	Multiple Process Instance Monitoring
Description	Test the Tracer's ability to correctly monitor multiple running instances (processes) of the same target executable program. This validates that the 'pidof' lookup and subsequent monitoring loop can handle multiple PIDs for a single configured target name.
Component	REWIRE Tracer
Input	A single target program name in the configuration file, with at least two instances of that program running concurrently.
Output	A pass/fail result. A 'pass' indicates the Tracer successfully monitors and reports the integrity of both running process instances.
Status & Results	**[TO BE DONE]** - The tests will be performed and documented in D6.2.

Table 8.3: Unit Test UT_TRACE_3: End-to-End CIV Integration Test

Test Case ID	UT_TRACE_3
Title	End-to-End CIV Integration Test
Description	Test the end-to-end functionality of the Tracer within the complete REWIRE TCB and Configuration Integrity Verification (CIV) workflow. This test validates the entire sequence: triggering by the Attestation Agent, evidence collection and signing by the Tracer, and secure communication of the results back to the Attestation Agent for final verification.
Component	REWIRE Tracer, Attestation Agent, Security Monitor
Input	A valid CIV attestation request initiated by a Verifier, targeting a process monitored by the Tracer.
Output	A pass/fail result. A 'pass' indicates that the entire workflow completes successfully, resulting in a correctly signed attestation response.
Status & Results	**[TO BE DONE]** - The tests will be performed and documented in D6.2.

Chapter 9

Conclusions

Throughout this deliverable, we have detailed the final technical specifications, cryptographic designs, and performance evaluations that constitute the REWIRE Runtime Assurance Framework. This work successfully transitions the project's runtime security vision from the architectural concepts of D2.2 and the initial designs of D4.2 into a complete and validated set of technical artefacts. The framework, anchored in the REWIRE Trusted Computing Base, now stands as a holistic toolkit for enabling, monitoring, and maintaining trust in devices throughout their operational lifecycle.

The final protocols for secure lifecycle management—namely **Secure Software Update** and **Secure Live Migration**—have been fully specified and analyzed, confirming their robustness against a sophisticated threat model. These mechanisms are not merely theoretical; they are supported by TCB-level enablers like the **Mirrored Keys** scheme, whose practical, low-overhead performance for secure state transfer has been demonstrated. This work confirms that resilient computing, a core goal of REWIRE, is achievable even within the complex security model of Trusted Execution Environments by providing verifiable and secure state management.

A central contribution presented herein is the comprehensive evaluation of the **Configuration Integrity Verification (CIV)** scheme. The extensive benchmarking across diverse TEEs—Keystone, Intel SGX, and OP-TEE—has provided crucial empirical evidence of the framework's adaptability. The results have not only quantified the performance trade-offs but have also validated the viability of REWIRE's open-source-first approach. We have demonstrated that while proprietary, hardware-accelerated TEEs currently offer the highest performance, the modular and flexible architecture of Keystone on RISC-V is already highly competitive in the complex policy-based operations that are most critical to REWIRE's innovative attestation model, proving its potential as a foundation for future secure systems.

Furthermore, the cryptographic foundations of the framework have been solidified. The **Zero-Touch Onboarding (ZTO)** process has been finalized into two distinct, powerful flows, supported by a fully analyzed and benchmarked **Attribute-Based Signcryption (ABSC)** scheme. This provides REWIRE with a uniquely flexible mechanism to bootstrap trust, capable of adapting from standard zero-trust principles to a more stringent "Below Zero Trust" posture. The final design of the **REWIRE Tracer**, enhanced with memory introspection and performance caching, provides the efficient, non-intrusive evidence collection necessary to fuel these advanced attestation schemes.

In conclusion, Deliverable 4.3 has successfully delivered a complete, specified, and performance-validated runtime assurance framework. The components and protocols detailed within are now finalized and constitute a stable technical foundation, ready for the final project phase. This work provides the necessary inputs for the integration and end-to-end evaluation within the project's use cases, which will be carried out in WP6 and documented in the final deliverable, D6.2.

Bibliography

[1] About OP-TEE &x2014; OP-TEE documentation documentation — optee.readthedocs.io. https://optee.readthedocs.io/en/latest/general/about.html. [Accessed 24-07-2025].

[2] API Archives - GlobalPlatform — globalplatform.org. https://globalplatform.org/document_types/api/. [Accessed 24-07-2025].

[3] ELF - OSDev Wiki — wiki.osdev.org. https://wiki.osdev.org/ELF. [Accessed 24-07-2025].

[4] GitHub - riscv-non-isa/riscv-sbi-doc: Documentation for the RISC-V Supervisor Binary Interface — github.com. https://github.com/riscv-non-isa/riscv-sbi-doc. [Accessed 28-07-2025].

[5] Intel® Software Guard Extensions (Intel® SGX) — intel.com. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html. [Accessed 24-07-2025].

[6] The miracl core cryptographic library. https://github.com/miracl/core. Accessed: 2025-06-23.

[7] RISC-V Ratified Specifications — riscv.org. https://riscv.org/specifications/ratified/. [Accessed 28-07-2025].

[8] SPDM — DMTF — dmtf.org. https://www.dmtf.org/standards/spdm. [Accessed 24-07-2025].

[9] Ayokunle Micheal Akinsiku. A comprehensive survey of federated learning approaches for privacy-preserving machine learning. *Tech-Sphere Journal for Pure and Applied Sciences*, 2(1), 2025.

[10] Mihir Bellare and Gregory Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *ACM CCS*, pages 390–399, 2006.

[11] Victor Castano and Igor Schagaev. *Resilient computer system design*. Springer, 2015.

[12] Adam Caulfield, Norrathep Rattanavipanon, and Ivan De Oliveira Nunes. On the verification of control flow attestation evidence. *arXiv preprint arXiv:2411.10855*, 2024.

[13] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. In *USENIX Security Symposium (USENIX Security)*, pages 2093–2110, 2020.

[14] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 89–98, New York, NY, USA, 2006. Association for Computing Machinery.

[15] Eliot Lear, Ralph Droms, and Dan Romascanu. Manufacturer usage description specification. Technical report, 2019.

[16] Allison Lewko and Brent Waters. Decentralizing attribute-based encryption. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 568–588, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[17] Zhen Liu, Zhenfu Cao, and Duncan S. Wong. Efficient generation of linear secret sharing scheme matrices from threshold access trees. Cryptology ePrint Archive, Paper 2010/374, 2010. https://eprint.iacr.org/2010/374.

[18] Arm Ltd. TrustZone for Cortex-A — arm.com. https://www.arm.com/technologies/trustzone-for-cortex-a. [Accessed 24-07-2025].

[19] National Institute of Standards and Technology. A framework for designing cryptographic key management systems. Technical Report SP 800-130, NIST, August 2013. Accessed: 2025-06-17.

[20] National Institute of Standards and Technology. Recommendation for key management, part 1: General. Technical Report SP 800-57pt1r4, NIST, January 2016. Accessed: 2025-06-17.

[21] Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Computing Surveys (CSUR)*, 35(3):309–329, 2003.

[22] REWIRE. Rewire operational landscape, requirements, and reference architecture - initial version. Deliverable D2.1, The REWIRE Consortium, 12 2023.

[23] REWIRE. Conceptual architecture of rewire customizable tee & attestation model specifications. Deliverable D4.1, The REWIRE Consortium, 04 2024.

[24] REWIRE. Rewire design time secure operational framework - final version. Deliverable D3.3, The REWIRE Consortium, 30 2024.

[25] REWIRE. Rewire design time secure operational framework - initial version. Deliverable D3.2, The REWIRE Consortium, 04 2024.

[26] REWIRE. Rewire runtime assurance framework - initial version. Deliverable D4.2, The REWIRE Consortium, 04 2024.

[27] REWIRE. Rewire integrated framework (1st release) and use case analysis. Deliverable D6.1, The REWIRE Consortium, 05 2025.

[28] REWIRE. Rewire integrated framework (final release), use case evaluation and project impact assessment. Deliverable D6.2, The REWIRE Consortium, 36 2025.

[29] REWIRE. Rewire operational landscape, requirements, and reference architecture - final version. Deliverable D2.2, The REWIRE Consortium, 24 2025.

[30] Keystone Team. Keystone — keystone-enclave.org. https://keystone-enclave.org/. [Accessed 24-07-2025].

[31] Stefano Tessaro and Chenzhi Zhu. Revisiting bbs signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 691–721. Springer, 2023.

[32] Eric Voit, Henk Birkholz, Thomas Hardjono, Thomas Fossati, and Vincent Scarlata. Attestation Results for Secure Interactions. Internet-Draft draft-ietf-rats-ar4si-08, Internet Engineering Task Force, February 2025. Work in Progress.

[33] John Vollbrecht, James D. Carlson, Larry Blunk, Dr. Bernard D. Aboba, and Henrik Levkowetz. Extensible Authentication Protocol (EAP). RFC 3748, June 2004.